ALMA MATER STUDIORUM — UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA
IN
Ingegneria Elettronica, Informatica e delle Telecomunicazioni

Cycle XXI
Disciplinary Sector: ING-INF/05

SELF-ORGANISING COORDINATION SYSTEMS

Candidate

Dott. Ing. MATTEO CASADEI

Coordinator:

Chiar.mo Prof. Ing. PAOLA MELLO

Supervisor:

Chiar.mo Prof. Ing. ANTONIO NATALI

Co-Supervisors:

Ill.mo Prof. Ing. ANDREA OMICINI
Prof. MIRKO VIROLI

ACADEMIC YEAR 2007–2008

Acknowledgements

I owe more than a big thanks to many people who have been keeping to support me over the past three years: I am not just talking about people who believed in what I was doing, but also about actually incredible guys who have stood for me during one of the most difficult period of my life ever. It should be clear by now that the former class of people mainly refers to colleagues, while the latter one refers to friends. Actually some of the guys of the first class are also part of the second one, as typically happens when one spends great part of his/her time working, quite a natural thing when you enjoy your job, but also very harming for relationships! That is what made me undergo a very painful and hard time: I have been lucky enough to get rid of it. From now on, I will definitely and always keep the right balance between work and private/social life, never sacrificing the latter for the former!

First of all, I must thank Elena, my mate in life and work: she has always been a firm support over the last year, sharing with me everything, from tough and unpleasant situations to those little achievements which make every life meaningful and definitely worth to be lived. I am sure, I will find a way to adequately reward her patience.

My deepest gratitude goes to Ambra, Maurizio, and Luca for the invaluable help and the many discussions, which have greatly contributed to my professional growth and will definitely represent fundamental lessons for my future activity.

It is just impossible to forget Melissa, Martina, and the other guys of the APICe laboratory, who have made the laboratory a friendly and comfortable workplace, lightening the atmosphere whenever required: they have been an important part of my life and, even though I still do not know what the future will look like, I am going to keep them all in my heart.

Last but not least, the most grateful thought goes to my supervisors Antonio, Andrea, and Mirko for their constant support. I think you all have done a good work with me: in the end, I owe you most of my expertise and skills.

Matteo Casadei, March 2009

Contents

Abstract

Many research fields are pushing the engineering of large-scale, mobile, and open systems towards the adoption of techniques inspired by self-organisation: pervasive computing, but also distributed artificial intelligence, multi-agent systems, social networks, peer-to-peer and grid architectures exploit adaptive techniques to make global system properties emerge in spite of the unpredictability of interactions and behaviour. Such a trend is visible also in coordination models and languages, whenever a coordination infrastructure needs to cope with managing interactions in highly dynamic and unpredictable environments. As a consequence, self-organisation can be regarded as a feasible metaphor to define a radically new conceptual coordination framework. The resulting framework defines a novel coordination paradigm, called self-organising coordination, based on the idea of spreading coordination media over the network, and charge them with services to manage interactions based on local criteria, resulting in the emergence of desired and fruitful global coordination properties of the system. Features like topology, locality, time-reactiveness, and stochastic behaviour play a key role in both the definition of such a conceptual framework and the consequent development of self-organising coordination services.

According to this framework, the thesis presents several self-organising coordination techniques developed during the PhD course, mainly concerning data distribution in tuple-space-based coordination systems. Some of these techniques have been also implemented in ReSpecT, a coordination language for tuple spaces, based on logic tuples and reactions to events occurring in a tuple space. In addition, the key role played by simulation and formal verification has been investigated, leading to analysing how automatic verification techniques like probabilistic model checking can be exploited in order to formally prove the emergence of desired behaviours when dealing with coordination approaches based on self-organisation. To this end, a concrete case study is presented and discussed.

Keywords: *Coordination Models and Languages*, *Tuple Centre*, *Self-Organisation*, *Self-Organising Coordination*, *ReSpecT*, *Collective Sort*, *SwarmLinda*, *Tuple Clustering*, *Tuple Sorting*, *Stochastic Simulation*, *Probabilistic Model Checking*, *Multi-Agent Systems*.

1 Introduction

1.1 Motivation and Research Context

Traditional coordination models like LINDA [Gel85], REO [Arb04], and their derivatives, always enact coordination rules that are *predictable*, namely, whose impact on system interactions is known and fully re-producible—of course, they can be non-deterministic though. This is a natural consequence of coordination models and languages starting as a branch of traditional software engineering, which promotes methods of software design with the goal of building correct, efficient, and predictable applications.

However, the emergence of new application scenarios – where dynamism, unpredictability, openness, and large-scale become key system properties – is calling for a radically new approach [ZP03, MMTZ06], as emerging in the context of many research fields like distributed artificial intelligence, multi-agent systems, self-organising systems, and pervasive computing. Today's and tomorrow's networks will increasingly require the ability of communication and coordination services to cope with unpredictable changes, including changes in load, task, physical and logical topology: this can be achieved by the development of self-organising infrastructures [DDF+06], where among the others, peculiar features are *time* and *stochasticity*. Instead of achieving efficiency, optimality, and predictable control over system interactions, new properties like robustness to failures and adaptivity to dynamic changes are instead to be achieved by *emergence*, i.e. as the result of local interactions without any form of global and supervised control [CDF+01]—nature mostly works in the same way in contexts like chemistry, biology, ecology, and so on.

Few works have recently witnessed a similar trend in coordination models and languages as well. In TOTA [MZ04], an infrastructure based on tuple spaces is conceived where tuples can be copied and spread in neighbouring nodes along with a fading mechanism so as to form so-called *computational fields* (co-fields), which can be exploited by agents to find one another (and retrieve data items) in spite of their mobility and changes in network topology. In this infrastructure, *topology* and *locality* of interactions play a crucial role since a distributed data structure of tuples (the co-field) is created on a step-by-step basis.

As regards stochasticity, few formal models have been introduced to tackle stochastic

aspects in coordination models. In stoKLAIM [DNLM05], a formal model extending Linda is described, where agents insert and retrieve tuples in a stochastic way, namely by specifying an operation rate that affects timing and probability of the corresponding primitive execution. Similarly in [BGLZ04, BGLZ05], formal underpinnings for probabilistic extensions of Linda are studied, featuring the ability to specify a probability (or a rate) for the execution of primitives—e.g. for retrieving certain tuples instead of others.

Finally, we note that several works outside the core coordination community are actually addressing the problem of mediated interaction through a self-organising environment. A key example is that of multi-agent-system environments inspired by *stigmergy* [HM99], a technique by which ants coordinate their behaviour. The main idea of this paradigm is that agents leave pheromone-like data items on the local environment that as time passes *(i)* distribute in neighbouring nodes, *(ii)* aggregate, and *(iii)* fade. By properly exploiting such data items spread in the environment, agents can self-organise their behaviour, e.g. by adaptively creating and maintaining a path towards a resource [PBS02]. The potential applicability of this framework to tuple-based coordination infrastructures is clear—e.g. as shown by some experiments based on Tota.

The aforementioned works show that there is an undoubtable interest in the application of self-organising approaches to coordination models and languages, with the implicit goal of achieving adaptivity properties in interaction management, as required by today's scenarios of pervasive computing.

1.2 Overview and Contribution

According to the above premises, the work presented in this thesis is an effort in the direction of finding new ways of dealing with coordination issues in today's software systems, that cannot be coped with by traditional models and approaches to coordination. To this end, the thesis proposes the use of self-organisation as an inspiring metaphor, which has recently proven to be a feasible paradigm for system engineering in computer science, as witnessed by many research fields in software systems [MMTZ06]. Given the emergent and local nature of every approach based on self-organisation, the role of simulation plays a key role for providing an effective design and engineering of self-organising techniques in coordination. Accordingly, this work also deals with modelling and simulating self-organising systems. In particular, the main focus here is on *stochastic* modelling and simulation approaches, which have been already recognised as essential in the early stages of system design—this is for instance highlighted in [Gar08], where it is also remarked that little work exists on simulation applied to early phases of software development [May93, DWHS05, BGP06]. Furthermore, the adoption of simulation needs to be regulated according to a specific methodology for software system design. In fact, the exploitation of design methodologies – already recognised as actually important in traditional software engineering approaches – becomes crucial in self-organising sys-

tem design because of not only the local and unpredictable nature of self-organisation, but also the peculiar dynamics and features of many scenarios where self-organisation is being applied [Gar08, GVCO08]. Correspondingly, the design of every self-organising coordination technique presented here is based on the methodological approach described in [Gar08, GVCO08], focussing especially on simulation, but also on verification. Verification refers to formally proving and validating system behaviour by exploiting automatic verification techniques like model-checking [CGP00]. Moreover, in this thesis the probabilistic extension of model-checking [KNP04] has been exploited to validate some of the proposed coordination techniques.

The activity undertaken over the PhD course is focussed on tuple-space-based coordination as a specific case of data-centric coordination. The resulting self-organising techniques devised, listed in the following, mainly deal with tuple organisation and distribution in networks of distributed tuple spaces, definitely targeted at promoting a novel way of conceiving process/agent interaction.

Collective Sort. Developed in the context of tuple distribution in fully connected networks of tuple spaces, this approach allows to sort tuples according to their kind—kind refers to the type of information contained in the tuple. The resulting distribution features clusters of tuples of the same kind. Collective sort was prototyped by exploiting a stochastic simulation framework realised in MAUDE [CDE+05], which is a modelling language based on rewriting logic [MOM02]. The study of collective sort was part of the activities carried on in the first year of the PhD course and part of the second, resulting in several publications, e.g. [CGV07, VCG07, CVG09].

SwarmLinda. Like collective sort, SwarmLinda focusses on tuple distribution in networks of tuple spaces. However the overall objective here is quite a different one: indeed, SwarmLinda is meant to propose an extension to the original LINDA tuple-space coordination model [Gel85]. In particular, the proposed extension is based on the adoption of swarm intelligence [BDT99], so that the resulting extended LINDA model can be adopted to promote a novel self-organised and decentralised approach to tuple distribution and retrieval. In particular, the work focussed on tuple distribution by providing the traditional `out` operation of LINDA with new semantics. The resulting operation behaviour promotes cluster formation in any-topology tuple-space networks, so that tuples containing similar information can be stored in the same area of the network. This can lead to a better system scalability, as well as ease tuple retrieval by processes/agents. The research on SwarmLinda has been done in collaboration with Professor Ronaldo Menezes of Florida Institute of Technology (FIT), as a part of an abroad research period spent at FIT during the second year of the PhD course. The main results of this work are reported in [CMVT07a, CMVT07b, CMVT07c, CMTV07]. As for collective sort, the prototyping of SwarmLinda was based on the use of the stochastic simulation framework developed in MAUDE.

Tuple Clustering and Tuple Sorting. These approaches can be seen as a generalisation of collective sort from a twofold perspective: *(i)* first of all, as a way of dealing separately with the issues of tuple clustering and sorting; *(ii)* secondly, as a brand-new version of collective sort aimed at being applied on networks irrespectively of their topology. Again, as in the case of SwarmLinda, the main inspiration came from swarm intelligence, in particular from *corpse clustering* and *larval sorting* in ants [BDT99, DGF+91, CDF+01]. The results of this activity – carried on in the last part of the PhD course – are reported in [CVS08, CV08]. The prototyping was performed by NetLogo [Wil08], an agent-based simulation framework. Tuple clustering resulted also in an implementation based on ReSpecT, a logic, reaction-based language for programming tuple spaces [Omi07].

In addition, some experiments were performed, regarding the adoption of ReSpecT as a sample language to define and concretely implement self-organising coordination strategies. Some of these experiments focussed on devising biochemistry-inspired coordination strategies. The early results of this activity can be found in [VCO09].

Finally, as an activity undertaken also over the last part of the PhD course, we focussed on issues related to formal automatic verification of self-organising systems with respect to coordination. To this end, the adoption of probabilistic model checking was investigated as a means to allow the verification of emergent and desired coordination behaviours on self-organising coordination systems. In this context, some fundamental questions exist that still need to be fully answered:

- *(i) What is a feasible approach for efficiently devising models of self-organising coordination techniques to be later verified by model checking?* Answering this question is key to cope with the so called state-space-explosion problem, which affects current model-checking techniques.
- *(i) How can the properties to verify be specified according to existing probabilistic temporal logics?* Answering this question requires to consider the related issues from a twofold perspective: first of all, it is necessary to identify a valid measure for the coordination behaviour to be verified, then it is crucial to translate such a measure into a property by exploiting appropriate temporal logic operators.

It is worth noting that this research was not aimed at developing an accurate methodology for verification of self-organising coordination systems. In fact the main goal was just to experiment with probabilistic model checking and assess its suitability for emergent-property verification on self-organising coordination systems. As such, some verification experiments on collective sort were performed: the corresponding results are reported in [CV09b, CV09a], which also give some suggestion as regards possible ways of dealing with state-space-explosion problem by relying on approximated model-checking techniques. Note that the aforementioned questions still need to be answered in full detail:

indeed, as it should be clear by know, the work on verification is based on a quite pragmatic premise, trying simply to experiment with probabilistic model checking and the corresponding issues.

1.3 Organisation of the Thesis

The rest of the thesis is structured as follows.

Chapter 2, Background. This chapter provides the necessary background to understand the case studies presented in the rest of thesis. In particular, after giving an introduction of what self-organisation and emergence are all about, the abstract framework for self-organising coordination is presented by defining a set of founding principles. In order to better understand the process underlying the development of the self-organising techniques shown in Chapters 3, 4, and 5, an introduction to stochastic simulation for self-organising system design is also provided.

Chapter 3, Collective Sort. The collective sort strategy is here presented in full detail, starting from the development of the formal model and its simulation to the discussion of an improved strategy for complete sorting and a comprehensive analysis of the simulation results.

Chapter 4, SwarmLinda. As for collective sort, this chapter provides a in-depth analysis of the research done on SwarmLinda. The formal model of SwarmLinda is presented, and the results obtained from stochastic simulations carefully analysed and discussed.

Chapter 5, Self-Organising Tuple Clustering and Sorting. Tuple clustering and tuple sorting can be though of as a generalisation of collective sort targeted at networks featuring non-specific topologies. The main inspiration for these strategies came from swarm intelligence. The modelling framework adopted was NetLogo. Again, the early simulation results are presented and analysed.

Chapter 6, Self-Organising Coordination by TuCSoN and ReSpecT. In this chapter, the ReSpecT language and the corresponding framework for tuple centres TuCSoN are exploited for implementing self-organising coordination techniques. To this end, an implementation of tuple clustering in ReSpecT is presented. Furthermore, it is also shown how ReSpecT can be used to prototype self-organising coordination techniques inspired by biochemistry.

Chapter 7, Formal Verification of Self-Organising Coordination Systems. This chapter reports some experiments regarding formal-property verification in self-organising coordination systems by probabilistic model checking. After describing the research context and related issues, collective sort is taken as a case study and

the emergence of complete sorting verified by adopting PRISM, one of the most used probabilistic model checkers.

Chapter 8, Related Work. This chapter discusses related work, focussing on current trends and approaches in coordination inspired by self-organisation.

Chapter 9, Conclusion. This chapter concludes the thesis, summing up the activity of the PhD course, providing final remarks, and discussing possible future works.

2 Background

This chapter provides the necessary background to understand the rest of the thesis. First, the notions of *self-organisation* and *emergence* are briefly recalled, then an introduction to *self-organising coordination* is provided and the resulting conceptual framework discussed. The chapter concludes by reporting a brief introduction to stochastic simulation and the corresponding simulation framework adopted for devising the self-organising coordination services presented in the rest of the thesis.

2.1 Self-Organisation and Emergence

The term *self-organisation* originally arose from research in physics, chemistry, and related fields, following the idea that there exist systems which are able to increase their inherent order by themselves as a result of their very same dynamics. It is generally agreed that the concepts underlying self-organisation were first introduced by a French philosopher of the XVII century, René Descartes, in [Des37]. Indeed in [Des37], the philosopher argues on the existence of God, and the fifth part of the book gives an example of how matter created and chaotically shaken by God can self-dispose and arrange itself as a result of the same laws established by God himself.

However, the first explicit and much more recent definition of self-organisation is due to William Ross Ashby, a psychiatrist who, in 1947, defined self-organisation as the *ability of a systems to modify by itself its internal organisation without any intervention of external forces* [Ash47]. Later, the term gained more attention as a result of its adoption by physicists in the field of complex systems. The last 30 years have witnessed a growing attention to self-organisation, which has led to the adoption of the underlying principles in many fields, ranging from social to life sciences. In fact, self-organisation has been adopted not only in physics in areas such as structure formation in thermodynamic systems, but also in fields like chemistry (e.g. in areas related to molecular self-assembly and reaction-diffusion systems) and biology. In particular, in the latter context, research on collective behaviour – focussing on creation of structures and coordination in social insects, as well as flocking behaviour such as fish schooling and bird flocking – has driven the development of a new branch of artificial intelligence called *swarm intelligence*. Swarm intelligence

[BDT99] refers to the application of the aforementioned principles to artificial intelligence so as to affect with novel ideas fields such as robotics, operational research, computer science, and so on. Swarm intelligence has been with no doubt a great source of inspiration for the work presented in this thesis.

Along the same line, the book by Scott Camazine [CDF+01] has also been really inspiring. In [CDF+01], Camazine gives what is generally considered a key definition of self-organisation in biological systems, stating that *self-organisation refers to a process whereby patterns at the global level of a system emerge as the exclusive result of the many interactions among system components. In addition, the rules regulating such interactions rely only on local information without any correspondence to the pattern observable at the global level.* As made it clear in the coming sections, the above definition contains all the key ingredients upon which the concept of self-organising coordination is based. Camazine's definition allows to further argument about four key characteristics that usually recur in any self-organising system: *(i) autonomy* referring to the fact that the evolution of the system is controlled by the system itself, without any intervention of external forces; *(ii) organisation* meaning that self-organising systems usually work so as to continuously re-organise themselves; *(iii) dynamics* referring to the fact that self-organisation requires to study a system from the standpoint of its dynamics and not only from a static viewpoint; *(iv) adaptation* meaning that self-organising systems are able to automatically adapt their behaviour to ongoing perturbations occurring in the environment where they live.

Camazine's definition of self-organisation is clearly based on the concept of *emergence*. The term emergence comes from *emergent*, a term coined by George Henry Lewes in a well-known definition [Lew04], stating that the difference between resultants and emergents in chemical reactions lies in the fact that while resultants are reducible to their reactants, the same is not true for emergents, which instead show properties that cannot be explained in terms of the composing entities. The above definition is quite close to the holistic view arguing that sometimes the properties of a system cannot be understood as the mere sum of the properties of its composing entities. This view of emergence is sometimes referred to as *strong emergence* in contrast to *weak emergence*, meaning that new properties of a system can be achieved as a result of elementary interactions among components of the system.

Though it is undoubtably true that self-organisation and emergence are strongly related concepts, there is some confusion about the two terms: indeed, it is easy to get misled by the available literature and convince oneself that self-organisation only occurs as a result of an emergent process, even though emergence is not a necessary condition for self-organisation. Self-organisation as well is not a necessary condition for emergence, as there exist emergent processes which are not self-organising at all. However, emergence is coupled with self-organisation throughout the thesis, and refers to the feature of the proposed coordination services to show global desired behaviours resulting from simple and local interactions among the composing entities of the systems: in other words, according

to the definition given for weak emergence, the global behaviour *emerges* from the local interactions among the components of the system.

2.2 Towards a Coordination Model Based on Self-Organisation

It is a matter of fact that many research fields are pushing the engineering of large-scale, mobile, and open systems towards the adoption of self-organisation techniques: pervasive computing first, but also distributed artificial intelligence, multi-agent systems, social networks, peer-to-peer and grid architectures exploit adaptive techniques to make global system properties *emerge* in spite of the unpredictability of interactions and behaviour. We observe that a similar trend is likely to affect the field of coordination models and languages as well—in particular, whenever a coordination infrastructure is charged with the task of managing interactions in such highly dynamic and unpredictable systems. This trend is actually witnessed by some previous works on extending coordination models with quantitative aspects like timing [ORV05] and probability [BGLZ04, BGLZ05, DNLM05], implementing coordination infrastructures by exploiting nature-inspired techniques [MT04, MZ04], and devising self-organising services for managing data-centric coordination systems [CGV07]. Moreover, academic and industrial attempts to devise infrastructures for mediated interactions are converging towards the adoption of self-organising mechanisms, as in the case of *stigmergic* infrastructures [PBS02].

In order to analyse more deeply the links between coordination and self-organisation, and understand properties, requirements, and opportunities for their cross-fertilisation, one of the objectives of the thesis is to propose a conceptual framework for *self-organising coordination* as a novel paradigm to help conceive and implement real systems. This is based on the idea of spreading coordination media over the network, and charge them with services to manage interactions solely based on local criteria, resulting in the *emergence* of interesting and fruitful global coordination properties of the system. Namely, spatial and time patterns will occur in the coordination space in spite of the unpredictable interactions of agents exploiting such coordination services. By drawing inspiration from previous works and self-organising mechanisms in nature, the next section highlights the key role of topology, locality, time-reactiveness, and stochastic behaviour in the development of self-organising coordination services.

2.3 Self-organising Coordination

This section discusses a reference conceptual framework for self-organising coordination. First some of the expected features for this meta-model are analysed.

Topology — Being the target of scenarios like pervasive computing, multi-agent systems, Internet computing, and distributed artificial intelligence, we assume that the application at hand is deployed over a topologically-structured distributed system; namely, each location (i.e., each node) is directly connected to a generally small set of neighbouring locations—this is actually a quite general model, useful to define the basic terminology for subsequent discussions. Coordination media and agents are deployed over locations. As usual, while each agent resides over a single location, coordination media might be distributed over a connected subset of the system—though this is not a mandatory feature.

Locality — Topology is strictly tight with the scope of interactions. A coordinated system can feature two kinds of interaction, between an agent and a coordination medium, and between two coordination media. Note that the latter is not a mandatory one: most models do not provide inter-medium interactions but simulate it by using an agent-in-the-middle approach—with few exceptions like e.g. TuCSoN [COV08]. Then, both kinds of interaction should occur *locally*, i.e. across the same location or across two neighbouring locations as defined by the topology. In other words, agents will necessarily have a partial (local) view of the system, and long-path interactions should occur on a step-by-step basis—both in space and time.

On-line character — Coordination media typically enact coordination rules that are reactive to interactions, that is, coordination rules fired as some interaction occurs. With self-organising coordination instead, it is typical to feature coordination rules that can also react to time passing, namely, a "coordination behaviour" is enacted as an always-running process. This can be seen as an *on-line* and background service that, other than affecting interactions, also evolves the state of coordination media through time. This process truly couples agents autonomous behaviour with a carefully balanced definition of system behaviour. For instance, in stigmergic coordination the fading mechanism should be properly tuned to effectively work: the rate at which pheromone should fade strictly depends on the rate at which agents move and work.

Time — As the above on-line property implies, the process of self-organising coordination is strongly dependent on time—as any other self-organising process in nature [PS97]. In particular, coordination rules are generally timed. On the one hand, the self-organising coordination service must be provided at a certain "rate" (a given part of the work is to be done within a time unit): this could for instance be achieved by making coordination rules firing other ones after sometime has elapsed, in a cyclic way. Similarly, certain coordination primitives can also be time-dependent, as typically necessary in open contexts like Internet computing [ORV07]—e.g. a coordination primitive may fail as a result of a deadline expiration.

Probability — Non-determinism is a typical coordination pattern, exploited as a powerful abstraction principle to separate modelling and implementation. For instance, in LINDA *any* tuple can be read that matches a specified template: however, this might cause the same tuple to be retrieved, ultimately leading to an unfair management of

resources. Self-organisation – originated in the fabric of nature and under its laws – works in a different way. A self-organising process is typically composed of a huge amount of instances of the same atomic action, each time leading to different effects—for instance, noise is known to play a crucial and necessary role in self-organising systems [NPL75]. The only means to describe the resulting behaviour is hence by stochastic models. It should be specified that some results are more likely than others, furthermore, high adaptiveness to unpredictable situations implies the need to give even a very low probability to certain behaviours—which might eventually have a dramatic impact, e.g. causing a failing system to recover.

According to the above properties, we can define self-organising coordination as follows:

> *Self-organising coordination* is the management of system interactions featuring self-organising properties, namely, where interactions are local, and global desired effects of coordination appear by emergence.

Constructively, self-organising coordination is achieved through coordination media spread over the topological environment, enacting probabilistic and time-dependent coordination rules.

2.4 Stochastic Simulation for Self-Organising Systems

Stochastic simulation has been used as the main tool for designing and testing the behaviour of all the self-organising coordination services presented in the thesis. To this end, we relied on a stochastic simulation framework based on the work by Gillespie [Gil77] and developed in the MAUDE term rewriting language, which is meant to speed up the process of modelling and simulating stochastic systems. Here only some features of this framework are briefly described.

MAUDE is a high-performance reflective language supporting both equational and rewriting logic for specifying a wide range of applications [CDE+05]. Other than specifying algorithmic aspects through algebraic data types, MAUDE can be used to provide *rewriting laws* – i.e. transition rules – that are typically used to implement concurrent *rewriting semantics*, and then able to deal with aspects related to system interaction and evolution. In the course of finding a general simulation tool for stochastic systems, MAUDE seemed a particularly appealing framework, as it allows to directly model system structure and dynamics, or prototype new domain-dependent languages to have more expressiveness and compact specifications.

Inspired by Priami's work on stochastic π-Calculus [Pri95], MAUDE was so exploited to realise a general simulation framework for stochastic systems, which does not mandate a specification language as e.g. π-Calculus, but is rather open to any language equipped with a stochastic transition system semantics. Correspondingly, a system can now be modelled as a LTS (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, i.e. its occurrence frequency. This idea generalises the activity mechanism of stochastic π-Calculus, 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: in fact the way global rate is computed is custom, and ultimately depends on the application at hand—e.g. the global rate can be fixed, or depend on the number of system sub-processes willing to execute the 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 higher the probability for the action to occur); *(iii)* changing accordingly system state; and finally *(iv)* increasing the time counter according to an exponential distribution, so that the average frequency is the sum of action rates.

As an example, consider the $Na - Cl$ chemical reaction dynamics, provided e.g. in SPiM documentation[1]. Syntax and semantics of this system is expressed in the MAUDE framework as follows:

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

vars Na Na+ Cl Cl- : Nat .
eq < Na,Na+,Cl,Cl- > ==> =
   ( ion   # (float(Na * Cl) * 1.0)   ->
                [< p Na,s Na+,p Cl,s Cl- >]  );
   ( deion # (float(Na+ * Cl-) * 2.0) ->
                [< s Na,p Na+,s Cl,p Cl- >]  ) .
```

This system is characterised by a state expressed as `<Na,Na+,Cl,Cl->`, where `Na` is the number 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: `ion` stands for ionisation and `deion` for deionisation. The actual transition system is specified by a single equation, associating any state to 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`), the other that behaves instead in the opposite way. Note that, according to Gillespie's selection algorithm in [Gil77], the rates of ionisation and deionisation are here proportional to the product of the two reactants, multiplied by a constant value: in particular, here the factor of deionisation is as twice as that of ionisation. By a command of the kind

[1] SPiM, Stochastic-Pi Machine, is a stochastic simulation framework based on stochastic π-Calculus: for further details, please refer to `http://www.doc.ic.ac.uk/~anp/spim/Chemical.pdf`

```
rewrite [300: <100,0,100,0> @ 0.0]
```

the system yields a trace of 300 steps, starting from state `<100,0,100,0>` and starting time `0.0`; an example of such trace is:

```
[300 : < 100,0,100,0 > @ 0.0],
[299 : < 99,1,99,1 > @ 5.2282294378567067e-5],
[298 : < 98,2,98,2 > @ 6.9551290710937174e-5],
[297 : < 97,3,97,3 > @ 8.5491215950091466e-5],
...
[3 : < 57,43,57,43 > @ 4.0424914101137542e-2],
[2 : < 58,42,58,42 > @ 4.0506028901053114e-2],
[1 : < 59,41,59,41 > @ 4.0661029058233995e-2],
[0 : < 60,40,60,40 > @ 4.0695684943167353e-2]
```

This output can be easily exploited to trace charts of the most relevant quantities for the application at hand.

The actual implementation details of the simulation framework are not discussed here: the interested reader can refer to Appendix A, where the whole MAUDE specification of the stochastic simulation engine is reported.

3
The Collective Sort Problem

In systems coordinated by a distributed set of tuple spaces, it is crucial to assist agents to retrieve the tuples they are interested in. This can be achieved by devising sorting techniques that group similar tuples together in a common tuple space, so that the position of a tuple can be inferred by similarity. Accordingly, this chapter formulates the *collective sort* problem for distributed tuple spaces, where a set of (manager) agents is in charge of moving tuples so as to achieve a complete sorting, namely, each of the N tuple spaces aggregates just tuples belonging to exactly one of the N available kinds. After pointing out the requirements for effectively tackling the problem, a self-organising solution resembling *brood sorting* performed by ants is proposed. In particular, the solution is based on simple agents performing partial observations and accordingly taking decisions on tuple movement. Convergence is addressed by a fully adaptive form of simulated annealing, based on *noise* tuples inserted and removed by agents on a need basis so as to avoid sub-optimal (partially complete) sorting. Emergence of sorting properties and scalability are evaluated through stochastic simulations performed by the stochastic-simulation engine written in MAUDE. The main inspiration for collective sort comes from [CGV07, VCG07]: however, the content of this chapter is in great part based on [CVG09].

3.1 Introduction

Among many scenarios relying on coordination models and languages, the most popular is based on the idea that agents in a distributed system can interact with each other through tuple spaces spread over the network, where tuples can be inserted and retrieved relying on so-called generative communication [Gel85, Gel89, OZ99]. This approach has been shown to support time and space decoupling, as well as promote a clear separation between the computational part of the system, which should stay inside agents, and the coordination part of the system, implemented through tuple spaces.

In open systems, however, due to the unpredictability of agents' behaviour, it is often difficult to know in which tuple space a certain tuple may occur. As a consequence, when an agent needs to retrieve tuples matching a given pattern, the only strategy would be to randomly select one tuple space among the available ones, and try another one

in the case the tuple is not found—leading to obvious performance issues. Accordingly, a strategy is required to assure that agents have some knowledge about the location of the tuples of interest, so that such tuples can be more quickly retrieved. A general solution to this problem is to devise approaches for moving tuples to the most proper tuple space, and locate them accordingly. Works like the TOTA middleware [MZ04] and stochastic KLAIM [DNLM05], though starting from different perspectives, all develop on the idea of extending the basic tuple space model of LINDA with features related to tuple repositioning by moving or copying.

Collective sort develops along the idea of providing a sorting technique for tuple spaces, where a set of *sorting agents* is in charge of moving tuples until each space holds only tuples of the same *kind*. To support this behaviour, sorting agents – which are part of the infrastructure providing the coordination service – must be designed so as to agree on where to sort (the set of N tuple spaces subject to ordering), and how to sort (a clustering relation that groups the tuples of interest into N kinds, so that 1 kind can be exactly associated to 1 space). Similarly to sorting in standard data structures like e.g. arrays, such an aggregation technique – a case of *segregation* in the context of collective robotics [MHH98] – can be regarded as an approach with the ultimate goal of simplifying the process of finding tuples: if a certain tuple is eventually found in a tuple space, then any tuple of the same kind can be found in the same space. Note that even if full sorting is not completely reached, partial sorting may still improve performance of tuple retrieval since the probability of finding the tuple at the first attempt becomes higher than $1/N$. This technique is hereafter referred to as *collective sort* for distributed tuple spaces.

Such a sorting service is meant to work in "background" to the standard activity of tuple spaces, that is, tuple sorting proceeds while *user agents* coordinate their activity by inserting and retrieving tuples. In other words, as an online service. Unlike standard sorting techniques, here sorting should effectively work in dynamic and unpredictable scenarios where user agents keep moving, inserting, and dropping tuples. Therefore, the tuple space that will eventually aggregate a certain kind of tuples is not known statically: it is chosen implicitly and probabilistically as tuples start aggregating in a space rather than another as a consequence of multiple tuple movements. Hence, the main concern here regards robustness and reactiveness to changes rather than moving rate: sorting needs to be a property emerging in spite of external interactions—of course, the more the external environment keeps altering tuple configuration, the more resources must be devoted to sorting if convergence is to be achieved.

By looking at existing systems, we see that interesting related behaviours already manifest in Nature. Ants use a self-organising technique called *brood sorting* to solve a similar problem [BDT99]: they move items (brood or larvae) based on local and partial criteria, and sorting emerges as a global system property. The solution to collective sort proposed here is precisely inspired by brood sorting. Interestingly, full sorting can be achieved by requiring just sorting agents having neither specific computational ability (intelligence or memory) nor a complete visibility of tuple spaces—though, such aspects of

course impact performance. The proposed solution is based on the following ingredients: *(i)* probabilistic access to tuples in tuple spaces; *(ii)* local decision on tuple movement based on a couple of observations (two read operations, on two spaces); and *(iii)* avoidance of non-optimal sorting by a fully adaptive approach in the style of *simulated annealing* [KGV83].

As for any self-organising technique, probability is a key aspect. Not only small variations on tuple configuration can lead to completely different system behaviours, but the same can also happen in different system "runs" from the same initial conditions—since tuples are retrieved probabilistically. Therefore, we shall rely on stochastic simulation tools in order to check whether the proposed solution meets our expectations in term of quality. To this end, we adopt the stochastic simulation library developed in the MAUDE term rewriting system [CGV07] and described in Chapter 2—of course other tools like e.g. SPIM [Phi06], SWARM [swa06] and REPAST [rep06], could be used as well.

3.2 Collective Sort

3.2.1 Motivation

A network composed of tuple spaces and agents is assumed, where agents address tuple spaces by identity. Agent interaction relies on so-called generative communication, namely, agents put tuples (records of primitive values) in tuple spaces and later retrieve such tuples by content, that is, using a partial specification known as *tuple template*.

The presence of a tuple in a particular space may affect the behaviour of the overall system since it reifies the occurrence of an event related to system coordination. Such events may include: *(i)* agents requesting services of some kind provided by other agents; *(ii)* agents providing the outcome of the execution of a service; *(iii)* agents depositing data values which are part of the overall system state; *(iv)* agents publishing part of their internal state and knowledge; and *(v)* agents updating some shared variable upon which other agents synchronise their activity. When storing a tuple, the choice of the tuple space to be used is critical. If an agent knows the identifier of the tuple space where a specific tuple is stored, the tuple can be retrieved by only one read operation; however, if this is not the case, the agent may end up trying different tuple spaces until finding the right one. Therefore, an agent needs to be aware of the location of the tuples it is interested in; if this is not possible, even just some kind of awareness could be helpful.

In standard data structures, like arrays, access to data is simplified – i.e. quicker – by keeping information sorted: typically, a sorting algorithm is exploited along with insertions and removals that preserve sorting, and fast searching operations are conceived so as to leverage sorting. Sorting itself is based on a total order relation over data that is predefined and static.

The idea here is to take a collection of tuple spaces, initially hosting tuples stored in a completely random manner, and apply a similar approach. Since a tuple space is an

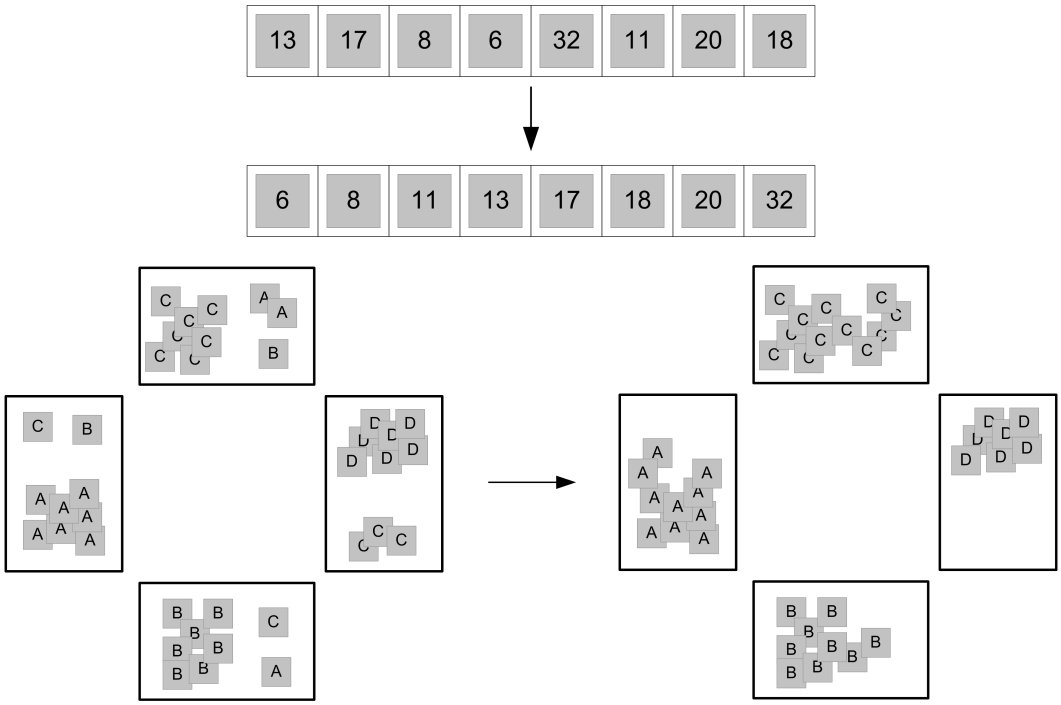


Figure 3.1: Array Sorting vs. Collective Sorting.

unstructured and unbounded bag (i.e. a multiset), the only relevant information for an agent is in which tuple space a tuple is located—there is no notion of "position" of a tuple within a tuple space. Accordingly, our goal is to devise a sorting procedure that moves tuples from one space to another until each space holds only tuples of the same *kind*. If a tuple of kind k is known to reside in space s, all tuples of the same kind can be expected to be located in s, thus simplifying searches.

Given the goal of ordering N tuple spaces, we hence assume that the tuples to be sorted (which might be a subset of all the tuples in the system) are clustered into N kinds, and sorting agents are aware of such clustering. As for the total order relation used in standard structures like arrays, our order relation may be predefined, that is, conceived at design-time and before actually powering on the sorting service. On the other hand, it would be possible to also change the sorting configuration dynamically – due to changes in the set of spaces, or in case of the adoption of a more symmetric clustering relation – but a mechanism for making sorting agents aware of this change is necessary. Hence, the focus here is not on how kinds are formed: in principle, tuples which are "similar" according to some metric, depending on the coordinated system at hand, should belong to the same kind—however, as far as tuple retrieval is concerned, a kind is better composed of all the tuples that match one or more specific templates. See Figure 3.1 for a pictorial comparison between standard array sorting and collective sorting in tuple spaces.

Accordingly, the problem of gathering tuples of the same kind over a set of N tuple

spaces is named as *collective sort*. It should be clear from now on that, unlike standard algorithms, collective sort is meant to work *at runtime*. While it is widely known that the content of an array is frozen during sorting, tuple spaces work in unpredictable dynamic scenarios, so that collective sort needs to be conceived as a background activity aimed at moving the system towards sorting as user agents keep changing the state of tuple spaces.[1] Depending on the ratio between rate of changes and resources devoted to sorting, the system might evolve according to one of the following three behaviours: *(i)* full sorting is achieved (modulo a small noise due to mutations operated by user agents), *(ii)* a certain level of (partial) sorting can be maintained, and *(iii)* the system becomes more and more unsorted as time passes, until becoming chaotic—in next section we shall quantify the degree of order in terms of entropy. It is worth noting that agents can take advance of a partially sorted system as well, since the probability of finding a tuple of kind k, where a previous one was found, is indeed higher than in other tuple spaces. Accordingly, the average retrieval cost is lower than in fully unsorted cases.

3.2.2 Seeking for a Self-Organising Solution

Collective sort in distributed tuple spaces is reminiscent of a classical problem in robotics known as *segregation*, where robots roam the ground with the goal of finding, grouping, and separating items—for further details refer to related works in Chapter 8.

In that context, solutions are typically searched in Nature, which is a rich source of simple but robust strategies. The segregation behaviour has already been observed in social insects and referred to as *brood sorting* [BDT99]. When organising brood and larvae, ants tend to group and keep such items separated from an initial situation where they are randomly situated in the ground. Although ants' actual "behaviour" is still not fully understood, there are several models that are able to mimic the dynamics of the system. Ants wander randomly on the ground and their behaviour is modelled by two probabilities, respectively, the probability of picking up P_p and dropping P_d an item, which are evaluated with respect to the recently encountered items. The idea is that an ant *(i)* picks up an item if its concentration is low with respect to previous experience, *(ii)* starts wander randomly, and *(iii)* drops the item where its concentration is higher with respect to where it was originally picked up.

Brood sorting is intrinsically *self-organising* [CDF+01], i.e.: *(i)* it is a process in which a pattern at the global level of the system *emerges* just as the result of the numerous interactions among the low-level components of the system, and *(ii)* the rules specifying interactions among system's components are executed by using only local information, without any reference to global pattern. Namely, ants are guided by spatially local observations and motivated by the only need of picking items up where concentration is

[1] One could think of an analogy with concurrent garbage collection, where user processes keep mutating data while the collector tries to manage memory in parallel.

low, dropping them where concentration is higher: such numerous interactions make full sorting (i.e. the segregation pattern) emerge at the global level.

The above solution to brood sorting (and any self-organisation-based approach in general) manifests interesting features. First, it is intrinsically robust, since it does not require global information: it promptly reacts to changes in the environment (e.g. new brood, larvae, or ants are dynamically added or removed), to faults like environment splits (e.g. a barrier splitting the ground in two parts), and to local malfunctioning (e.g. some ant behaving in a completely different way than expected). Second, it is intrinsically probabilistic since in the real world small fluctuations always happen that, due to bifurcation effects, might cause the system behaviour to globally change—this is indeed a source of robustness. As for all self-organising approaches, performance is of course worse than solutions based on global observations, due to the overhead caused by the need of continuously performing pointwise (local) observations. Hence, self-organising systems stress the tradeoff between performance of global approaches and robustness of local approaches—consider that global approaches are not always available or feasible, as for distributed systems in general. Self-organising solutions are therefore considered interesting for developing robust online services that should function in unpredictable environments, where performance and robustness live together as key factors.

As a result, it is interesting to seek for a solution to collective sort inspired by ants' brood sorting. However, it should be noted that the two application scenarios have key differences, that might require a significant adaptation:

- Topological space — Instead of being a continuous environment, or a network of connected subparts of the environment, our scenario features a flat set of N tuple spaces, each being a conceptually unbounded bag of tuples.

- Agent Mobility — Instead of being performed by mobile agents carrying items and wandering randomly the environment, the sorting service is executed by an infrastructure composed of software agents, each associated with a tuple space. Since sending tuples is typically less expensive than moving software agents, such agents should not be likely to move.

- Actions and Perceptions — Instead of perceiving items based on a range of locality, software agents should be able to look for tuples in either the tuple space they are assigned to (called local tuple space), or a different tuple space (called remote tuple space)—the latter operation is necessarily more expensive. Similarly, actions correspond to removing tuples and inserting them elsewhere.

3.2.3 Architectural and Behavioural Constraints

Based on the above considerations, a working solution for the collective sort problem is developed that could achieve the robustness properties sought by self-organisation ap-

proaches. First of all, it is important to characterise all the architectural and behavioural constraints for any candidate solution:

- Service Architecture. In a system with N tuple spaces, and multiple *user agents* coordinating their activity through those spaces, collective sort needs to be considered as an online service provided by one or more *sorting agents*, each assigned to exactly one of the N spaces. Each agent works at a certain rate, that is, it executes a number of instances of an interaction protocol per time unit. Of course, it is understood that sorting performance is directly dependent on the sorting rate of such agents.

- Sorting Agent Behaviour. The behaviour of sorting agents, and ultimately of the proposed algorithm, is hence described in terms of a (possibly probabilistic) protocol to be executed multiple times by each agent—most likely, taking an observation and accordingly performing some actions, inspired by brood sorting. This protocol is to be the composition of primitive operations over tuple spaces, that is, reading, removal, and insertion of tuples. Note that tuple counting is not allowed by standard tuple space systems, hence the only means for observing a space is to repeatedly read single tuples on it. Some memory and limited symbolic ability might be assumed but this is not mandatory—e.g. the agent may remember what the last moved tuple is, or have the ability to check whether two tuples belong to the same kind.

- Data Modelling. We assume there is a strict connection between the notion of kind and that of tuple template, so that it is easy for an agent to get a tuple of a certain kind from a tuple space by asking for a tuple matching a template, or get a tuple of *any* kind by asking for a more generic template. Whereas LINDA does not allow to look for tuples matching one of n templates, this is not a conceptual or technological problem per se, and can in fact be implemented over existing tuple-based infrastructures such as TuCSoN [OZ99, OD01]—hence, in the collective sort solution we identify the concept of kind with that of tuple template. Moreover, we assume that reading or removal of tuples matching a given template is a *uniform* operation that yields a probabilistically fair result [VCG07], namely, among many tuples matching a given template, the probability for a specific tuple to be chosen is the same as others.

3.2.4 Quality Attributes

Other than architectural and behavioural requirements, which are meant to shape the structure of a solution, it is also interesting to point out the quality attributes expected from a successful solution, expressed in terms of qualitative and quantitative aspects.

As a way of measuring the degree of sorting of a certain configuration of tuples, we rely on Shannon *entropy* [Sha48], which represents the uncertainty in the observation of

a random variable—this is also called information entropy, or simply entropy from now on. This is expressed as $K * \sum_{i=1}^{n} p_i log(p_i)$, where K is any constant value and p_i is the probability for the variable to assume the i^{th} of n possible values. In our case, the ordering of a tuple space can be associated with the variable that represents the kind of a tuple randomly drawn in the space. Given N kinds $k_1 \ldots, k_N$, if we denote with q_k the amount of tuples of kind k, and with n the total number of tuples in the space, then the probability of a tuple to be of kind k_i is q_{k_i}/n, which is basically the concentration c_{k_i} of tuples of kind k_i in the space. Accordingly, the entropy associated with tuple space s can then be computed as:

$$H_s = K * \sum_{i=1}^{N} c_{k_i} \log(c_{k_i}) \tag{3.1}$$

while the global system entropy is simply the sum of each space entropy, namely $H = \sum_{i=1}^{N} H_{s_i}$, which ranges from 0 to $K * N \log_2 N$. K is then set to $1/(N \log_2 N)$ in order to bound the global system entropy between 0 and 1, where 0 means complete sorting (each space holds tuples of one kind), while 1 means complete disorder (each space has an equal concentration of tuples per kind).

Concerning the outcome of ordering, the quality attributes we seek for the collective sort solution can hence be listed as:

- Full Sorting. In case of a quiescent system – one in which the *mutation rate* of user agents is zero – complete sorting ($H = 0$) must be reached from any initial configuration. Namely, system evolution should never get stuck into unordered states (where $H > 0$).

- Reactiveness. Given a certain non-zero mutation rate, and a desired level of sorting $H_D < 1$, there should be a sorting rate leading the system to an entropy value constantly lower than H_D.

Another key issue concerns the performance which can be achieved in sorting. Being an online service, we note that sorting time strictly depends on the resources devoted to sorting. Supposing a fixed number of network operations devoted to sorting per time unit, performance can be characterised in terms of the amount of network operations (reading, removal or insertion of tuples) required to achieve sorting.

As a starting reference, we can consider an initial situation with N spaces and kinds, and T tuples per kind chaotically inserted in the spaces ($H = 1$). The optimal algorithm would consider only tuples that are out of place, and accordingly move such tuples directly to the proper destination space: since each space holds T tuples and $T * (N - 1)/N$ of them are out of place, we would have a total of $T * (N - 1)$ tuples moved. However, this result would be accomplished only if mutation rate is zero and global information is available, but this is an ideal situation that does not fit for collective sort.

First of all, we cannot suppose that mutation rate is zero. In general, "reactiveness" also highlights the need of promptly reacting to an unpredictable, possibly significant change in tuple configuration, e.g. a user agent inserting a significant number of tuples into a single space in few time units. This means that the sorting service should devote some network resources to space observation (intercepting changing situations) and others to tuple transfers—as mentioned in the previous section. Different policies can be evaluated to balance observation and transfer rate, so as to either promoting convergence time in the static case or prompt reactiveness to changes in the dynamic case. Secondly, in our framework we do not have global information readily available. As already mentioned, this is because tuple spaces do not allow to count the number of tuples matching a given template. Accordingly, multiple probabilistic read operations are used to observe a space.

As a result, collective sort is to be run by continuously observing tuple spaces in order to be reactive to changes in tuple configuration, and by emergently selecting the tuple space that should gather a certain kind of tuple. Hence, this causes an unavoidable overhead with respect to the ideal case above. Moreover, it should be noted that, as in array sorting, the relative overhead obviously increases with the size of the problem, namely with the number of tuples and tuple spaces. As a general rule, the following result is sought for:

- Convergence cost. The average "cost" (network operations) for sorting starting from a chaotic configuration where $H = 1$ has to be at most one order of magnitude greater than the solution based on static, global information. Moreover, sorting cost should be expected to scale at most polynomially with the number of tuples and tuple spaces.

3.2.5 An example scenario

A key scenario of nowadays large-scale distributed systems is based on wireless sensor networks, namely, systems with a high number of small, wireless devices deployed in the physical world to monitor environmental properties [BDF+07]. A fundamental issue in this context is to devise effective strategies to gather the generated data [KEW02]. Recent works like [CMP07, YLB08] focus on building suitable algorithms for gathering such a data into a small set of *sink nodes*.

Based on this context, we show an application scenario for collective sort, as a way to clarify its main motivations and applicability. We suppose that sensors are spread on a wide area to monitor a set of environmental properties (temperature, pressure, humidity, light) and accordingly generate events reporting changes in such properties and/or faults. Such sensors form a highly dynamic set, as they may fail, move—e.g. since they could be deployed on board of a vehicle or a mobile device—or unpredictably hibernate to save energy. A proper infrastructure like envisioned in [CMP07] is exploited to gather all the generated events into a set of N sink nodes, reliably connected to a stable network—e.g.

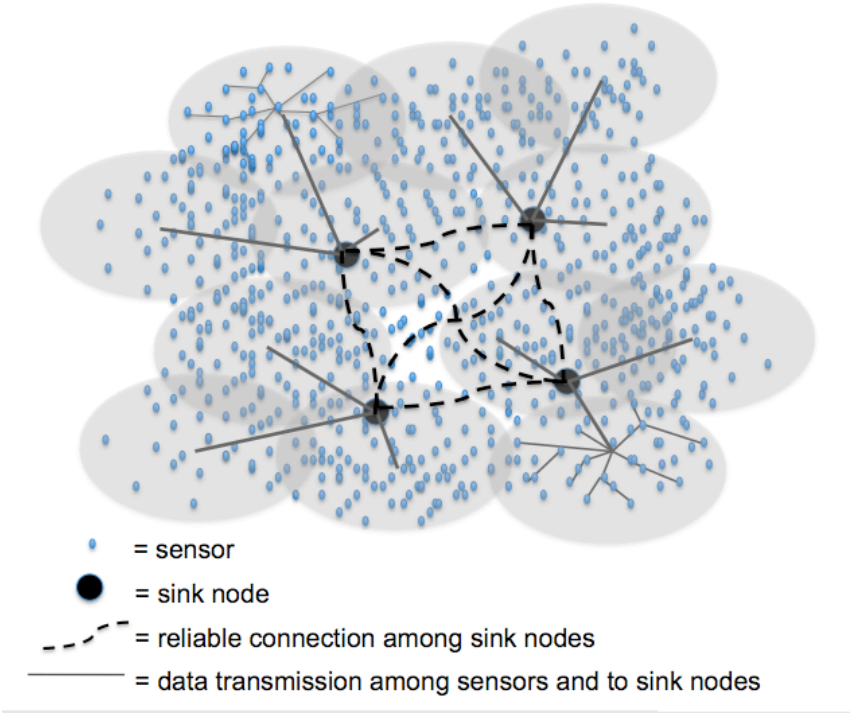


Figure 3.2: Applicative scenario with a wireless sensor network and multiple sink nodes.

they can be though of as environmental stations connected through a WAN. Each sink node is associated with a tuple space that unpredictably receives from sensors tuples of the kind:

```
event(sensor_id,sensor_position,time,property_name,property_value)
```

Hence, sensors will play the role of user agents. Additionally, other user agents – called manager agents – are connected to sink nodes, and are in charge of properly managing the generated data, executing the following tasks: generating a global map of a certain property, looking for peak values of a certain property, gathering general statistics of faults, erasing old events when updated data arrives, erasing duplicated events if the same one arrives to different sinks, and so on. See Figure 3.2 for a pictorial representation of this application scenario.

Whereas having multiple sinks enhances the efficiency of data gathering [CMP07, YLB08], this clearly introduces implementation issues since manager agents are forced to look at the tuples they are interested in throughout the set of spaces: the manager agent calculating the global temperature map looks for the template `event(?,?,?,temperature,?)`, the agent searching for faults would look for `event(?,?,fault,fault,fault)`, and so on. In order to avoid the overhead induced by the need of looking for tuples in all the N tuple spaces, it is helpful to set up a collective sort service that at runtime keeps the generated

data sorted in an emergent way. Note that the particular setting of this application is such that it may be hard to assign tuples to spaces statically, because the space where certain tuples will be deposited cannot be known at design time—the routing algorithm for sinks may be probabilistic, sensors can move, faults can occur in unpredictable places, certain properties may rapidly evolve only in certain places of the network, and so on.

Accordingly, after setting up a finite set of tuple templates of interest and deciding how they should be clustered into N kinds, N sorting agents can be deployed in the environmental stations, working at a certain sorting rate. As an example with $N = 4$, tuples modelling faults can go to space 1, tuples modelling peek values to space 2, tuples with updates on temperature and pressure to space 3, and finally tuples with updates on humidity and light to space 4. If during sorting, it become clear that one space is gathering much more tuples than others, kinds might be redesigned and sorting agents can be accordingly updated—here we treat this aspect as orthogonal to the sorting behaviour, though.

As the sorting service is powered on and reaches the desired level of sorting, each manager agent will shortly find the tuple space gathering the tuple it is interested in, thus improving the execution performance of its task.

3.3 A Solution to the Problem

The solution hereafter presented satisfies the quality attributes outlined in the previous section. In particular, the objective here is to show that this can be obtained without relying on agent intelligence – as many self-organisation approaches highlight – but by devising an agent behaviour based on very simple protocols composed of basic tuple operations.

For presentation purposes, and to make our design choices clearer, this section incrementally introduces the solution conceived for collective sort. An intermediate approach is discussed, along with some preliminary results that required a design improvement, ultimately leading to the solution fully evaluated in Section 3.4.

3.3.1 Basic Strategy

After depicting the general architecture for collective sort, a basic strategy for the solution inspired by ants' brood sorting is described.

Similarly to ants, each sorting agent performs local system observations, namely, observations on the local tuple space (where an item is possibly picked up) and observations on some remote and neighbouring tuple space randomly chosen (where the item may be dropped). According to such observations, if it can be inferred that some tuple should be sent to a remote tuple space s, then the agent locally removes the tuple and inserts it in s. This observation-action cycle is executed by each agent with a fixed rate r, so that the

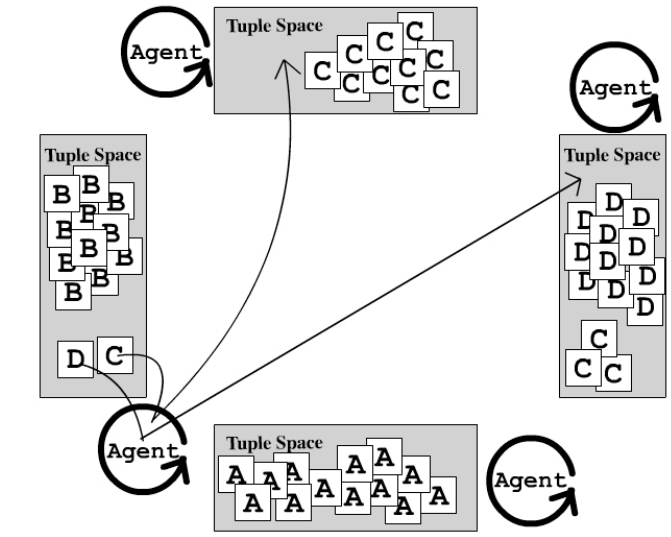


Figure 3.3: Architecture for collective sort—The agent on left-bottom should eventually relocate tuples `C` and `D` to different tuple spaces.

global sorting rate is $N * r$—which represents also the number of moving attempts per time unit. This scenario is depicted in Figure 3.3.

Therefore, each agent has the general goal of moving away tuples from its local tuple space whenever they are not forming a collection. In particular, the agent protocol we consider is as follows:

- `FIRE:` a remote tuple space R is drawn randomly;
- `LOCAL-OBS:` a uniform `read` operation is performed on local space L, yielding a tuple of kind K_L;
- `REMOTE-OBS:` a uniform `read` operation is performed on R, yielding a tuple of kind K_R;
- `MOVE:` if $K_L \neq K_R$ a tuple of kind K_R is moved from L (if any exists there) to R.

Uniform `read` operation, also called `urd`, is the operation exploited by sorting agents to read any tuple from the tuple space in a probabilistic way—remember that any tuple has the same probability of being retrieved. If `urd` operation on a tuple space yields a tuple of kind K, it means that (probabilistically) tuples of kind K are those mostly occurring so that K becomes the best candidate for finally aggregating on that space. Accordingly, once task `REMOTE-OBS` is executed, the agent knows that space L is likely to aggregate K_L while space R is likely to K_R. Accordingly, the rationale behind task `MOVE`, if K_R and

K_L are different, is to fruitfully send a tuple of kind K_R from L to R, so that both K_R in R and K_L in L will aggregate more.

The observation and the resulting decision made by the agent are however affected by probability, so that the correctness of this distributed algorithm needs to be checked by simulation, in order to test, first of all, whether complete sorting is reached starting from any initial situation, then evaluate the corresponding quality attributes.

3.3.2 Simulation Method

As far as collective sort evaluation is concerned, system evolution can be modelled as a Continuous-Time Markov Chain (CTMC), namely a stochastic transition system where transitions are labelled with rates, representing the average frequency at which the transition occurs [CGV07][2]. Let r be the rate of each sorting agent, the basic CTMC model of a collective sort run initially selects the next sorting agent that fires, through N transitions labelled with a rate r: this gives each sorting agent the same probability of being selected, while keeping the global sorting rate fixed to $N*r$. Then, the steps of the sorting agent's protocol are executed through transitions with very high rates (so as to make the corresponding transitions occur almost instantaneously), but taking into account probability when executing `urd` operations—each matching tuple has same probability of being returned based on the corresponding template concentration. This process is either executed t times, where t characterises the duration of the simulation, or until complete convergence is reached ($H = 0$). The state of system configuration is basically a matrix $(q)_{ik}$ of natural numbers, where q_{ik} is the amount of tuples of kind k into space i.

There are many simulation tools that could be used to experiment with collective sort, all providing different language expressiveness, but of course yielding the same simulation results—examples include SPIM [Phi06], SWARM [swa06] and REPAST [rep06]. Following the work in [CGV07] we adopted the simulation engine written in the MAUDE term-rewriting system presented in Chapter 2. The main reasons for this choice are that MAUDE *(i)* allows – thanks to term-rewriting paradigm – to flexibly structure system behaviour as a typed operational semantics in Plotkin's style [Plo91], *(ii)* executes transitions and computations with high performance thanks to advanced matching algorithms, *(iii)* is equipped with a full-fledged library for mathematical computations, and *(iv)* supports interaction with external tools—which could be built to control simulations and draw results. In fact, MAUDE allows to set up the syntax of system configuration in a flexible way by means of sorts and constructors (i.e., functions). For instance, the code:

```
sort Tuple TupleMSet Space DataSpace .
op _[_] : Qid Nat -> Tuple [ctor] .
subsort Tuple < TupleMSet .
op _|_ : TupleMSet TupleMSet -> TupleMSet [ctor assoc comm] .
```

[2] If modelling time passing is not of interest, but one only cares about counting events, Discrete-Time Markov Chains could be used instead.

```
op <_@_> : Nat TupleMSet -> Space [ctor] .
subsort Space < DataSpace .
op _|_ : DataSpace DataSpace -> DataSpace [ctor assoc comm]
```

defines the sort `Tuple` (`a[100]` represents 100 copies of a generic tuple kind `a`), `TupleMSet` (multisets of tuples separated by the associative and commutative composition operator "`|`"), `Space` (`<1@M>` represents space `1` with a multiset of tuples `M`), and `DataSpace` (a composition of spaces, again by operator "`|`"). As a result, an initial configuration where each tuple space has the same number of tuples per kind, for instance $T = 100$ and $N = 4$, is described as:

```
< T1 @ (K1[25])|(K2[25])|(K3[25])|(K4[25]) > |
< T2 @ (K1[25])|(K2[25])|(K3[25])|(K4[25]) > |
< T3 @ (K1[25])|(K2[25])|(K3[25])|(K4[25]) > |
< T4 @ (K1[25])|(K2[25])|(K3[25])|(K4[25]) >
```

Transition rules are written in our framework as a unary postfixed function `==>` associating system state with the set of all possible target states, each with its own rate. As an example, equation

```
eq (init | DS)==> =      (0.25->[[0]|DS]); (0.25->[[1]|DS]);
                         (0.25->[[2]|DS]); (0.25->[[3]|DS]).
```

associates an initial state including the `init` term with four possible states (denoting the choice of one sorting agent), where `init` is substituted with term `[i]` (`i` is the selected sorting agent), each labelled by rate 0.25 (global sorting rate is set to 1)[3]. Once the entire transition system is defined, a simulation run can be executed by a Maude command of the kind:

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

which produces on the standard output a trace of 5000 system states, starting from configuration `SS` and time `0.0`: such a trace is then used to draw a chart showing system evolution. The complete specification for the case $N = 4$ is reported in Appendix A.

An example of simulation trace starting from the above initial state is pictorially represented in Figure 3.4 (a), reporting the dynamics of the "winning" tuple kind in each tuple space—namely, the tuple kind that eventually aggregates there. Note that tuples reach their full aggregation level at different points in time, in an unpredictable way. The chart in Figure 3.4 (b) displays instead the evolution of tuple space `T1` taken as a reference: notice that only tuples of kind `K1` aggregate there despite the initial concentration of `K1` in `T1` was the same as other tuples. In particular, at some point around step 1000, there is a bifurcation which promotes aggregation of `K1` tuples instead of `K2`.

[3]The above code works only with $N = 4$ just to simplify the description, but our specification deals with the general case.

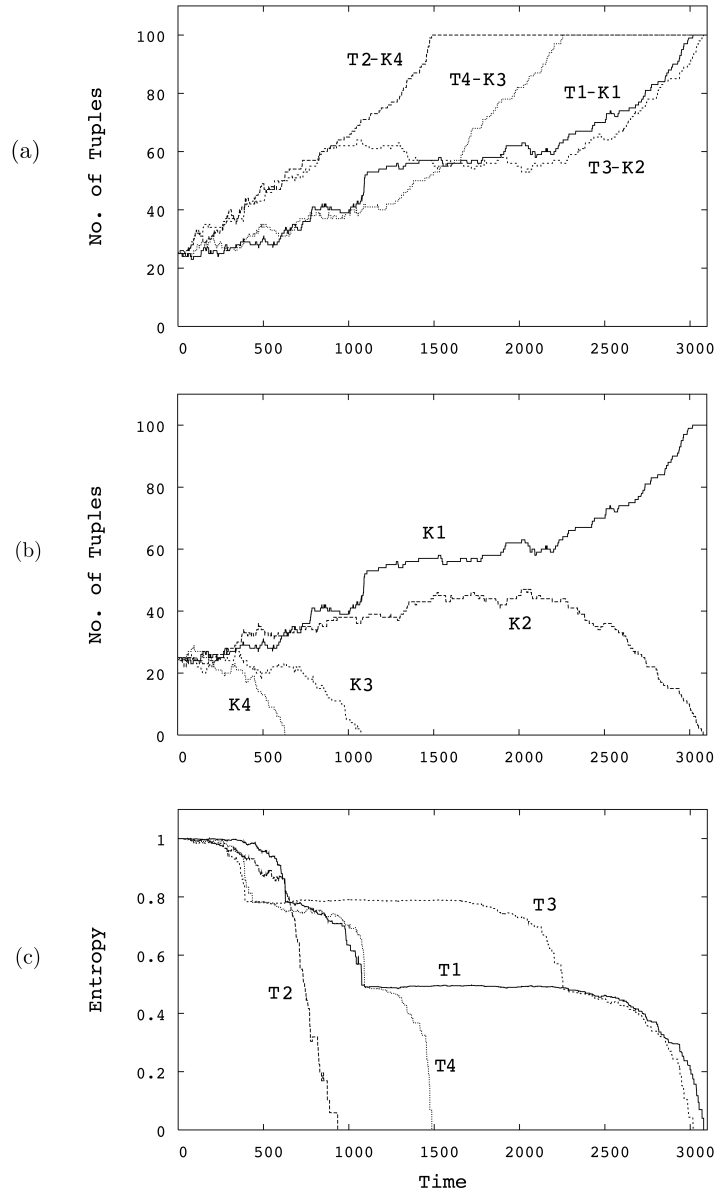


Figure 3.4: Charts of a simulation trace: (a) Winning tuple kind; (b) Tuple space T1; (c) Entropy in each tuple space (normalised).

It is interesting to analyse also 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 simulated strategy aims at increasing the inner order of the system, entropy is expected to decrease to zero, as actually shown in Figure 3.4 (c). Each chart reports the number of protocol instances (move attempts) executed by agents: the chart shows that in this simulation

full sorting is reached after around 3000 time units—i.e. 3000 executions of the agent protocol.

3.3.3 On Convergence

At a first glance, the solution developed so far appears to converge to complete sorting from any initial configuration of tuples. However, it is easily detectable that there are some stable states attracting the system trajectory and having positive entropy, that is, characterised by an incomplete degree of sorting. A state of this kind is called *local minimum* (from the standpoint of entropy). An example of such a minimum is the following state, obtained by the traces shown in Figure 3.5:

```
< T1 @ (K1[100])|(K2[0]) |(K3[0])  |(K4[0]) > |
< T2 @ (K1[0])  |(K2[69])|(K3[0])  |(K4[0]) > |
< T3 @ (K1[0])  |(K2[31])|(K3[0])  |(K4[0]) > |
< T4 @ (K1[0])  |(K2[0]) |(K3[100])|(K4[100]) >
```

Tuple kind `K2` is the only aggregating in both `T2` and `T3`, and at the same time, both kinds `K3` and `K4` aggregate in space `T4`. It is easy to recognise that once this state is reached, no agent will ever move a tuple, since in no space a tuple is found that aggregates less than elsewhere. Our simulations show that: *(i)* about 5% of runs from the initial chaotic configuration with $N = 4$ ends in a local minimum; *(ii)* this probability increases with N, e.g. it is above 15% when $N = 7$, since many more local minima exist; *(iii)* simulations from states that are sufficiently near to a local minimum always end up in it—local minima are attractors. This makes the approach discussed so far inadequate with respect to the quality attributes previously defined, so that a suitable solution is required before proceeding with any further evaluation. The attempt of solving this problem led to the solution actually proposed in this chapter, as discussed in the following.

The main reason that the local minimum analysed above cannot be escaped lies in the fact that the strategy we developed does not explicitly avoid the case where the same tuple aggregates in two different tuple spaces. In fact, due to task `MOVE` in the sorting agent protocol, nothing is done when $K_L = K_R$! As a consequence, it may happen that the same tuple kind fully aggregates on two different tuple spaces, and dually, two remaining tuple kinds aggregate in the same space as shown in the local minimum above.

These two issues can actually be coped with by a unique solution coming from a more careful analysis of brood sorting in social insects. There, an ant picks an item up and releases it where a new place is found with greater concentration, expressed as quantity of brood over a unit of space. That is, an ant is implicitly able to compare the amount of brood with a standard quantity, which in that specific case is represented by the amount of free space. If a similar notion were defined in collective sort, that could in principle allow to solve the two issues above. On the one hand, if space `T3` could be recognised as having "less" tuples `K2` than space `T2`, then movements from space `T3` to `T2` could be

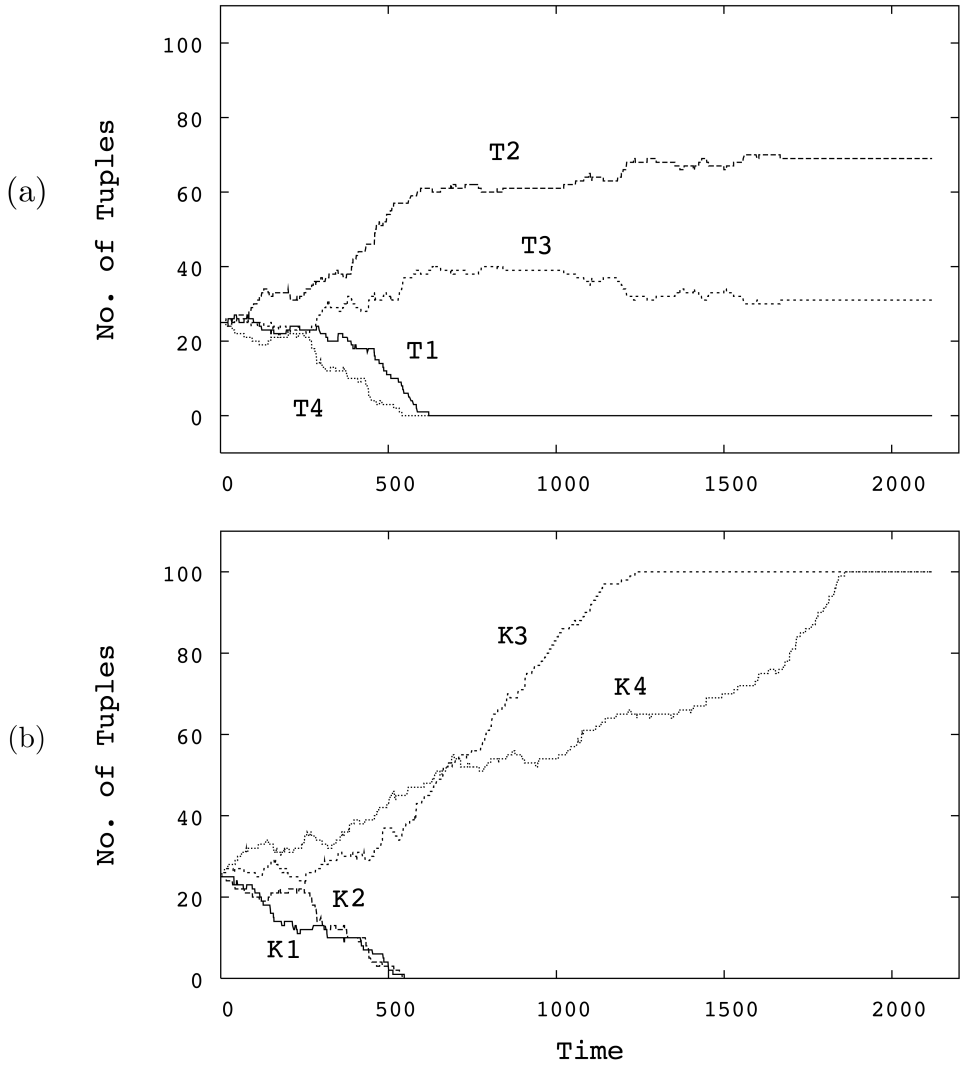


Figure 3.5: Charts of a simulation trace leading to a local minimum: (a) Tuple kind K2 aggregating in spaces T2 and T3; (b) Both kinds K3 and K4 aggregating in space T4.

promoted. Hence, as one of the two spaces stops aggregating tuples, some tuples `K3` or `K4` could be moved there from space `T4`.

3.3.4 System annealing by noise tuples

To implement a mechanism supporting this idea, another kind of tuple called `noise` is introduced, which is initially supposed to have a constant concentration (i.e. amount) in all spaces throughout the entire sorting process. Such tuples are not inserted and retrieved by user agents, and consequently are not subject to sorting, but managed (in-

serted/retrieved) by sorting agents. Now, when a sorting agent performs a uniform read to randomly select a tuple, such an observation gets "perturbed" since there is a non-zero probability that the result is a `noise` tuple. Simply, if the tuple space holds, say, 95 regular tuples (those to be sorted) and 5 noise tuples, there is a 5% of probability of retrieving `noise`. Following the previous version of the algorithm, the new interaction protocol is such that a tuple is moved from the local space to a remote space if and only if the two observations involve different tuple kinds—but now one of them could be `noise`. As an example, if the remote observation is `noise` and the local one provides kind `k`, then the locally observed tuple is moved anyway, even though this does not necessarily decrease entropy. That is, the role of `noise` tuples is to alter probabilistically the correctness of agent actions, which now may temporarily increase disorder when observations are perturbed by some noise. As a result, this mechanism causes tuples to be moved even though the system is in a local minimum, hopefully making system trajectory escape from it. This technique actually resembles the principles underlying *simulated annealing* used in optimisation algorithms [KGV83]. There, a perturbation is added in order to avoid the risk of finding non-optimal solutions: such a perturbation is initially high and fades continuously as the system searches solutions, until completely disappearing.

In our case, the occurrence of `noise` tuples models such a perturbation: what should be the dynamics of noise through time, then? A feasible approach would be to set an initial amount of noise equal in all tuple spaces, and either leave it unaltered during system life cycle or decrease it at a fixed rate. However, this choice would require to set noise amount at design time, but then optimality of this amount would depend on the average occupation of tuple spaces during system execution [VCG07]: this is not appealing since the searched approach needs to work independently of the number of tuples in the system. What it is actually needed is a fully adaptive noise mechanism, where an initially very low noise concentration increases as the system approaches a local minimum, and decreases when the minimum is escaped. In this way, we could expect the system performance to be only slightly affected whenever the system stays sufficiently far from local minima; on the other hand, noise production may become significant only in unfortunate cases where local minima are approached.

To achieve this result, noise is managed as follows:

- As already described, noise alters observations performed by uniform read operations, and hence the pertinence of tuple movements, since a tuple is moved if the local and remote observations are different—though one of them could be noise.
- Initially only one noise tuple occurs in each tuple space, hence, perturbation is very low; as a result, sorting performance is not significantly affected.
- Each time two tuple spaces seem to aggregate the same tuple kind—two equivalent non-noise observations are likely to be performed so that noise is increased; in fact, since that would mean we are likely approaching a local minimum, system annealing needs to be increased.

- When some tuple is transferred due to pertinent observations—two different non-noise observations are made—noise is decreased; as this would mean we are likely escaping a local minimum, system annealing has to be increased.

Accordingly, the agent protocol is changed as follows:

- `FIRE:` a remote tuple space R is drawn randomly;
- `LOCAL-OBS:` a uniform `read` operation is performed on local space L, yielding a tuple of kind K_L;
- `REMOTE-OBS:` a uniform `read` operation is performed on R, yielding a tuple of kind K_R;
- `MOVE:` if $K_L \neq K_R$ a tuple is moved from L to R, i.e.:
  - if $K_R =$ `noise`, such a tuple has to be of kind K_L;
  - otherwise, the tuple has to be of kind K_R (if existing in L);
- `NOISE:` if $K_L \neq$ `noise` and $K_R \neq$ `noise` local noise amount is changed, i.e.:
  - if $K_R = K_L$ then noise is increased by one in L;
  - if $K_R \neq K_L$ then noise is decreased by one in L.

Now both K_L and K_R could be noise. Task `MOVE` states that differences in observations made in L and R should always cause transfer: if K_R is not noise, a K_R tuple is moved to R, otherwise, a K_L tuple is moved to R. Task `NOISE` increases noise when L and R are aggregating the same (non-noise) tuple $K_R = K_L$, and decreases noise whenever a non-perturbed transfer is actually executed.

Consider now the worst case of a symmetric local minimum:

```
< T1 @ (K1[100])|(K2[100])|(K3[0]) |(K4[0]) > |
< T2 @ (K1[0])  |(K2[0])  |(K3[50])|(K4[0]) > |
< T3 @ (K1[0])  |(K2[01]) |(K3[50])|(K4[0]) > |
< T4 @ (K1[0])  |(K2[0])  |(K3[0]) |(K4[100]) >
```

Noise is initially expected to increase in both tuple spaces `T2` and `T3` (`NOISE`). At some point, movement of tuples `K3` is going to occur between `T2` and `T3` since some noise is observed (`MOVE`). Due to a bifurcation effect, if either space `T2` or `T3` have a greater concentration of tuples `K3` with respect to noise, this will cause more tuples to be transferred there, so that space `T2` or `T3` will eventually fully aggregate tuples `K3`. Accordingly, the other tuple space will be emptied, lose noise tuples, and finally become target of tuples of kind `K1` and/or `K2`. This is actually what can be observed from the traces in Figure 3.6 (a) and (b), showing how the local minimum is escaped in spaces `T2` and `T1`: in both

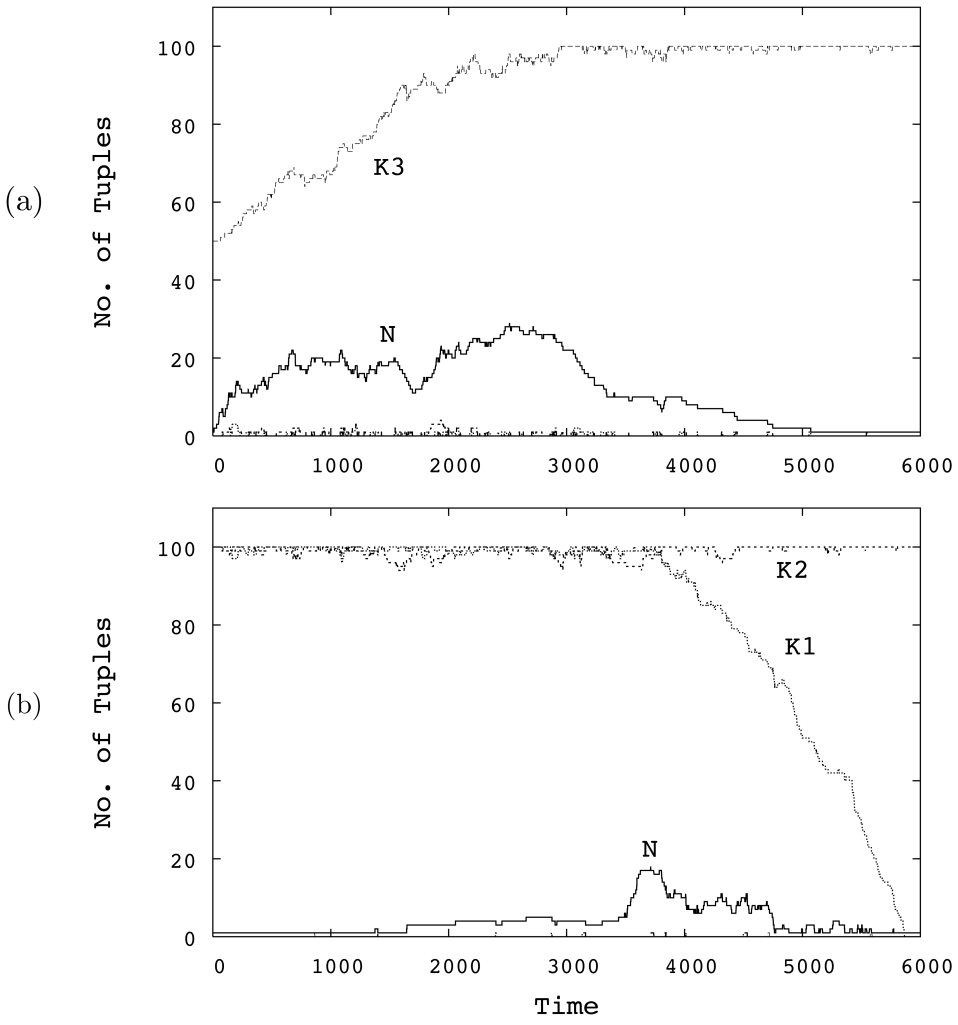


Figure 3.6: Charts of a simulation trace escaping from a local minimum: (a) Situation in space T2: winning tuple K3 and noise in evidence; (b) Situation in space T1: kind K1 leaves the space.

cases it is possible to recognise that as noise tuples increase, the system escapes the local minimum configuration leading to fading of noise tuples.

More simulations performed to evaluate this solution actually show that: *(i)* using noise slightly affects performance, for typically systems stay away from local minima and generate little noise; *(ii)* starting from a local minimum, the system is always able to escape it; and *(iii)* full sorting is always eventually reached. This solution hence appears to be a promising one. The next section provides a comprehensive evaluation of the proposed solution.

3.4 Evaluation

According to the quality attributes described in Section 3.2.4, concerning convergence, scalability, and reactiveness, the proposed approach is here evaluated. To this end, standard analysis techniques for distributed systems usually include automatic machine checking or hand-written proofs.

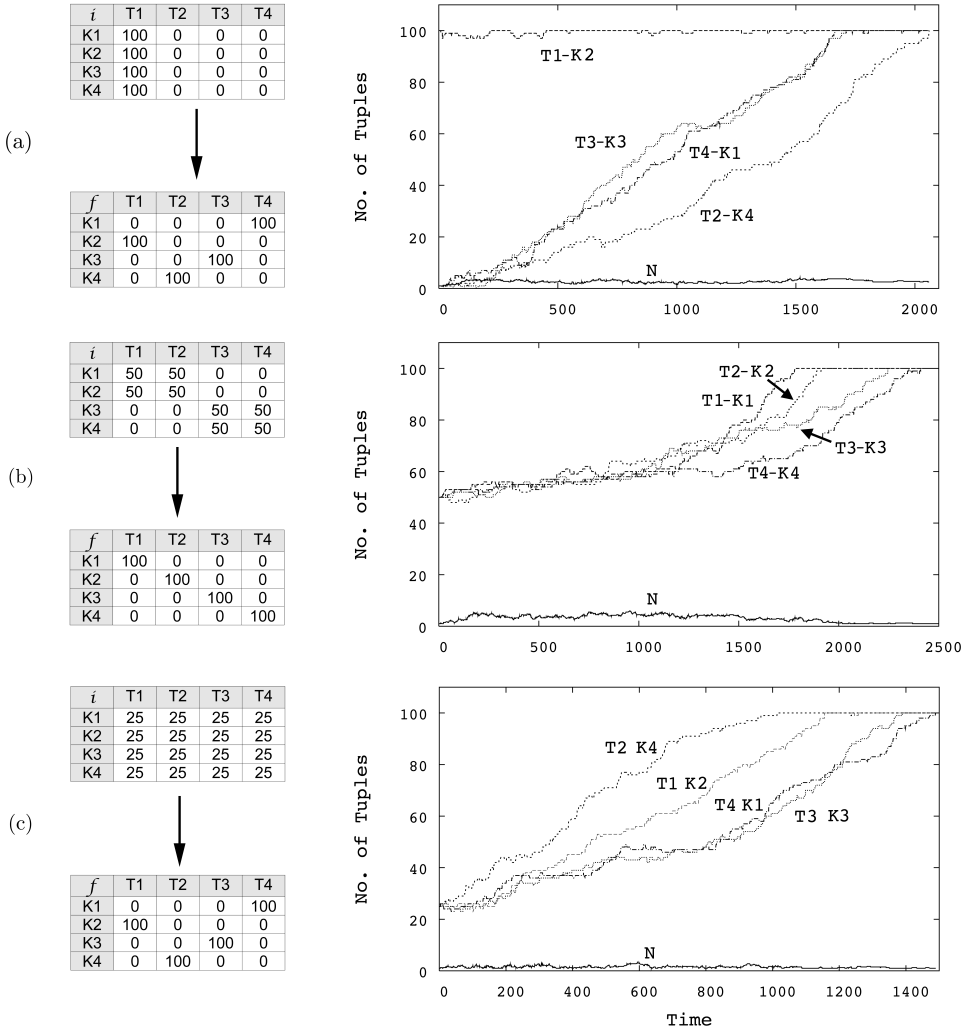
In the former case, analysis would be obtained e.g. by a probabilistic model checking technique [KNP07, RKNP04]. There, the graph of all possible states and transitions is constructed and annotated with probabilities and rates, so that queries in a probabilistic logic like PCTL [HJ94] can be checked by navigating the entire graph: e.g., for collective sort we could ask what the probability is that from an initial state with $N = 4, T = 100, H = 1$ an ordered state ($H = 0$) is reached within 3000 time units. However, this technique suffers from state-space explosion problem, so that even the construction of the entire graph is a very expensive operation so that proper optimisation techniques are required for model construction. A more detailed discussion on formal verification techniques for self-organising systems is reported in Chapter 7 where, as a reference example, some emergent properties expressed in PCTL logics are verified on different collective sort instances modelling the first solution described in Section 3.3.

In the latter case, we should try to find proofs that the proposed algorithm converges, and in general, that the quality requirements are met. However, this is particularly complex, and very few examples exist in literature that apply this approach to self-organising systems. A recent exception is the work in [YN08], which focusses on a very straightforward self-organisation scenario, in which a chain of cellular-like components is subject to a repeated applications of local transformation rules. Necessary and sufficient conditions are provided for the emergence of a certain global pattern of cellular states—similarly to what happens in certain stages of embryogenesis. This research direction, although very interesting, is still in its infancy and cannot be applied to generally analyse systems like collective sort.

On the other hand, computer-based simulation techniques have gained a growing attention over the past years as an important tool for studying complex systems [Yan97, Bal00], especially when previous analytic results are not available [Yan97]—e.g. due to analytical intractability of the target system. Accordingly, simulation is the main approach adopted for evaluating collective sort.

3.4.1 Full Convergence

In our solution, on the one hand, non-perturbed tuple movements cause entropy to decrease; on the other hand, the noise mechanism successfully perturbs those configurations that approach local minima. Figure 3.7 shows examples of how the system can come to full sorting starting from different initial configurations—these are taken as samples of several simulations we ran, all of which reached full sorting. Interestingly, we can observe



i	T1	T2	T3	T4
K1	100	0	0	0
K2	100	0	0	0
K3	100	0	0	0
K4	100	0	0	0

f	T1	T2	T3	T4
K1	0	0	0	100
K2	100	0	0	0
K3	0	0	100	0
K4	0	100	0	0

i	T1	T2	T3	T4
K1	50	50	0	0
K2	50	50	0	0
K3	0	0	50	50
K4	0	0	50	50

f	T1	T2	T3	T4
K1	100	0	0	0
K2	0	100	0	0
K3	0	0	100	0
K4	0	0	0	100

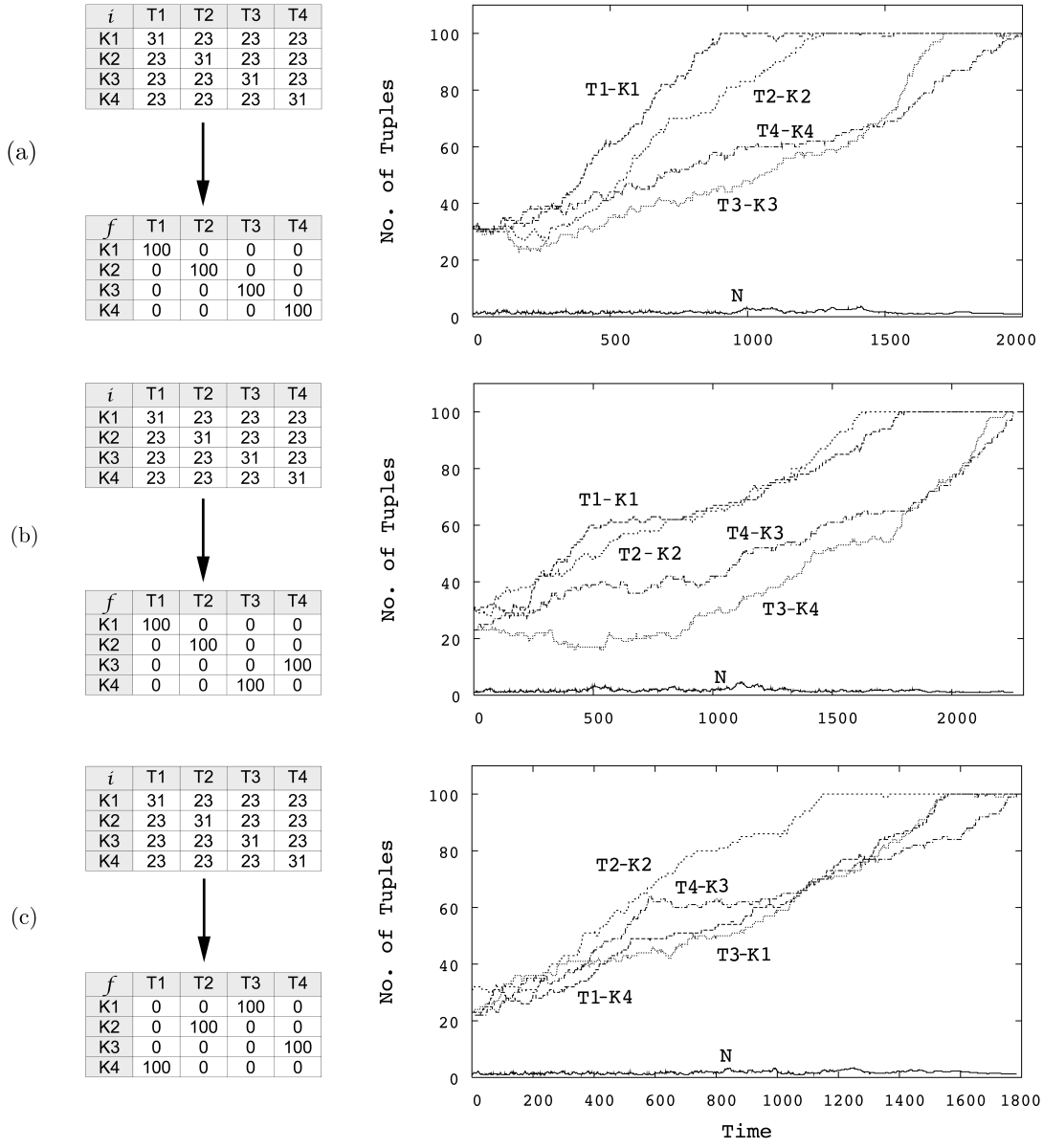
i	T1	T2	T3	T4
K1	25	25	25	25
K2	25	25	25	25
K3	25	25	25	25
K4	25	25	25	25

f	T1	T2	T3	T4
K1	0	0	0	100
K2	100	0	0	0
K3	0	0	100	0
K4	0	100	0	0

Figure 3.7: Sorting from different configurations: winning tuples and noise.

that in these cases – differently from the one shown in Figure 3.6 – total noise is typically very low: this is mainly because here the system stays sufficiently far from local minima. In all the three cases, the system dynamics is highly probabilistic and hardly predictable.

Moreover, it is typically the case that the same system shows different dynamics in different runs. As another illustrative example, consider the starting configuration shown in Figure 3.8, which apparently seems to lead to the full sorted configuration in chart (a) ever. Many cases – about 20% of times – show that the system actually converges to a different final state, where one or more kinds actually aggregate in spaces with an initial smaller concentration than other kinds. This, again, highlights the true unpredictable character of any self-organising solution like the one provided here for collective sort.



i	T1	T2	T3	T4
K1	31	23	23	23
K2	23	31	23	23
K3	23	23	31	23
K4	23	23	23	31

f	T1	T2	T3	T4
K1	100	0	0	0
K2	0	100	0	0
K3	0	0	100	0
K4	0	0	0	100

i	T1	T2	T3	T4
K1	31	23	23	23
K2	23	31	23	23
K3	23	23	31	23
K4	23	23	23	31

f	T1	T2	T3	T4
K1	100	0	0	0
K2	0	100	0	0
K3	0	0	0	100
K4	0	0	100	0

i	T1	T2	T3	T4
K1	31	23	23	23
K2	23	31	23	23
K3	23	23	31	23
K4	23	23	23	31

f	T1	T2	T3	T4
K1	0	0	100	0
K2	0	100	0	0
K3	0	0	0	100
K4	100	0	0	0

Figure 3.8: Different sorting result from the same configuration.

As a further measure for evaluating collective sort convergence, we considered the variability of sorting time throughout a large series of 1000 simulations with $N = 4$ and $T = 400$ tuples per kind. Figure 3.9 shows how sorting time distributes over the executed simulations. It is easy to recognise a peak centered around 2500 time units, meaning that most of the simulation runs (~ 220) led to a sorting time value close to 2500 time units, while 92.2% of the simulation runs led to a sorting time ≤ 5000 time units. In addition,

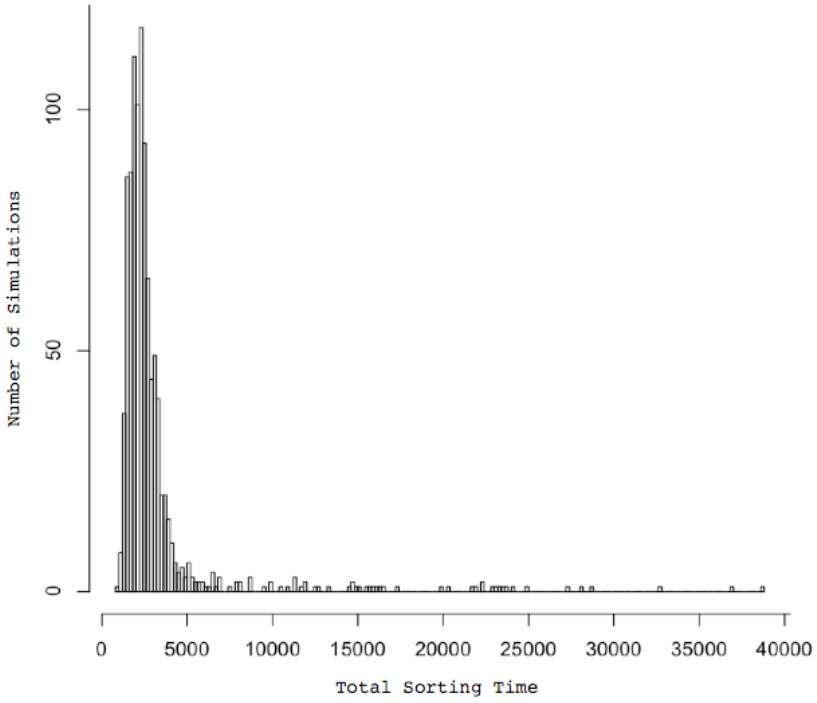


Figure 3.9: Total-sorting-time distribution over 1000 simulation runs executed with $N = 4$ and 400 tuple per kind.

a few percentage of simulations led to a high value of convergence time (up to 40000 time units): this is due to evolutions of collective sort that approached one or iteratively more states of local minimum, which usually require a much higher amount of time to achieve full sorting due to the perturbation effect generated by noise tuples. This chart then allows to show the influence of noise on convergence: most of the times it does not affect convergence time, while in other cases where sorting approaches a local minimum, noise starts working ultimately leading to convergence but requiring a higher convergence time.

3.4.2 Sorting Cost and Scalability

Concerning performance, it should be noted from previous charts that the average time for reaching full sorting is around 2500 time units, considering the basic case with 4 tuple spaces, 4 tuple kinds, and an initial set of 400 tuples. The global sorting rate considered is 1.0, that is, there is an average of one transfer attempt per time unit, and accordingly a sorting agent rate of 0.25 transfer attempts per time unit.

The optimal solution to the problem – in which a snapshot of the system is taken and agents are accordingly pre-programmed by an explicit list of tuple movements to be performed – would require instead around 300 time units (see Section 3.2.4), which is the time necessary to move the tuples that are not already in the right space. Namely, in the

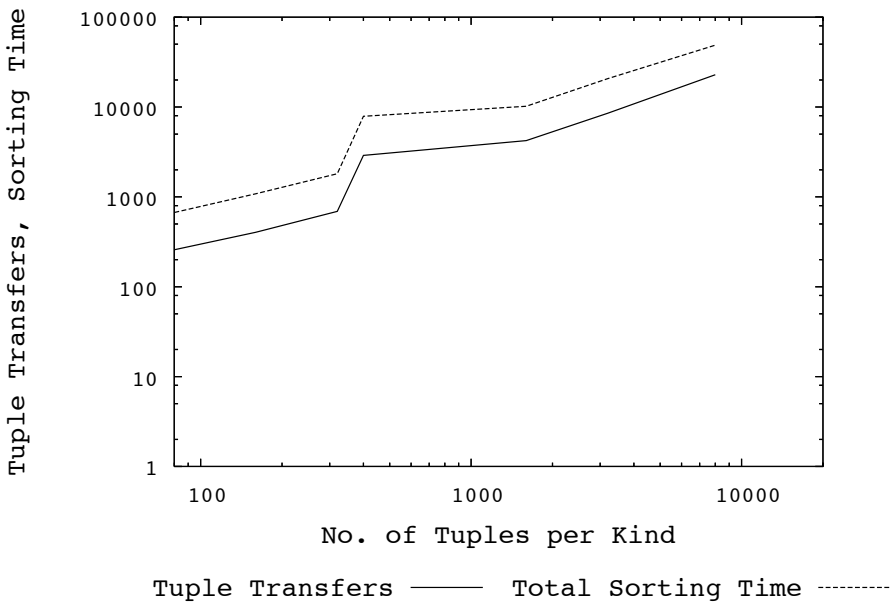


Figure 3.10: Scalability in the number of tuples ($N = 4$): transfers and elapsed time units.

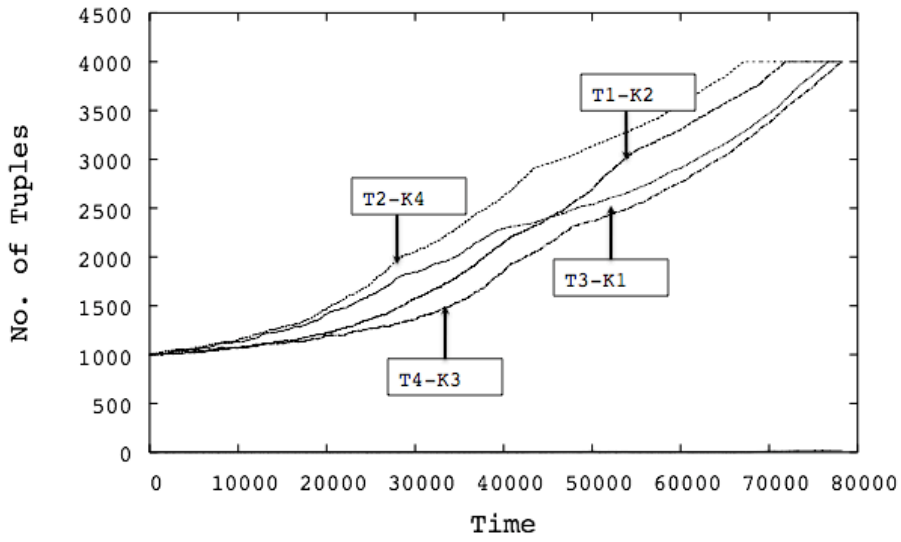


Figure 3.11: Winning tuple (a) and noise (b) evolution for a simulation with $N = 4$ and 4000 tuples per kind initially distributed in a uniform way.

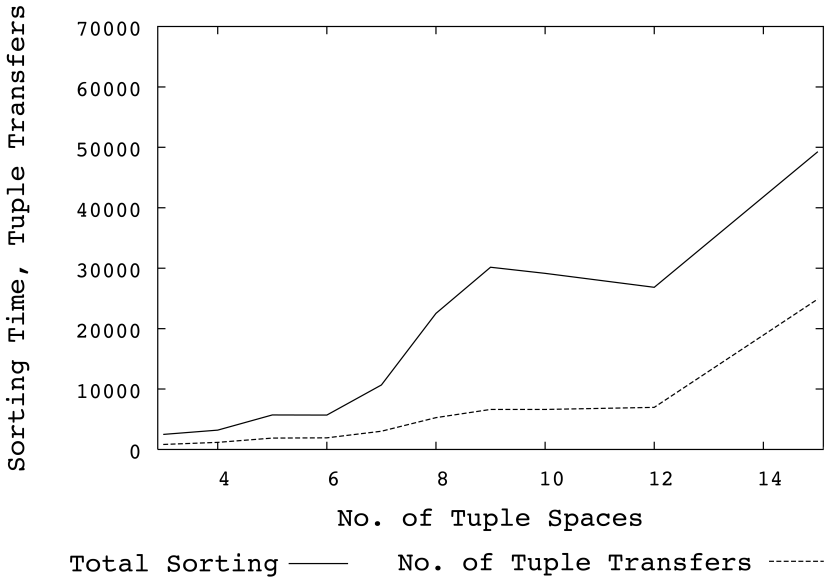


Figure 3.12: Scalability in N (n. of tuple spaces/kinds): transfers and elapsed time units.

basic case of the problem, system performance is around one order of magnitude worse than optimal solution: this is still within the expected range of the quality requirements sought for.

It is now interesting to see how much our solution scales with the dimension of the problem. First of all, we consider scalability in the number of tuples, that is, how the size of the set of tuples affects the system behaviour. Figure 3.10 shows average performance values (taken from a set of 20 runs), on a system with $N = 4$, an initially chaotic configuration, and an increasing number of tuples from 80 to 8000. Two values are depicted: tuple transfers and sorting time. The first one is simply increased each time a tuple is moved, the second one is instead the elapsed time units—or moving attempts. Another interesting value is network usage (or number of remote interactions), which can be directly computed as the sum of tuple transfers and elapsed time—since other than tuple transfers, we have a remote observation per time unit. As a main result of these charts *(i)* the proposed approach scales linearly with the number of tuples, and *(ii)* the number of transfers is around 1/3 of the number of moving attempts—this parameter somehow representing the pertinence of observations.

A simulation trace showing the evolution of collective sort with $N = 4$ and 4000 tuples per kind is reported in Figure 3.11. It is easy to see that the evolution of the simulation is

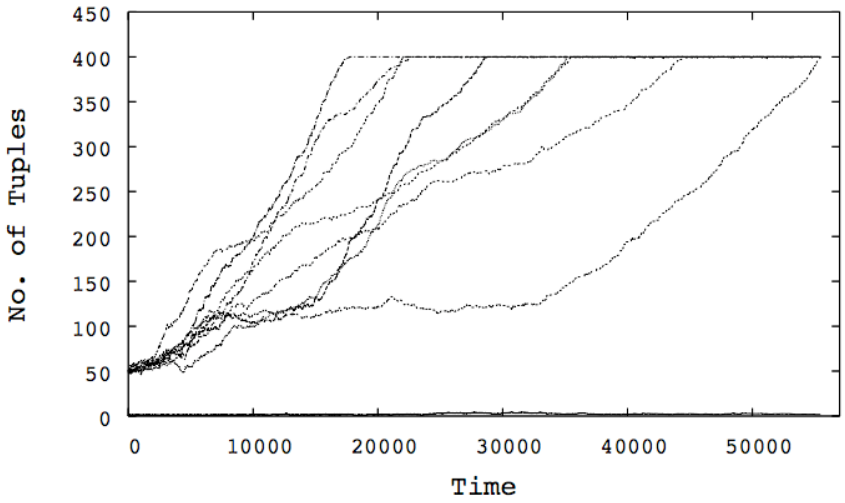


Figure 3.13: Winning tuple (a) and noise (b) evolution for a simulation with $N = 8$ and 400 tuples per kind initially distributed in a uniform way.

not greatly affected by the higher tuple concentration, the only difference is that system evolution appears much more deterministic than e.g. in Figure 3.7 (c). In addition, total noise concentration keeps on values which are 2 orders of magnitude lower than total tuple concentration as in experiments with lower tuple concentrations. This simulation shows that non-determinism, as well as the need of relying on noise, actually decreases as the system size becomes larger and larger: hence, unexpected situations are more likely to occur on smaller-scale systems.

Scalability in N, the number of tuple spaces (and kinds), is a more critical issue. Figure 3.12 shows average performance values (taken from a set of 20 runs) on a system with a fixed number of tuples equals to 400, an initially chaotic configuration, a fixed sorting agent rate equal to 0.25, and an increasing number of tuple spaces, from 3 to 15—the global sorting rate is $0.25 * N$. As in the previous case, we measured the number of tuple transfers and elapsed time units. Here we notice that the difficulty for the proposed solution to scale is higher, though the result appears reasonable anyway: while average time for full sorting is 2500 with $N = 4$, it is about 30000 with $N = 10$—the latter is indeed a more complex problem.

The sorting cost actually appears to be quadratic in the number of tuple spaces. This result seems however a key characteristic of collective sort, rather than depending on the proposed solution—this is a problem even in array sorting, which does not scale linearly. In general, if some tuple remains far from where its kind is aggregating, the time (and network operations) needed by its sorting agent to find the proper remote

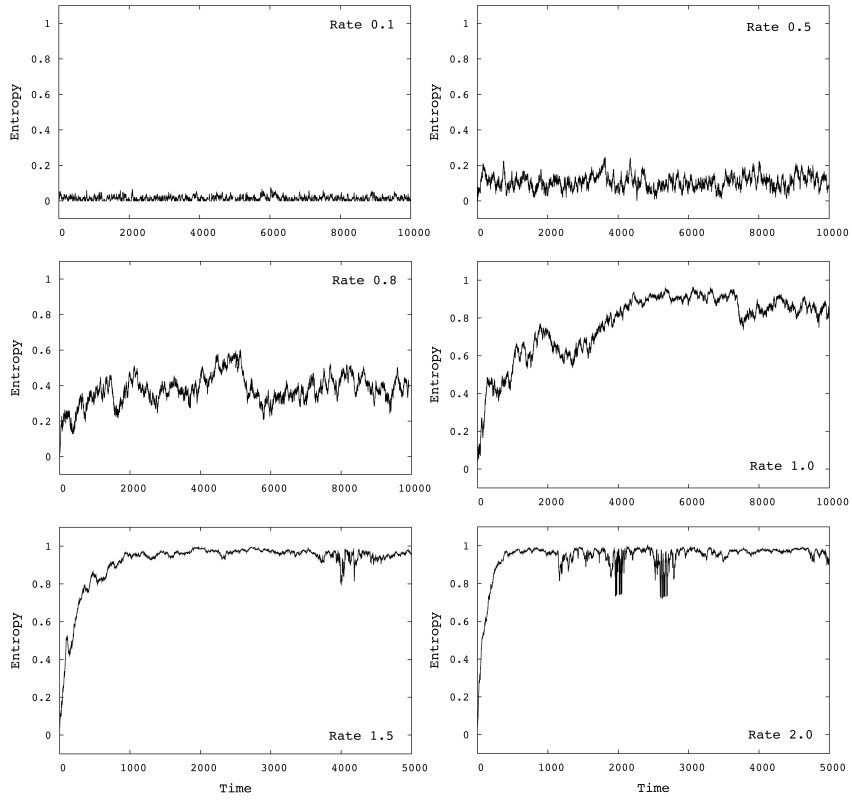


Figure 3.14: Evolution of entropy with different perturbation/sorting ratio.

tuple space is linear in N, and this applies to all sorting agents/kinds. This appears to be an intrinsic consequence of the fact that a global status is not available to sorting agents, and observations are necessarily pointwise. In addition to the results shown in Figure 3.12, Figure 3.13 reports a simulation trace executed with $N = 8$ and 400 tuples uniformly distributed among the N available kinds. Again, noise keeps on values which are 2 orders of magnitude lower than tuple concentration even though collective sort operates on a larger set of tuple spaces. To summarise, even considering scalability, the proposed approach appears to successfully meet the desired requirements.

3.4.3 Reactiveness

The main reason that collective sort has been solved by using a self-organising approach is to tackle unpredictable interactions with the environment. A typical usage scenario includes user agents that exploit the coordination service provided by the tuple spaces, that is, they keep inserting and removing tuples. The details of this behaviour cannot be fully known a priori, hence, sorting should be able to react to changes of the surrounding conditions in an adaptive way. Providing a comprehensive and general set of simulations

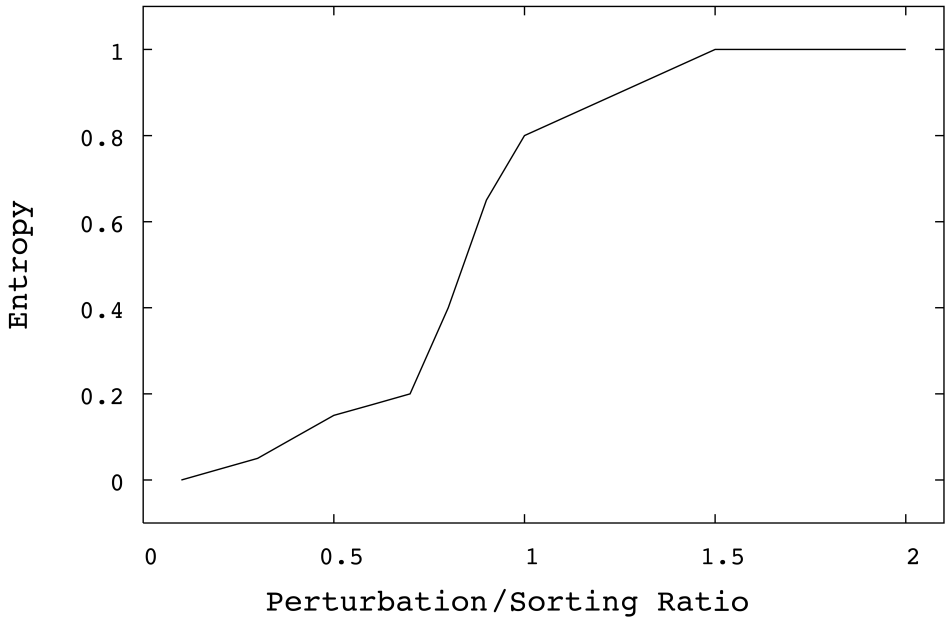


Figure 3.15: Maximum entropy depending on perturbation/sorting ratio.

in this context is not easy, since user agents can manifest an extremely wide range of different interactive behaviours. Here we briefly show how the ratio between user agent rate and sorting agent rate, called *mutation/sorting ratio*, influences the result of sorting. To this end, we keep the global sorting rate fixed to 1.0 and include in the simulation a *mutation rate* for user agents, that is, the rate at which each user agent randomly moves a tuple from one space to another. Starting from an initially sorted configuration of tuples (400 tuples and $N = 4$, with $H = 0$), depending on mutation rate, we easily expect that *(i)* full sorting is almost always maintained, *(ii)* a certain level of (partial) sorting can be maintained, or *(iii)* the system becomes more and more unsorted as time passes. Evolution through such situations is reported in Figure 3.14, where each chart provides the evolution of entropy over time for different mutation rates.

As shown in the summary Figure 3.15, the key factor is the mutation/sorting ratio, which gives a clear indication of the adequacy of sorting resources, in terms of the maximum level of entropy they can guarantee—a mutation/sorting ratio smaller than 0.5 seems to be a reasonable trade-off between the required sorting resources and the corresponding achieved sorting level. It is clear that a form of load-balancing is required to be sure that sorting resources are adequate with respect to the current degree of disorder, and can self-adapt to it—increased on a by-need basis and then decreased. To this end, techniques related to the prey-predator approach as studied e.g. in [GVO06, GVC06a] could be evaluated in future research.

4 SwarmLinda

Coordination systems have been used in a variety of different applications but have never performed well in large scale, faulty settings. The sheer scale and level of complexity of today's applications is enough to make the current ways of thinking about distributed systems (e.g. deterministic decisions about data organisation) obsolete. All the same, computer scientists are searching for new approaches and are paying more attention to stochastic approaches that provide good solutions "most of the time". The trade-off here is that by loosening certain requirements the system ends up performing better in other fronts such as adaptiveness to failures. Adaptation is a key component to fault-tolerance and tuple distribution is the center of the fault-tolerance problem in tuple-space systems. Focussing on the aforementioned context, this chapter presents an alternative approach to collective sort, showing how tuple distribution in LINDA-like systems can be solved by using an adaptive self-organised approach *à la* swarm intelligence. In particular, the main goal of SwarmLinda is to extend the traditional LINDA coordination model by providing the `out` primitive with a semantics inspired by swarm intelligence, resulting in a novel approach to tuple distribution. The main differences from collective sort lie in the fact that now there is no network topology and number of tuple spaces/tuple kinds fixed in advance. In addition, the nature of SwarmLinda is *online*: in other words, tuples are supposed to get stored in the "right" area of the network as soon as the corresponding insertion phase occurs by `out` operations. On the contrary, collective sort does not rely on any specific semantics of the `out` primitive. In fact, tuples already stored in the network are simply moved among tuple spaces as a result of the strategy followed by so-called sorting agents having the goal of locally ordering the information in the network. The results discussed in this chapter demonstrate that the approach can efficiently and adaptively address the problem of tuple distribution. The work presented here is in great part inspired by the research on SwarmLinda described in [CMVT07a, CMVT07b, CMVT07c, CMTV07].

4.1 Introduction

Most computer science researchers have realised that the sheer scale of current systems and applications is "forcing" them to look away from standard approaches to deal with these

systems and concentrate their efforts in the search for solutions inspired from other areas such as biology and chemistry. Why have not they done this before? It is quite simple: *(i)* computer scientists are used to having total control over their systems' workings, and *(ii)* the scale of the problems they face does not require unconventional solutions. The consequence is that antiquate ways of thinking flood today's applications leading to solutions that are complex and brittle. It is outside the scope to discuss who is to blame for this scenario. In fact, no one may be to blame since it would have been hard to foresee such an increase in computer usage let alone people's dependency on computer systems.

The aforementioned issues could be improved if we better understand that uncertainty is not a synonym for "incorrectness". We all live in a world where the causality of events makes non-determinism/uncertainty a norm. Although far from the level of complexity of the real world, computer applications exhibit complexity that is hard to deal with even by today's most powerful computers. For a sample of challenging problems we are facing, see the grand challenges listed by the Computing Research Association (CRA) [SD03] and by the UK Computing Research Committee (UKCRC) [UK 04].

Coordination systems are constantly being pointed as a mechanism to deal with some of the complex issues of large-scale systems. The so-called separation between computation and coordination [GC92] enables a better understanding of the complexity of distributed applications. Yet, even this abstraction has had difficulties in overcoming hurdles such as fault-tolerance present in large-scale applications. A new path in dealing with complexity in coordination systems has been labelled *emergent coordination* [OM06]. Examples of this approach include mechanisms proposed in models such as the first SwarmLinda proposal [MT03] and TOTA [MZL03]. This chapter explores one mechanism proposed in SwarmLinda that refers to the organisation of data (tuples) in distributed environments using solutions borrowed from natural forming multi-agent swarms, more specifically based on ant's brood sorting behaviour [DGF+91].

In tuple-space coordination systems, the coordination itself takes place via generative communication using tuples. Tuples are stored and retrieved from distributed tuple spaces. As such, the location of these tuples is very important to performance—if tuples are maintained near the processes requiring them, the coordination can take place more efficiently. Standard systems base this organisation on hash functions that are quite efficient but inappropriate to large distributed applications because they are rather difficult to adapt to dynamic scenarios; particularly to scenarios where distributed nodes/locations can fail. Here, we propose a mechanism to implement the algorithm suggested in the original SwarmLinda. The performance of the resulting approach is tested by using a metric based on the spatial entropy of the system as for collective sort. We argued that this approach is more appropriate to faulty systems because it does not statically depend on the existence of any particular node (as in hashing). In fact, the organisation pattern of tuples emerges from the configuration of nodes, tuple templates, and connectivity of nodes. This emergent pattern adapts to variations in environments including node failure.

After presenting the devised strategy in Section 4.3, the resulting approach is initially

applied on two simple network instances (Section 4.5.2). Then, Section 4.5 reports a performance evaluation of the strategy applied to scale-free-topology networks. Finally, Section 4.6 introduces the so-called *over-clustering problem*, proposing a feasible solution.

4.2 SwarmLinda

LINDA is a coordination model based on associative memory as a communication paradigm. LINDA provides processes with primitives enabling them to store and retrieve tuples from tuple spaces. Processes use the primitive `out` to store tuples. They retrieve tuples using the primitives `in` and `rd`; these primitives take a template (a definition of a tuple) and use associative matching to retrieve the desired tuple—while `in` removes a matching tuple, `rd` takes a copy of the tuple. Both `in` and `rd` are blocking primitives, that is, if a matching tuple is not found in the tuple space, the process executing the primitive blocks until a matching tuple can be retrieved.

SwarmLinda uses several adaptations of algorithms taken from the abstraction of natural multi-agent systems [BDT99, Par97]. Over the past few years, new models originating from biology have been studied in the field of computer science [BDT99, KE01]. In these models, actors sacrifice individual goals (if any) for the benefit of the collective. They act extremely decentralised, carrying out work by making purely local decisions and by taking actions that require few computations, thus improving scalability. These models have self-organisation as one of their main characteristic. Self-organisation can be defined as a process where the entropy of a system (normally an open system) decreases without the system being guided or managed by external forces. It is a phenomenon quite ubiquitous in nature, in particular in natural forming swarms. These systems exhibit a behaviour that seems to surpass the sum of all the individuals' abilities [BDT99, Par97].

To bring the aforementioned ideas into SwarmLinda, we interpret the "world" of nodes as a graph in which ants search for food (similar tuples), leaving trails to successful searches. Adopting this approach, one can produce a self-organised pattern in which tuples are non-deterministically stored in specific nodes according to their characteristics (e.g. size, template).

The above is just an illustration. A more realistic SwarmLinda should consider a few principles that can be observed in most swarm systems [Par97]:

Simplicity: Swarm individuals are simple creatures, doing no deep reasoning and implementing a small set of simple rules. These rules lead to the emergence of complex behaviours.

Dynamism: Natural swarms adapt to dynamically changing environments. In open distributed LINDA systems, the configuration of running applications and services changes over time.

Locality: Swarm individuals observe their direct neighbourhood and make decisions based on their local view.

LINDA systems do not define the idea of ants or food. The description of SwarmLinda is based on the following abstractions: the *individuals* are active entities that are able to observe their neighbourhood, move in the environment, and change the state of the environment in which they are located; the *environment* is the context in which individuals work and observe; the *state* is an aspect of the environment that can be observed and changed by individuals.

We aim at optimising the distribution and retrieval of tuples by dynamically determining storage locations for them based on the template of that particular tuple. It should be noted that we do *not* want to program the clustering but rather make it emerge from the algorithms implemented through the mechanism of self-organisation. Note that according to Camazine [CDF+01], self-organisation refers to the various mechanisms by which patterns, structures and order emerge spontaneously within the system. Here, the clustering of tuples is the pattern that emerges in the system.

SwarmLinda describes four algorithms [TM03]: tuple distribution, tuple retrieval, tuple movement, and balancing template. This chapter deals only with issues related to tuple distribution.

4.2.1 Tuple Distribution

The process of tuple distribution or tuple sorting is one of the most important tasks to be performed in the optimisation of coordination systems. It relates primarily to the primitive `out` as this is LINDA's way of allowing a process to store information. Tuple distribution in SwarmLinda stores tuples based on their type (template), so that similar tuples stay close to each other. To achieve this abstraction we see the network of SwarmLinda nodes as the terrain in which `out`-ants roam. These ants have to decide at each hop in the network, whether or not the storage of the tuple should take place. The decision is made stochastically but biased by the amount of similar tuples around the ant's current location.

For the above to work, there should be a guarantee that the tuple will eventually be stored. This is achieved by having an aging mechanism associated with the `out`-ant. For every unsuccessful step the ant takes, the probability of storing the tuple in the next step increases and is guaranteed to eventually reach 1 (one).

The similarity function is another important mechanism. Note that it may be too restrictive to have a monotonic scheme for the similarity of two tuples. Ideally we would like to have a function that says *how similar the two tuples are* and not only if they are exactly of the same template. Later in Section 4.3, we describe the similarity mechanism used in our experiments.

Now compare the above with a standard approach based on hashing. As a tuple needs to be stored, a hash function is used to determine the node where to place the tuple. This

approach, although efficient in closed systems, is inappropriate to dynamic open cases because:

1. The modification of the hash function to include new nodes is not trivial.
2. The clustering of tuples is quite excessive. All the tuples of the same kind will (deterministically) be placed on the same node.
3. Due to 2, the system is less fault tolerant. The failure of a node can be disastrous for applications requiring the tuple kind(s) stored in that node.

Our approach is based on the brood sorting algorithm. We demonstrate that it is possible to achieve good entropy for the system (level of organisation of tuples) without resorting to static solutions such as hashing. The approach proposed here adapts to changes in the network, including failures.

4.3 A Solution for Tuples Distribution

Tuple distribution that can work well when failures are commonplace is still an open issue in the implementation of large scale distributed tuple spaces [MT03]. Several approaches for distributing tuple spaces have been proposed [Tol98, CLZ98, WMLF98], but none of them has proven useful (without modifications) in the implementation of failure-tolerant distributed tuple spaces [MT03].

In order to find a solution to this problem, we took inspiration from self-organisation, in particular from *swarm intelligence*. In the past years, many models deriving from biology and natural multi-agent systems – such as ant colonies and swarms – have been studied and applied in the field of computer science [MMTZ06]. The solution to the distribution problem adopted in SwarmLinda is dynamic and based on the concept of *brood sorting* as proposed by Deneubourg *et al.* [DGF+91].

We consider a network of distributed tuple spaces, in which new tuples can be inserted using the LINDA primitive `out`. Tuples are of the form $N(X_1, X_2, ..., X_n)$, where N represents the tuple name and $X_1, X_2, ..., X_n$ represent the tuple arguments. A good solution to the distribution problem must guarantee the formation of clusters of tuples in the network. More precisely, tuples belonging to the same template should be stored in the same tuple space or in neighbouring tuple spaces. Furthermore, tuples with a similar template should be stored in near tuple spaces. Note that this should all be achieved in a dynamically and adaptive way.

In order to decide how similar a tuple and a template are, we need a *similarity function* defined as `similarity(tu,te)`, where `tu` and `te` are input arguments: `tu` represents a tuple and `te` represents a tuple template. The values returned by this function are floating point values between `0` and `1`:

- `0` means that tuple `tu` does not match template `te` at all; complete dissimilarity.
- `1` means that tuple `tu` matches template `te`; complete similarity.
- Other values mean that `tu` does not match perfectly `te` but, nonetheless, they are not completely dissimilar.

In the following, we provide a description of the process involved in the distribution mechanism, when an `out` operation occurs in a network of distributed tuple space. For the sake of simplicity, we assume only one tuple space for each node of the network.

Upon the execution of an `out(tu)` primitive, the network, starting from the tuple space in which the operation is requested, is visited in order to decide where to store tuple `tu`. According to *brood sorting*, the `out` primitive can be viewed as an ant, wandering in the network searching the right tuple space to drop tuple `tu`, that is, the carried food. The SwarmLinda solution to the distribution problem is composed of the following phases:

1. *Decision Phase.* Decide whether to store `tu` in the current tuple space or not.
2. *Movement Phase.* If the decision taken in the previous phase is not to store `tu` in the current tuple space, choose the next tuple space and repeat the process starting from 1.

4.3.1 Decision Phase

During the decision phase, the `out`-ant primitive has to decide whether to store the carried tuple `tu`-food in the current tuple space. This phase involves the following steps:

1. Use the *similarity function*, calculate the concentration F of tuples having a template similar to `tu`.
2. Calculate the probability P_D to drop `tu` in the current tuple space.

The concentration F is calculated considering all the templates for which tuples are stored in the tuple space. F is given by:

$$F = \sum_{i=1}^{m} (q_i \times similarity(tu, T_i)) \tag{4.1}$$

where m is the number of templates in the current tuple space, q_i is the total number of tuples matching template T_i. Note that $0 \leq F \leq Q$, where Q is the total number of tuples within the current tuple space.

According to the original brood sorting algorithm used in SwarmLinda [DGF+91], the probability P_D to drop `tu` in the current tuple space is given by:

$$P_D = \left(\frac{F}{F + K} \right)^2 \tag{4.2}$$

Differently than the original idea in brood sorting, here the value of K is not a constant; K represents the number of steps remaining for the `out(tu)` primitive, namely, the number of tuple spaces that an `out`-ant can still visit. When an `out` operation is initially requested on a tuple space, the value of K is set to *Step*, that is, the maximum number of tuple spaces each `out`-ant can visit. In the experiments that follow, we assume *Step* as a parameter specific of the network topology we are using.

Each time a new tuple space is visited by an `out`-ant without storing the carried tuple `tu`, K is decreased by 1. When K reaches 0, P_D becomes 1 and `tu` is automatically stored in the current tuple space independently of the value of F. K is adopted to implement an *aging mechanism* to avoid having `out`-ants wander forever without being able to store the carried tuple `tu`. If $K > 0$ and the tuple `tu` is not stored in the current tuple space, a new tuple space is chosen among the neighbours of the current one. The following section provides a description of the process involved in the choice of the next tuple space.

4.3.2 Movement Phase

The movement phase occurs when the tuple carried by an `out` is not stored in the current tuple space. This phase has the goal of choosing, from the neighbourhood of the current tuple space, the best neighbour for the next hop of the `out`-ant. The best neighbour is the tuple space with the highest concentration F for the carried tuple `tu`. According to self-organisation principles, the choice of the next tuple space is local, i.e. based only on the *neighbourhood* of the current tuple space.

If we denote by n the total number of neighbours in the neighbourhood of the current tuple space, and F_j the concentration of tuples similar to `tu` in neighbour j (obtained using Equation 4.1), we can then say that the total number of tuples similar to `tu` in the neighbourhood is given by:

$$\mathrm{Tot}_{tu} = \sum_{j=1}^{n} F_j \tag{4.3}$$

The probability P_j of having an `out`-ant move to neighbour j, is calculated by:

$$P_j = \frac{F_j}{\mathrm{Tot}_{tu}} \tag{4.4}$$

Adopting this equation for each neighbour, we obtain:

$$\sum_{j=1}^{n} P_j = 1 \tag{4.5}$$

The higher the value of P_j for a neighbour j, the higher the probability for that neighbour to be chosen as the next hop of the `out`-ant. After a new tuple space is chosen, the whole process is repeated starting from the decision phase (as described in Section 4.3.1).

4.3.3 A Case Study

In order to provide an example of the SwarmLinda distribution mechanism, we adopt the case study presented in Figure 4.14. The figure shows a network of four tuple spaces TS_1, TS_2, TS_3, TS_4: in TS_1 the insertion of a new tuple $a(1)$ is being executed. As reported in the figure, the tuple spaces contain tuples belonging to one of four different templates: $a(X)$, $b(X)$, $c(X)$ and $d(X)$.

For the sake of simplicity, we assume $Step = 1$, and we use a very simple *similarity function*, given by:

$$similarity(tu, T) = \begin{cases} 1 & \text{if tuple } tu \text{ matches template } T, \\ 0 & \text{otherwise.} \end{cases}$$

Using the SwarmLinda distribution mechanism, the first step is the execution of the decision phase on tuple space TS_1. Tuple space TS_1 is characterised by the following concentration values:

$$q_{a(X)} = 6,\ q_{b(X)} = 3,\ q_{c(X)} = 2,\ q_{d(X)} = 0$$

where $q_{a(X)}$ is the concentration of tuples matching template $a(X)$ and so on. Given these values, the concentration F_1 (of tuples similar to $a(1)$) is calculated using Equation 4.1:

$$\begin{aligned} F_1 = {} & similarity(a(1), a(X)) \times q_{a(X)} \\ & + similarity(a(1), b(X)) \times q_{b(X)} \\ & + similarity(a(1), c(X)) \times q_{c(X)} \\ & + similarity(a(1), d(X)) \times q_{d(X)} \end{aligned}$$

and, replacing the symbols with the values we get:

$$F_1 = 1 \times 6 + \ 0 \times 3 + \ 0 \times 2 + \ 0 \times 0 = 6$$

Then, according to Equation 4.2, the probability P_D to drop $a(1)$ in TS_1 is:

$$P_D = \left(\frac{F_1}{F_1 + K}\right)^2 = \left(\frac{6}{6+1}\right)^2 \approx 0.734$$

Supposing that $a(1)$ is not stored in TS_1 – i.e. the calculated P_D is not satisfied – we have to choose the next tuple space among the neighbours of TS_1. The neighbourhood of TS_1 is composed of two tuple spaces: TS_2 and TS_4. If we assume $F_2 = 2$ and $F_4 = 8$, the probabilities to move in TS_2 or TS_4 are:

$$P_2 = \frac{F_2}{F_2 + F_4} = \frac{2}{10} = 0.2$$
$$P_4 = \frac{F_4}{F_2 + F_4} = \frac{8}{10} = 0.8$$

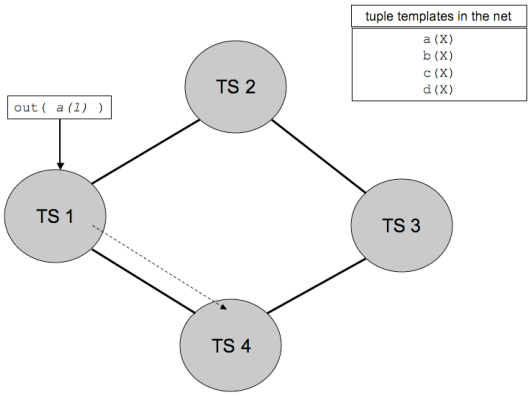


Figure 4.1: Case study: a network composed of four tuple spaces.

Before moving to the next tuple space, the value of K is decreased by 1: in this case the new value of K is 0. Since $K = 0$, $a(1)$ will be stored in the next tuple space, independently of the chosen tuple space.

4.4 Experimental Results

The results described in this section are based on an executable specification of SwarmLinda developed in the stochastic simulation framework presented in Chapter 2. As previously explained, our stochastic simulation framework enables a rapid prototyping and modelling of many complex systems, like SwarmLinda and the collective sort problem presented in Chapter 3. The next section briefly describes the executable specification of SwarmLinda. Then, before showing the results, we report a description of the methodology adopted for running the simulations. In particular, these simulations were performed on two different instances characterised by different network topologies.

4.4.1 Executable Specification of SwarmLinda

The executable specification of SwarmLinda is based on the stochastic simulation framework introduced in Chapter 2. According to this framework, the specification adopts the concept of *stochastic transition system*. The stochastic functions provided by the framework enable the implementation of the *unpredictable time evolution* proper of complex systems.

For the sake of brevity the complete specification of SwarmLinda is not presented here: the interested reader can refer to Appendix C for the complete code of the specification. In the following, we report the *similarity function*, defined in our specification using the MAUDE syntax:

```
op similarity : Term Term -> Float [comm] .
```

```
vars N1 N2 : Term . vars L1 L2 : TermList .

ceq similarity((N1(L1)),(N2(L2))) =
   0.0 if N1 =/= N2 .
eq similarity((N1(L1)),(N1(L2))) =
   0.6 + similarity(L1,L2) .
eq similarity((N1,L1),(N2,L2)) =
   similarity(L1,L2) .
eq similarity(nil,(N1,L1)) =
   -0.1 + similarity(nil,L1) .
```

Given two arguments of type `Term` (representing a tuple and a template), the function returns a `Float` value representing the degree of similarity between the tuple and template. According to this definition, considering a tuple $t(a_1, ..., a_n)$ and a template $t'(X_1, ..., X_m)$, the resulting behaviour of the function is:

$$similarity(t(a_1, ..., a_n), t'(X_1, ..., X_m)) =$$
$$= \begin{cases} 0.6 & \text{if tuple } t = t' \text{ and } n = m, \\ 0.6 - (0.1 \times \|m - n\|) & \text{if tuple } t = t' \text{ and } n \neq m, \\ 0 & \text{if } t \neq t'. \end{cases}$$

The similarity value is: *(i)* 0.6, if the tuple has the same name and the same number of arguments as the template; *(ii)* $0.6 - (0.1 \times \|m - n\|)$, if the tuple has the same name of the template but a different number of arguments; *(iii)* 0, if tuple name t is not equal to template name t'.

We can see that, when $t = t'$ and $n = m$, the value returned by this *similarity function* is different from the value returned by the *similarity function* described in Section 4.3. The maximum value returned by the *similarity function* – the value returned when a tuple completely matches a template – affects the maximum value of probability P_D (Equation 4.2). More precisely, the lower the maximum value that the function returns, the lower the value of P_D considering the same F and K (see Equations 4.1 and 4.2). Since we want to make the system adaptable to unpredictable state changes, we start choosing an upper-bound value for similarity lower than 1.

A tuple space is modelled by using the following (Maude) syntax

```
< Id @
 ['queue @(out(T1) | ... | out(Tn))]  |
 ['tot @ Tot] |
 ['conc @({TE1,Q1} | ... | {TEk,Qk})] |
 ['neighbours @(Id1 ... Idm)]
 >
```

where:

- `Id` is a natural number representing the identifier of a tuple space.
- `['queue @(out(T1) | ... | out(Tn))]` represents the input queue of tuple space `Id`, with the `out` operations to be executed.
- `['tot @ Tot]` represents the total concentration of tuples in tuple space `Id`, where `Tot` is a natural number describing such a quantity.
- `['conc @(TE1,Q1 | ... | TEk,Qk)]` represents the concentrations `Q1,...,Qk` of tuples for the different templates `TE1,...,TEk` within tuple space `Id`.
- `['neighbours @(Id1 ... Idm)]` represents the neighbours `Id1,...,Idm` of tuple space `Id`.

A network of tuple spaces is defined using the syntax

```
TS1 | TS2 | ... | TSq | ['steps,S]
```

where:

- `TS1...TSq` represents a network of q tuple spaces, each defined according to the syntax above.
- `['steps,S]` implements the *aging mechanism*, where `S` is the maximum number of steps, as described in Section 4.3.1.

4.4.2 Methodology

We want our distribution mechanism to achieve a reasonable distribution of tuples. Tuples having the same template should be clustered together in a group of nearby tuple spaces. To this end, the concept of *global system entropy* H introduced in Section 3.2.4 seems an appropriate metric to describe the degree of order reached in the network. In particular, the lower the value of H, the higher the degree of order in the network considered.

For each simulated network, we performed series of 20 simulations, using each time different values for the *Step* parameter (representing the maximum number of steps a tuple can take). One run of the simulator consists of the insertion of tuples in the network of tuple spaces – via `out` primitives – until there are no pending `out` to be executed in the entire network.

After the execution of a series of 20 simulations, the value of the *spatial entropy* H of the network is calculated as the average of the single values of H resulting from each simulation; we call this value *average spatial entropy* (H_{avg}). For each network topology presented in the next section, we considered only tuples of four different templates: $a(X)$,

$b(X)$, $c(X)$, and $d(X)$. Therefore, the possible values returned by the *similarity functions* are: *(i)* 0.6, if a tuple matches the considered template, *(ii)* 0, otherwise.

The following section presents the results obtained by simulating two different instances of network.

- *Collective sort instance*: a network of 4 tuple spaces completely connected, that is equal to the *collective sort case* discussed in Chapter 3.
- *Star instance*: a network of 6 tuple spaces characterised by a star-like topology. Note that this is used because of its contrast with collective sort. In collective sort we have nodes forming a clique. The star, instead, contrasts the results with networks that are not fully connected.

Despite of their simplicity, these networks let us appreciate the behaviour of the proposed scheme in quite distinct scenarios. Note that the values for the parameters (such as the *Step* parameter) are chosen to reflect the size of the network.

4.4.3 Star Instance

The network topology for the *star instance* is reported in Figure 4.2. As depicted in the figure, we verified the performance of our SwarmLinda distribution mechanism simulating the insertion of 15 tuples of each template. The insertion of the tuples takes place in each tuple spaces of the network. Consequently, at the end of a simulation, we have 90 tuples per template in the network.

A first set of simulations was run considering the network initially empty. We considered different values of the *Step* parameter in the range $[0-40]$, performing for each value a series of 20 simulations. In particular, the situation with $Step = 0$ corresponds to the case in which our distribution mechanism is turned off. Indeed, since no steps are allowed, the tuple carried by each `out` is stored in the tuple space in which the `out` operation is initially requested.

The results related to this set of simulations are reported in Figure 4.3 (a). The figure shows the trend of H_{avg} considering different values for the *Step* parameter. As expected, using $Step = 0$, we obtained a value of H_{avg} equal to 1 since our distribution mechanism is not used. In other words, we are in a deterministic situation, since each tuple is stored in the tuple space in which the corresponding `out` occurs. Increasing the value of *Step* causes H_{avg} to decrease. In particular, with $Step = 16$, H_{avg} reaches the value 0.2, that is the lowest value for this set of simulations. Further increasing the value of *Step* has minimum effect to H_{avg}.

The SwarmLinda distribution mechanism works dynamically—each `out` has to store the carried tuple `tu` in a tuple space with a high concentration of tuples similar to `tu` without any previous knowledge about the status of the network. For this reason, even though $H_{avg} = 0.2$ does not correspond to complete clustering, nonetheless it represents

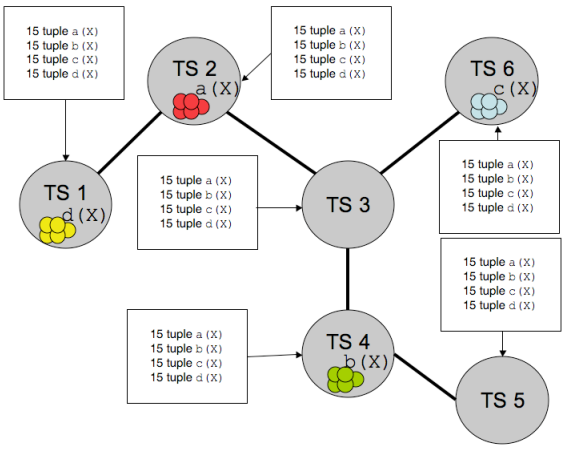
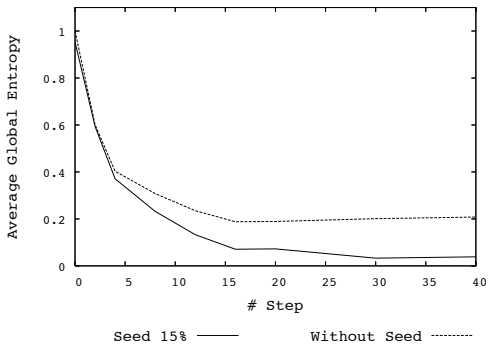


Figure 4.2: Star instance. This network was used in two different experiment: with seeding and without seeding. The picture depicts the seeds in 4 of the nodes but the network is assumed empty when we perform the no-seeding experiments.

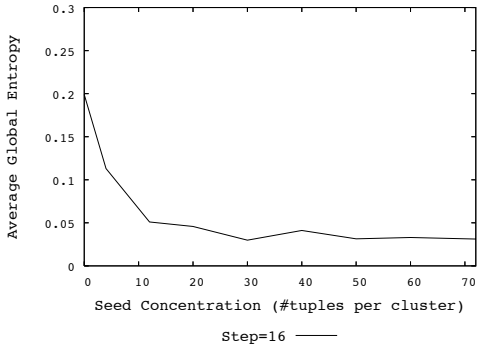
a good result. Moreover, it is important to find a good balance between the value of the *Step* parameter and the value of H_{avg} we want to achieve. Indeed, a too high value of *Step* may lead to a high network traffic. Hence, when we choose a *Step* value, it is important to consider not only the performance in terms of H_{avg}, but also the cost in terms of network traffic generated. Another point to be made is that complete clustering may actually not be desirable given that it makes the system less tolerant to failures. Excessive dependency on single nodes leads to the infamous single-point-of-failure problem.

The other experiment performed measures the behaviour of the tuple distribution mechanism when faced with a network where clustering is already present. In Figure 4.2 this consists of the configuration where the clusters depicted are considered in the execution. To perform this new set of simulations, we used the same number of tuples adopted for the previous case, but instead of considering the network initially empty, we considered an initial network configuration with clusters already formed. In particular, for each cluster of tuples belonging to a given template, we considered a concentration equal to 15% of the total number of tuples expected for that template. In the following, we refer to a cluster by the term *seed*. Figure 4.3 (a) also shows a comparison of the trend of H_{avg} obtained considering the presence of seeds against the results without seeds. The results are characterised by a better value of H_{avg}. This is expected since the network already had some level of organisation before the tuple distribution was executed.

Finally, we performed an analysis of the sensitivity of tuple distribution in relation to the seed concentration when $Step = 16$. A series of 20 simulations for different initial seed concentrations in the range $[0\% - 80\%]$ was performed. The results related to this case are shown in Figure 4.3 (b). As expected, the higher the value of seed concentration, the lower the value of H_{avg}.



(a) Trend of *average spatial entropy* H_{avg} with and without clusters (seeds) formed.



(b) Trend of *average spatial entropy* H_{avg} considering clusters of different size.

Figure 4.3: Charts showing the results for the Star instance.

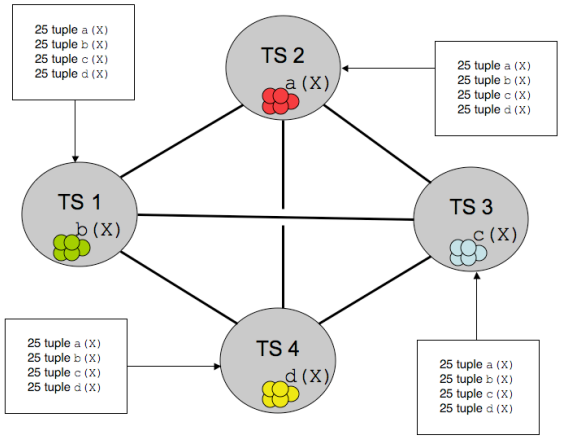
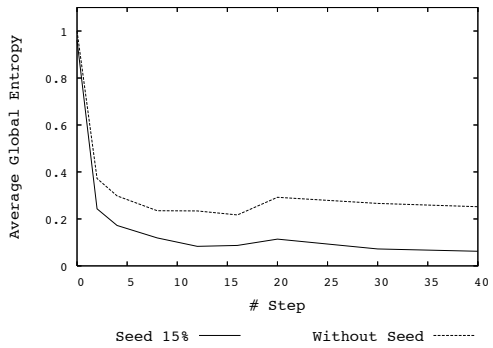


Figure 4.4: Collective sort instance. Again here the network shows nodes with seeds but these are only considered in the experiments that consider the existence of seeds.

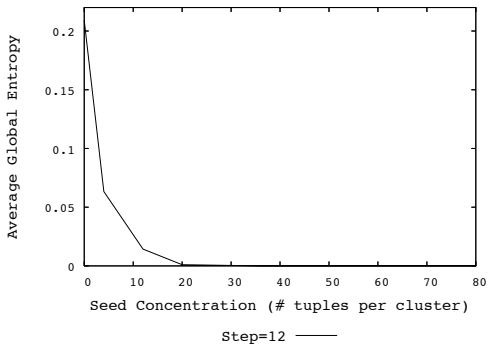
4.4.4 Collective Sort Instance

The network topology for the *collective sort instance* is shown in Figure 4.4. This instance corresponds to the case studied in Chapter 3, even though the approach adopted here is different.

It is worth comparing the two solutions that, although characterised by the same goal, are based on different approaches. Indeed, in the case discussed here the clustering process in realised *dynamically* as soon as an `out` operation is executed on a tuple space. Conversely, the approach described in Chapter 3 proceeds asynchronously in back-



(a) Trend of *average spatial entropy* H_{avg} with and without clusters (seeds) already formed.



(b) Trend of *average spatial entropy* H_{avg} considering clusters of different size.

Figure 4.5: Charts showing the results for the Collective Sort instance.

ground, independently of `out`-operation executions—in other words, it is conceived as a background service running concurrently to tuple insertions. This means that the tuple carried by an `out` is initially stored in the tuple space in which the operation occurs. At the same time, a set of software agents work in background with the task of moving tuples from a tuple space to another: the global goal is to achieve the complete clustering in the network.

We performed on the collective sort instance the same simulations executed previously on the star instance, using 100 tuples for each tuple template (Figure 4.4). Figure 4.5 (a) reports the trend of H_{avg} considering the network initially empty (without seeds). Figure 4.5 (a), also shows the trend of H_{avg} with seeds compared with the one resulting from the simulation without seeds already formed in the network. Finally, Figure 4.5 (b) shows the trend of H_{avg} considering different seed concentrations, using the same criterion described in Section 4.4.3. The results show the same trend of H_{avg}, already described in Section 4.4.3 for the star instance.

Now, we can compare the approach used here with that presented in Chapter 3. First of all, by considering the two solutions from the viewpoint of performance, we can notice that, while collective sort can achieve a perfect clustering of the tuples ($H_{avg} = 0$), the SwarmLinda solution can at most reach 0.2 as a minimum value of H_{avg}. However, the SwarmLinda solution tries to store a tuple in the right tuple space *upon* the execution of an `out` operation, whereas in the solution developed in Chapter 3 realises clustering by using software agents that work in background. Consequently, the latter solution seems to have a higher computational price than SwarmLinda.

The previous discussion has just the goal of highlighting the differences between the two solutions. As a consequence, the two approaches do not have to be viewed as conflict-

ing, but as alternatives, or better, as complementary approaches. Therefore, the choice of the solution to adopt should mainly depend on the specific application domain.

4.5 Applying SwarmLinda to Scale-Free Networks

In order to verify the applicability of the SwarmLinda distribution mechanism to large networks, we chose to perform simulations on scale-free topologies [BA99]. This choice was mainly driven by the consideration that almost every real network of computers features a scale-free topology—e.g. the WWW [BA99, Str01]. Next section reports a brief description of the scale-free networks used for our experiments, and the algorithm used to generate these networks.

4.5.1 Sample Scale-Free Networks

The networks used in our experiments were generated using the original *B-A Scale-Free Model Algorithm* presented by Barabási and Albert in [BA99]. This algorithm is briefly recalled in the following description. Given an initial small number m_0 of tuple spaces:

- at each step of the algorithm, a new tuple space is added and connected to $m < m_0$ already existing tuple spaces.
- The higher the degree k_i of an already existing tuple space i, the higher the probability of connecting the newly introduced tuple space to i.

Probability P_i to have the added tuple space connected to i is:

$$P_i = \frac{k_i}{\sum_j k_j}$$

so that already existing tuple spaces with a large number of connections have a high probability to get new connections. This phenomenon is also called *rich get richer*.

In order to generate our sample scale-free networks we chose $m_0 = m = 2$, and started with an initial network of two tuple spaces linked to each other. Then, each new tuple space was connected to two already existing ones according to the *B-A Scale-Free Model Algorithm*. We generated two scale-free networks composed of 30 and 100 tuple spaces.

The following section presents the results obtained by simulating the two scale-free instances. First, we performed simulations on a 30-tuple-space scale-free network in order to preliminary evaluate the performance of the distribution mechanism. Then after having proven that our approach is able to achieve a good tuple distribution, we executed simulations on the 100-tuple-space scale-free network, in order to further evaluate the performance of the distribution mechanism on larger networks.

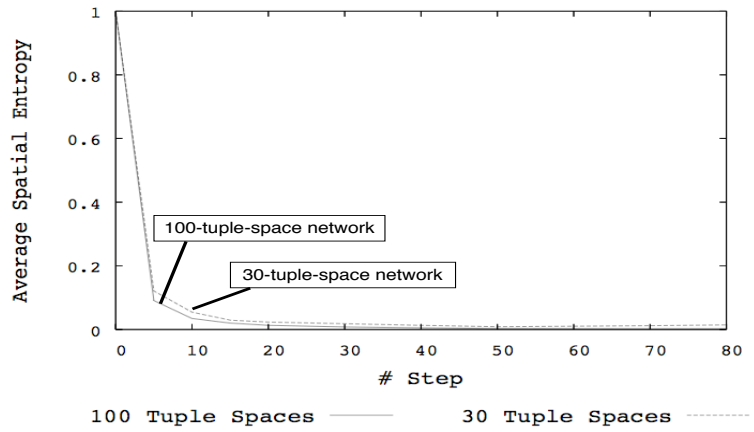


Figure 4.6: Trend of *average spatial entropy* H_{avg} resulting from the simulation of the two scale-free networks used in our experiments.

4.5.2 Simulation Results

Both network instances were simulated for values of *Step* in the range 0 to 80 steps, considering the occurrence of 60 *out* operations per tuple space—15 per tuple template. In particular, $Step = 0$ corresponds to a simulation performed without applying the distribution mechanism: indeed in this situation, every tuple is directly stored in the tuple space in which the corresponding `out` operation occurs.

The simulation results for the 30-tuple-space scale-free network are reported in Figure 4.6. Observing the results, we can clearly see that if we use a value of *Step* large enough to let an *out*-ant explore the network, the value of H_{avg} becomes small, meaning that the network features a high degree of clustering. For values of $Step > 20$, the distribution mechanism shows a high degree of insensitivity to different values of *Step*: this is due to the fact that, even though we choose $Step > 20$, the capability of our distribution mechanism to explore the entire network remains the same. For this reason, the trend of H_{avg} tends to an horizontal asymptote with a value approximately equal to 0.014.

Figure 4.6 shows the trend of H_{avg} for different values of *Step*, but it does not provide any information about actual tuple distribution in the network at the end of a simulation. Figure 4.7 reports a qualitative representation of tuple distribution in the network for two sample simulations chosen out of the 20 simulations used to calculate H_{avg} with $Step = 40$. Moreover, the tuple distribution shown in Figure 4.7 (left) corresponds to a *spatial entropy* value $H = 8.51 \times 10^{-3}$, while Figure 4.7 (right) reports a tuple distribution featuring $H = 1.42 \times 10^{-2}$. Although these final tuple distributions are different, the corresponding values of H demonstrate that we can achieve a quasi-perfect clustering. Indeed, Figure

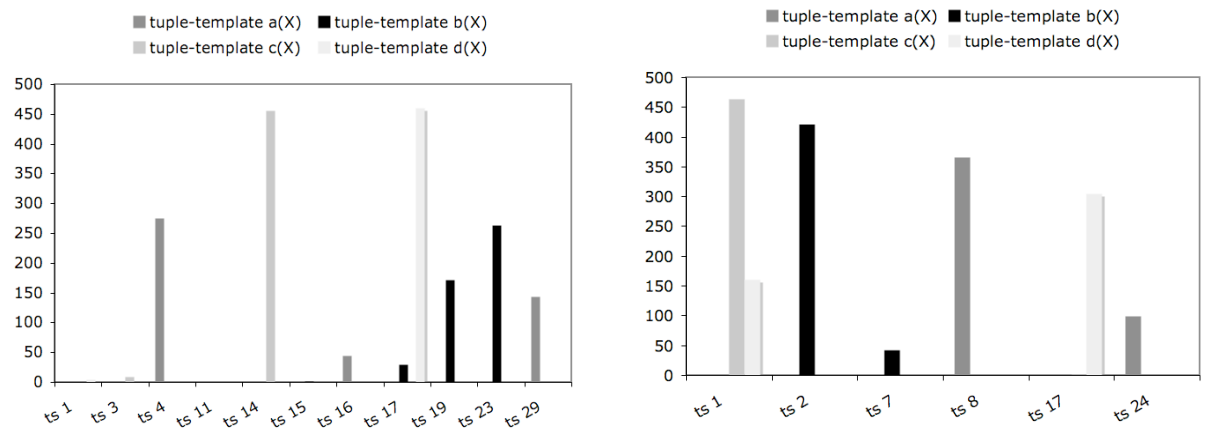


Figure 4.7: Final distribution of tuples obtained with $Step = 40$ for two different simulations on the 30-tuple-space scale-free network. *Spatial entropy* corresponding to these simulation: $H = 8.51 \times 10^{-3}$ (left) and 1.42×10^{-2} (right).

4.7 makes it clear the tendency of the distribution mechanism to organise tuples *per tuple template* amongst the tuple spaces composing the network.

Looking at the two tuple distributions reported in Figure 4.7 also makes it clear that even though the SwarmLinda distribution mechanism evolution does not allow to know in advance where tuples of a certain template will aggregate, a good level of information clustering (in terms of *spatial entropy*) can be achieved in every situation. Nonetheless, as depicted in Figure 4.9, the final tuple distribution pattern is sensitive not only to the values assumed by *Step*, but also to the initial conditions of the network. In fact, though the previous simulations were performed on an initially empty network, executing simulations on a network featuring one or more clusters already formed leads to a final tuple distribution where the inserted tuples tend to aggregate around the clusters. The outcome of a set of experiments – executed considering the presence of clusters in the network – is presented and discussed later.

The observed sensitivity of the distribution mechanism to system's initial conditions does not however mean that our approach is not self-organising. In fact, the capability to cluster information is independent of what Camazine *et al.* [CDF+01] call *external cue acting as a template for the aggregation of organisms*.

As Camazine *et al.* point out, the emergence of patterns in organism clustering is sometimes thought of as the result of a self-organising process even though it is not. As a consequence, we need to provide indications of the true self-organising nature of our tuple-distribution mechanism. One first possible indication is reported in Figure 4.8, showing the trend of concentration F for different simulations executed using $Step = 5$, 10 and 40. More precisely, F refers to the concentration of tuples similar to the current one in the tuple space where the tuple is stored.

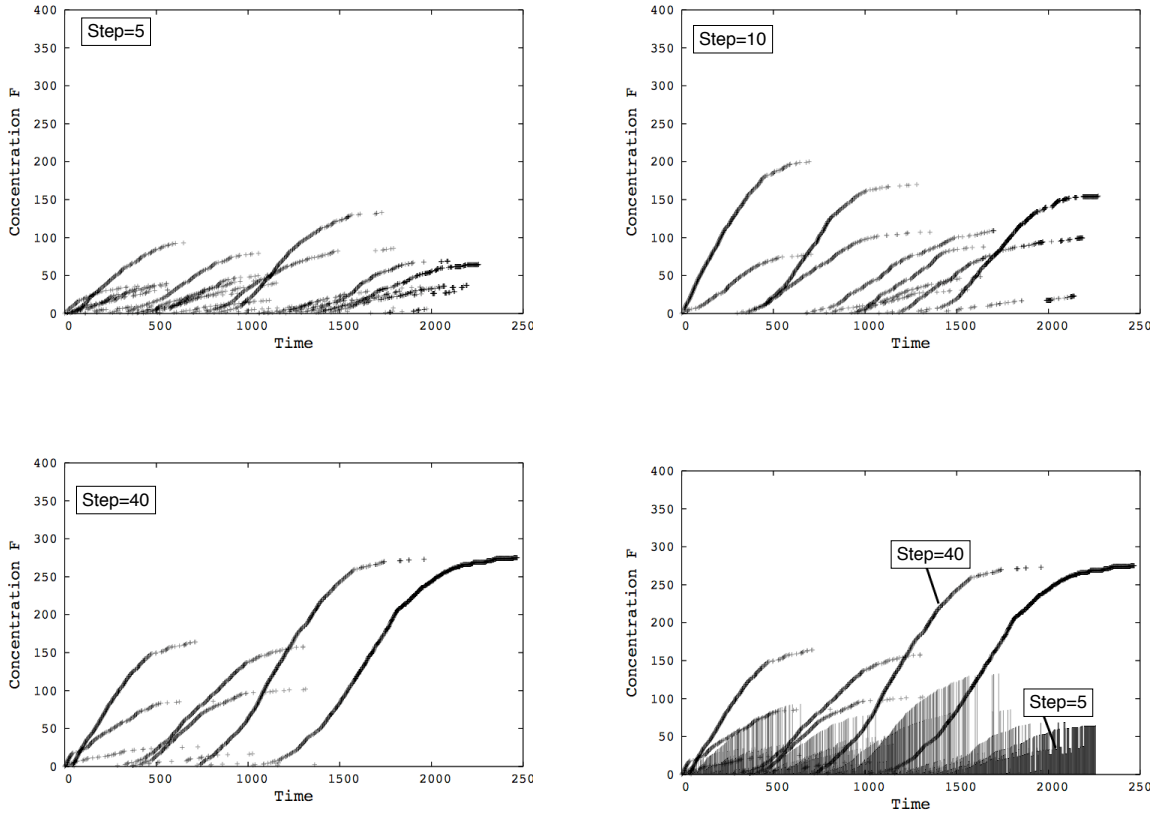


Figure 4.8: Evolution of concentration F for different simulations run on the 30-tuple-space scale-free network using $Step = 5$, 10 and 40. The bottom-right graph just highlights the differences between $Step = 5$ and $Step = 40$.

Even though Figure 4.8 allows to know neither the tuple space where a tuple is stored nor the tuple template of that tuple, we can easily recognise the formation of several clusters. This suggests that our SwarmLinda distribution mechanism exhibits a self-organising behaviour, making clusters of similar tuples emerge from an initial state characterised by an empty network. In particular, the emergence of such clusters is only driven by the local interactions occurring between a tuple space and its neighbours. Furthermore, looking at the different charts reported in Figure 4.8 makes it clearly visible an increasing order arising when higher values of $Step$ are used: note in particular the comparison reported in Figure 4.8 between the results for $Step = 5$ and the ones for $Step = 40$.

In spite of these results, we need a stronger argument to show that the pattern resulting from our distribution mechanism arises as a result of a self-organising process. In many

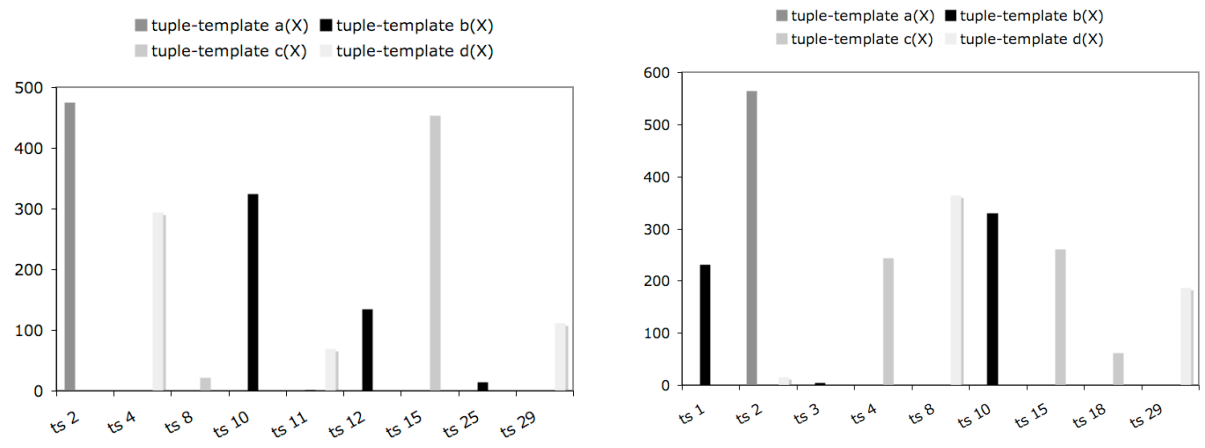


Figure 4.9: Final distribution of tuples obtained with $Step = 40$ for two different simulations on the 30-tuple-space scale-free network, considering a starting condition featuring clusters already formed on the network. Cluster size used in these simulations: 10 tuples (left) and 100 tuples (right).

cases of collective behaviour, organism clustering arises as a response of the individuals to an *external cue acting as a template*, and not as a natural outcome of a self-organised pattern formation [CDF+01]. In particular, such a behaviour has been observed in the stable fly of human and wood lice. In all of these cases, the aggregation process is the result of an external stimulus, an *environmental template* representing a fixed feature of the environment. Oppositely, in a self-organised aggregation the individuals respond to signals that are dynamic and affected over time from the behaviour of the individuals themselves.

One possible way to understand if one's system is really self-organising is to find an environmental template suspected to lead the aggregation of system's individuals, and see if that aggregation occurs even though the environmental template is removed. If, after removing the suspected environmental template, the individuals in the system fail to aggregate, we have a good indication that the aggregation of those individuals is based on an environmental cue and not on a self-organising process. Furthermore, systems achieving aggregation only by environmental templates are insensitive to different initial conditions, i.e. they come to the same final state independently of any variation in system's initial conditions. Nonetheless, other systems feature a behaviour driven by both a self-organising process and an external cue. Here, it would be desirable to find the relative contributions of these two factors to the aggregation process.

If we go back to our distribution mechanism, the role of *individual* is played by the *out*-ants wandering the network in the attempt to store the carried tuple, while the role of possible *external cue* can be played by clusters already formed in the network at the start of the simulation. The results shown previously are a first indication of the self-organising

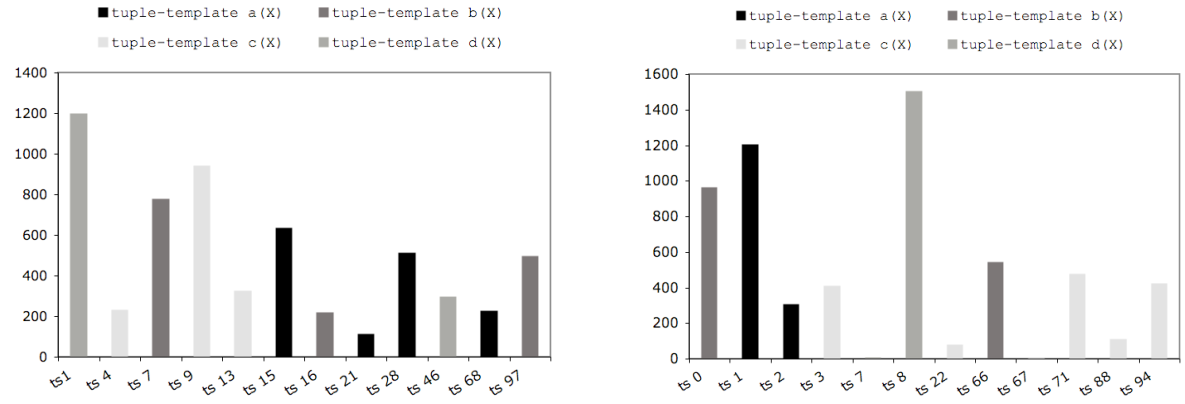


Figure 4.10: Final distribution of tuples obtained with $Step = 40$ for two different simulations on the 100-tuple-space scale-free network. *Spatial entropy* corresponding to these simulation: $H = 5.43 \times 10^{-3}$ (left) and 0 (right).

nature of our distribution mechanism. Even though we considered a network initially empty – with no clusters already-formed in the network – the results demonstrate that our distribution mechanisms can however achieve a strong level of information clustering.

In the attempt to provide a stronger argument of the self-organising nature of our distribution mechanism, we decided to perform further simulations considering the presence of clusters. More precisely, we ran a first set of simulations on the 30-tuple-space scale-free network considering four already formed clusters containing 10 tuples each: *(i)* 10 $a(X)$-template tuples in tuple space 2, *(ii)* 10 $b(X)$-template tuples in tuple space 10, *(iii)* 10 $c(X)$-template tuples in tuple space 15 and *(iv)* 10 $d(X)$-template tuples in tuple space 29. As in the previous experiments, we simulated the insertion of 60 tuples per tuple space—15 per tuple template. Figure 4.9 (left) shows the results of one of these simulations. It is easy to see that though the size of the clusters is small – compared to the number of tuples to be inserted in the network – they act as attractors for the tuples to be stored. Moreover, we are still able to achieve information clustering, but now the evolution of our distribution mechanism is driven not only by the time-evolving interaction between *out*-ants, but also by the presence of clusters. In fact, the distribution reported in Figure 4.9 (left) is characterised by the formation of clusters in the tuple spaces featuring the presence of clusters.

To better understand the role played by already formed clusters in the process of tuple clustering, we executed a second set of simulations characterised by the same initial conditions, but using larger clusters. The results of this second set of simulations are shown in Figure 4.9 (right). In this situation the attracting tendency of clusters is clearly recognisable and – due to clusters of larger size – is stronger than in the previous set of simulations.

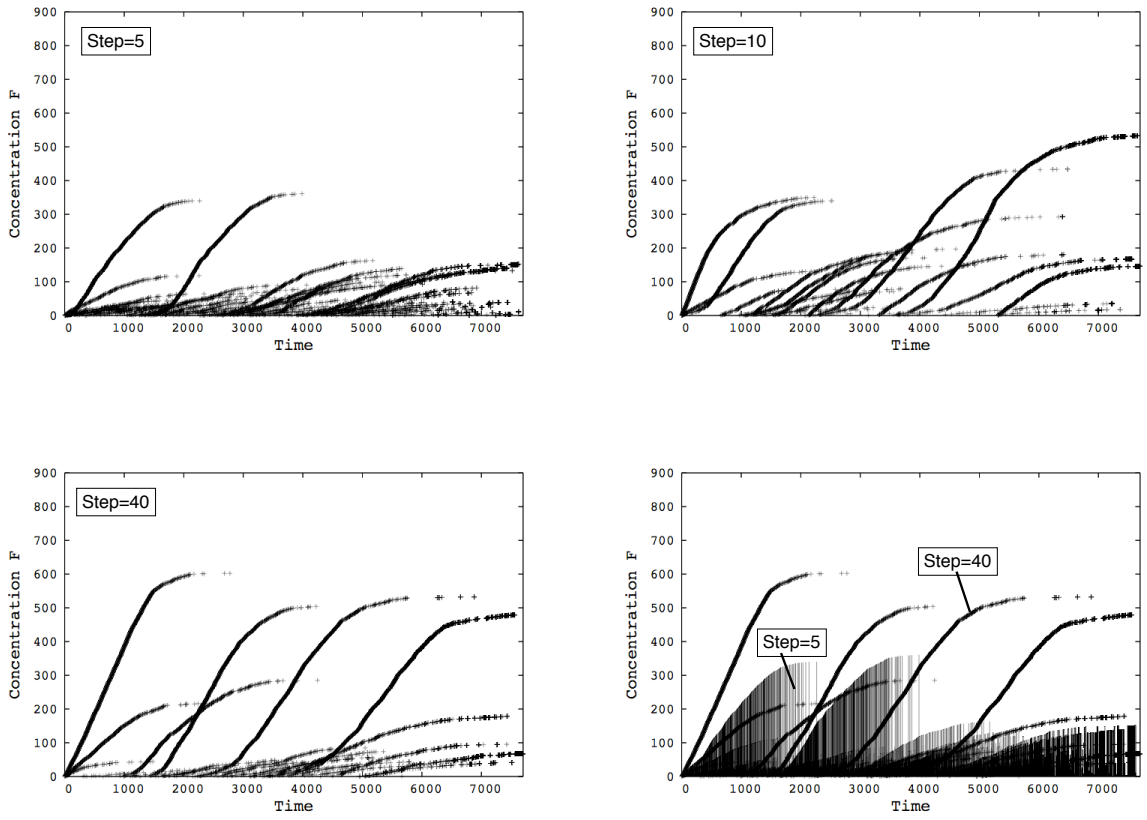


Figure 4.11: Evolution of concentration F for different simulations run on the 100-tuple-space scale-free network using $Step = 5$, 10 and 40. The bottom-right graph just highlights the differences between $Step = 5$ and $Step = 40$.

However, comparing these results with those obtained by simulating the network initially empty (Figure 4.7) shows that the distribution mechanism can achieve information clustering independently of the presence of clusters. Although already formed clusters tend to attract the evolution of larger clusters towards tuple spaces featuring the presence of clusters, they are not the only cause leading to information clustering. Indeed, the dynamic and evolving interactions between *out*-ants play also an important role in achieving a good level of clustering.

After having performed simulations on the 30-tuple-space scale-free network, in order to verify the behaviour of our mechanism on larger networks, we ran a new set of simulations on a 100-tuple-space scale-free network. Figure 4.6 shows the trend of H_{avg} for these simulations, and confirms the trend already observed on the 30-tuple-space scale-free net-

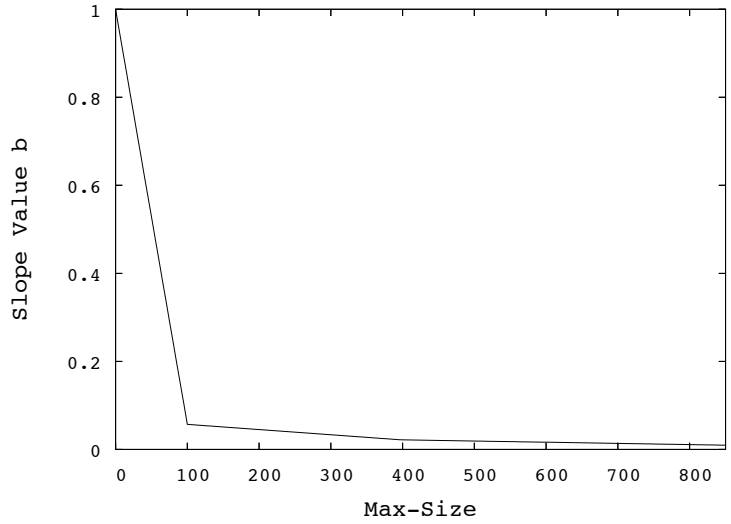


Figure 4.12: Function used to generate a slope value b given a *Max-Size* value.

work. Moreover, the value of the corresponding horizontal asymptote is equal to 0.007. Figure 4.10 reports the final tuple concentration achieved in two sample simulations executed with $Step = 40$, while Figure 4.11 shows the trend of F for different simulations executed with $Step = 5$, 10 and 40. All these results confirm the qualitative trend already observed on the 30-tuple-space scale-free network.

4.6 Avoiding Over-Clustering in SwarmLinda

As it should be clear by now, over-clustering can take place if the SwarmLinda algorithm is used without care. Over-clustering is the phenomenon occurring when a node becomes a strong aggregator of tuples belonging to the same kind. When such circumstances hold, the whole network is likely to become highly dependent on that node, resulting in a weaker system from the standpoint of robustness to failures.

4.6.1 A First Solution to Over-Clustering

In order to avoid over-clustering, we introduced the *Max-Size* parameter, that represents the maximum number of tuples allowed for each cluster. Then we exploit a *generalised logistic function* (a sigmoid function), so that probability P_D can be normalised according to a sigmoid curve. More precisely, we used an instance of the generalised logistic function,

characterised by the equation:

$$P'_D = P_D - \left[0.01 + \left(\frac{P_D - 0.01}{(1 + 0.5e^{-b(X-2m)})^2}\right)\right] \tag{4.6}$$

where P_D is the drop probability calculated by using Equation 4.2, b is the slope of the curve, and m is the value of the X variable at which we observe the maximum value of the curve derivative. Equation 4.6 defines a *complemented* generalised logistic function, since we want to obtain the maximum value of P'_D when X has a small value. Indeed, the value assigned to X is the concentration F calculated considering the current tuple to be stored. The value of m depends on the total number of tuples per template expected in the network, while the slope value b depends on the values chosen for *Max-Size*: the lower the value of *Max-Size*, the higher the value of b. Figure 4.12 shows the function used to obtain a value of b given a value of *Max-Size*.

According to Equation 4.6, as the number of tuples of a given template (the concentration F) approximates to *Max-Size*, the drop probability P'_D degenerates towards 0 (zero). More precisely, the P_D value calculated by Equation 4.2 is used as the maximum value returned by our generalised logistic function when $F = 0$. Then, depending on the concentration F calculated by Equation 4.1, the new value P'_D of drop probability is returned by adopting our generalised logistic function—again, a simple normalisation based on the current level of clustering.

In our experiments *Max-Size* is set to 100: this choice was based on the number of tuples stored in the network, and on the size of the network itself. Figure 4.13 presents the complemented generalised logistic function obtained with *Max-Size* = 100 and $P_D = 0.9$. In particular, the value of b is obtained by using the function shown in Figure 4.12 with *Max-Size* = 100. Then by applying the obtained value of b to Equation 4.6, we can generate the curve shown in Figure 4.13. According to Equation 4.6, this curve features a *lower asymptote* equal to 0.01, and an *upper asymptote* equal to P_D. An interesting future work would be to devise a solution based on a *dynamic Max-Size* parameter. In particular, we would like *Max-Size* to be a function of the clustering status, the size of the network, or perhaps the current entropy of a tuple space (its level of organisation).

Adopting this anti-over-clustering strategy, we performed a series of 20 simulations on the network reported in Figure 4.14. These simulations were performed by considering four different tuple templates ($a(X)$, $b(X)$, $c(X)$, $d(X)$) and the insertion of 210 tuples per template—in other words, for each template, 15 tuple insertions per tuple space. Simulations were performed by choosing *Step* = 30 and *Max-Size* = 100. Choosing template $c(X)$ as reference, Figure 4.15 compares the concentration of tuples of template $c(X)$ in the network with and without over-clustering. While in the case without anti-over-clustering we have a heavy concentration of $c(X)$ tuples on one tuple space, now the same amount of tuples is divided on two tuple spaces. More importantly, we do not lose clustering ability—the tuples are clustered in a group of close nodes rather than in individual nodes far from each other.

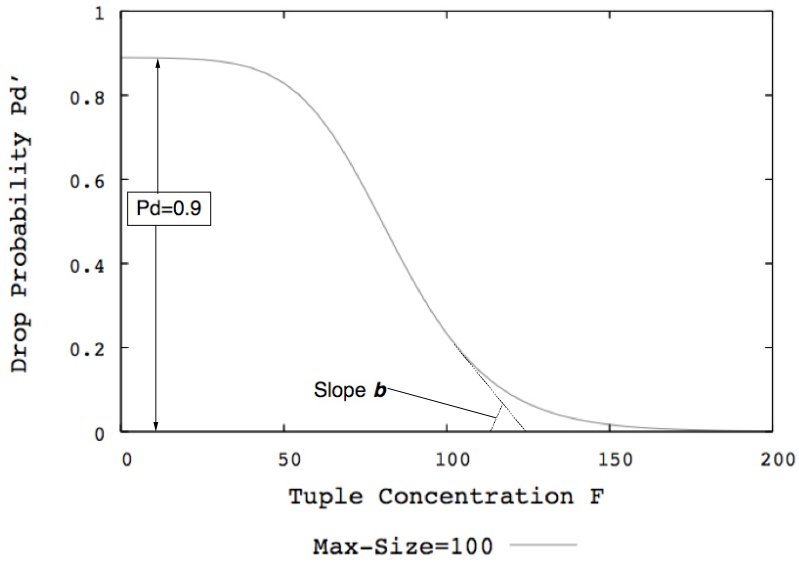


Figure 4.13: Complemented generalised logistic function obtained with *Max-Size* = 100 and an *upper asymptote* value $P_D = 0.9$.

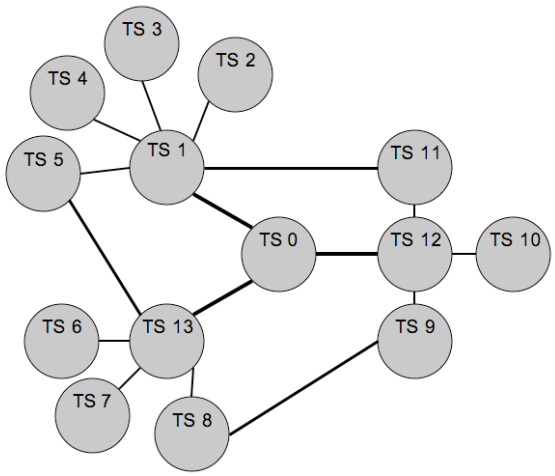


Figure 4.14: Topology used in the anti-over-clustering experiments.

4.6.2 Enhanced Anti-Over-Clustering Strategy

Although the approach described in Section 4.6.1 provides a fair solution to over-clustering, some improvements are needed. In particular, we need to modify the original SwarmLinda distribution mechanism presented in Section 4.3, adapting its behaviour to the anti-over-clustering strategy described in Section 4.6. Indeed, in Section 4.6.1 only the *decision*

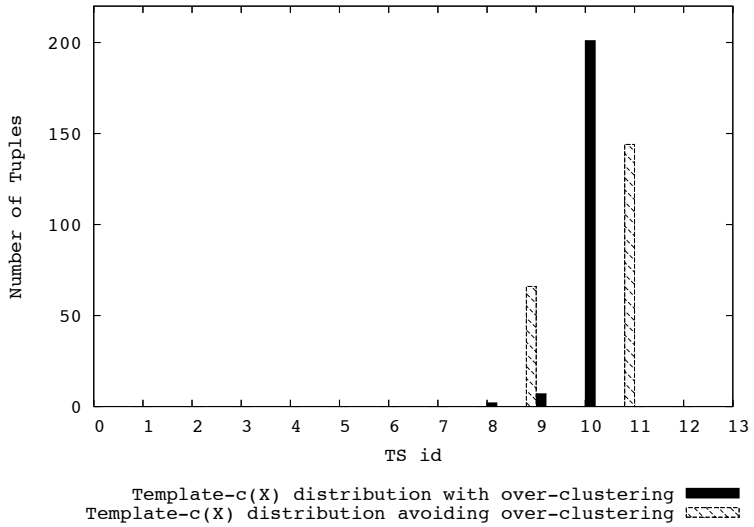


Figure 4.15: Distribution of tuples matching template $c(X)$ with and without over-clustering.

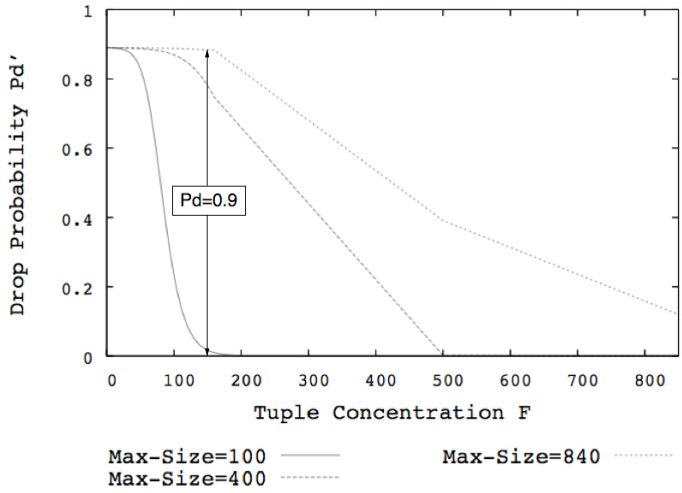


Figure 4.16: Complemented generalised logistic functions obtained using *Max-Size* = 100, 400 and 840, with an *upper asymptote* value $P_D = 0.9$.

phase of the SwarmLinda distribution mechanism is adapted to cope with over-clustering, whereas the *movement phase* is left unchanged. As a major consequence, when a tuple is

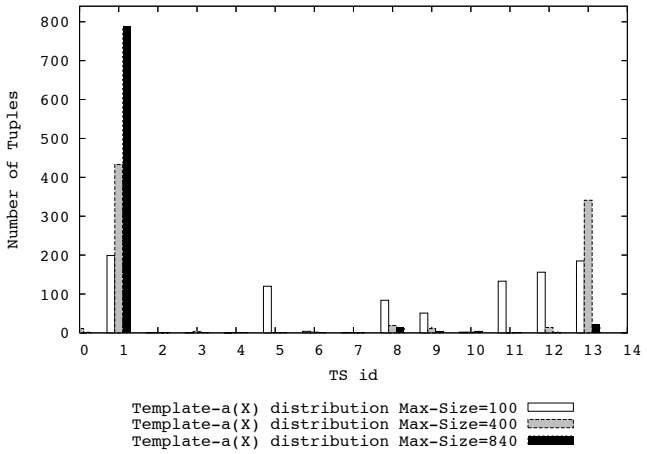


Figure 4.17: Enhanced anti-over-clustering strategy: comparison of the tuple distribution achieved with *Max-Size* = 100, 400 and 840.

not stored in the current tuple space, the choice of a good neighbour does not take into account over-clustering, making it possible to choose an over-clustered neighbour for the next hop. Since we wanted to have a distribution mechanism completely coherent with the anti-over-clustering strategy presented in Section 4.6.1, we developed a new *movement* phase by modifying Equation 4.4.

More precisely, the *new* probability P_j of having an `out`-ant move to neighbour TS_j becomes:

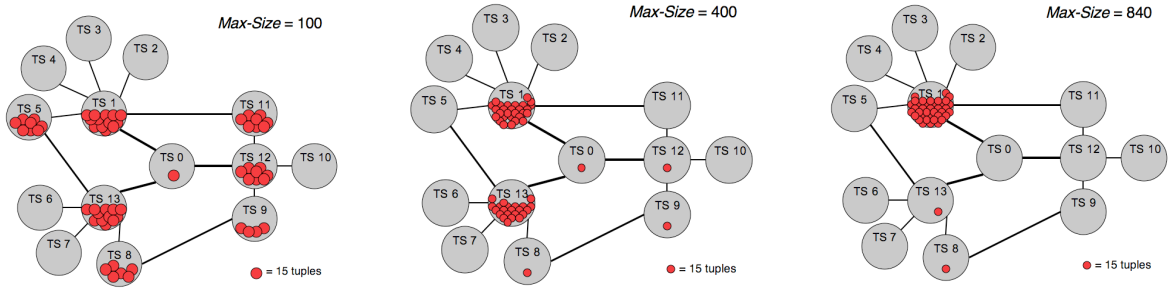
$$P_j = \frac{P'_j}{\sum_{k=1}^{n} P'_k} \tag{4.7}$$

where P'_j is obtained by Equation 4.6. Fundamentally, the new calculation for P_j considers how good P'_j is when compared to all the other probabilities of the neighbours. In other words, it takes P'_j as a relative value instead of an absolute one.

Again, the slope value b used to calculate P'_j depends on the chosen *Max-Size* value. The division by $\sum_{k=1}^{n} P'_k$ shown in Equation 4.7 is a normalisation that makes it possible to have:

$$\sum_{j=1}^{n} P_j = 1 \tag{4.8}$$

Adopting this new movement phase, we obtained a completely coherent SwarmLinda distribution mechanism characterised by an *enhanced* anti-over-clustering strategy. Note that the main advantage of this approach is to consider over-clustering even in the movement phase: though, since we are using a self-organised approach, we do not want to completely avoid the choice of an over-clustered tuple space, but rather, to make it less likely than other tuple spaces.



(a) Approximated tuple distribution with *Max-Size* = 100.

(b) Approximated tuple distribution with *Max-Size* = 400.

(c) Approximated tuple distribution with *Max-Size* = 840.

Figure 4.18: Charts reporting an approximated graphical representation of tuple distribution obtained with *Max-Size* = 100, 400 and 840.

The next section provides a description of the results achieved by applying the *enhanced* anti-over-clustering strategy on the scale-free network shown in Figure 4.14.

4.6.3 Result Evaluation of the Enhanced Anti-Over-Clustering Strategy

We performed a set of three simulations adopting the enhanced anti-over-clustering strategy on the same network reported in Figure 4.14. Since we were mainly interested in the observation of tuple distribution for different *Max-Size* values, we decided to use only tuple template $a(X)$. More precisely, for each simulation, we simulated the insertion of 840 tuples—60 per tuple space.

Each simulation was run adopting a different *Max-Size* value: in particular, we considered *Max-Size* equals to 100, 400 and 840. Every execution was performed using $Step = 30$. The choice of $Step = 30$ corresponds to a very critical condition for the enhanced anti-over-clustering strategy, since such a value, if we do not use any anti-over-clustering strategy, guarantees a situation of quasi-complete clustering. This choice was driven by the necessity of testing our enhanced anti-over-clustering strategy in the worst case.

Figure 4.16 shows the sigmoid functions corresponding to the *Max-Size* values chosen for the simulations. It is easy to see that the higher the *Max-Size* value, the lower the value of b. Given a value of F, a lower value of b generates a higher value of P'_D: this corresponds to a higher probability to store a tuple in the current tuple space. In other words, lower values of b allow the formation of larger clusters. In particular, the cluster size is directly related to the chosen *Max-Size* value: indeed, fixing *Max-Size*, the probability P'_D quickly decreases as the concentration F becomes close to the fixed *Max-Size* value.

The results obtained by running the three simulations are shown and compared in

Figure 4.17. The Figure makes it evident the influence of *Max-Size* on the cluster size. Indeed, while with *Max-Size* = 100 we obtained clusters characterised by an average size of 100 tuples, the simulation with *Max-Size* = 400 led to the formation of two clusters with an approximate average size of 400 tuples. Moreover, with *Max-Size* = 840 we obtained a quasi-perfect clustering, since the *Max-Size* value is equal to the number of tuples inserted in the network: as a consequence, even though this situation features over-clustering, we demonstrated that *Max-Size* can be used to tune the system behaviour in a custom way. These results show that we can use *Max-Size* to influence the tuple distribution in a self-organised fashion.

Figure 4.18 (a), (b) and (c) provide an alternative view of the tuple distribution resulting from the different *Max-Size* values adopted. More precisely, tuple distribution is shown with respect to network topology. It is easy to see that every value of *Max-Size* guarantees the formation of close clusters, maintaining in any case a good level of organisation in tuple distribution. We can also recognise that more connected tuple spaces have a higher probability to store a high number of tuples.

4.6.4 Coping with Over-Clustering: Concluding Remarks

In this section a solution to the over-clustering problem was proposed as an extension of the basic SwarmLinda strategy for tuple distribution. The work on avoiding over-clustering was initially driven by the consideration that over-clustering may occur even though we adopt self-organised solutions based on novel approaches. Using the proposed anti-over-clustering strategies, we executed some simulations considering the case without anti-over-clustering and the case with anti-over-clustering. In particular, this first solution to over-clustering was developed by considering the effect of over-clustering on the probability to store a tuple in a node. Then we improved this first anti-over-clustering strategy, making the entire SwarmLinda distribution mechanism coherent with the previously developed anti-over-clustering strategy. In this improved solution the effect of over-clustering is considered even in the movement phase, occurring when the considered tuple is not stored in the current tuple space. This enhanced anti-over-clustering strategy was tested with different *Max-Size* values, showing how *Max-Size* affects the tuple distribution in the network.

There are many improvements that need to be performed on this approach but, most importantly, we need to devise a truly self-organised solution able to *dynamically* adapt the *Max-Size* value. Indeed, this parameter drives the clustering behaviour as one can see in Figure 4.18. However a specific *Max-Size* value may be inadequate as system changes. For instance, a system featuring one tuple space containing 10 tuples may be considered over-clustered if 10 tuples are all the tuples in the system. What one would like is to make this value change as some of the characteristics change, such as the size of the network (number of nodes), the reliability of the nodes, the total tuple-space number in the system, and the number of tuple kinds in the system. However, the main challenge

consists in the fact that our approach is self-organising. As such, it should not be based on global parameters that are costly to maintain.

Last but not least, we still need to devise metrics for a more effective evaluation of the effect of clustering (with and without the approach to anti-over-clustering) in the survivability of SwarmLinda. Fault-tolerance and survivability of open LINDA systems is rarely tackled. We believe our approach improves the fault-tolerance of SwarmLinda and should make it a good choice for faulty systems.

5

Self-Organising Tuple Clustering and Sorting

In this chapter the abstract framework of self-organising coordination introduced in Chapter 2 is exploited for devising two applications inspired by corpse clustering and larval sorting in ant colonies, where a distributed tuple-space-based scenario is enhanced with adaptive tuple clustering and sorting, which can be regarded as generalisations of the collective sort problem and SwarmLinda. In particular, both the collective sort and SwarmLinda deal exclusively with the problem of tuple sorting from quite a specific perspective. In fact, while collective sort addresses the sorting of tuples already present in a tuple-space network featuring a flat topology, SwarmLinda – even though not relying on a specific topology – focusses on a tuple sorting mechanism based on out operations literally wandering the network so as to find a proper storage location for the carried tuples, i.e. new tuples are supposed to be stored in the proper area of the network as they are injected in the network itself. Here instead, the problems of tuple clustering and tuple sorting are addressed separately so as to provide a better clarification of the corresponding issues and devise solutions appropriate for generic-topology networks.

5.1 Self-Organisation for Tuple Organisation in Distributed networks

As clarified in Chapters 3 and 4, one of the most popular coordination scenario is based on the idea that agents in a distributed system can interact with one another through tuple spaces spread over the network, where tuples can be inserted and retrieved relying on so-called generative communication [Gel85, Gel89, OZ99]. This approach has been shown to support time and space decoupling, as well as to promote a clear separation between the computational part of the system, which should reside inside agents, and the coordination part of the system implemented by tuple spaces.

However, the position of tuples in such a distributed system is crucial since agents cannot simply try to randomly look for tuples: some searching strategy is required to

make sure that agents have at least a partial knowledge of the location of the tuples of interest—indeed, this is the main reason underlying the study of collective sort and SwarmLinda. A possible solution to this problem is to devise approaches to move tuples in the most proper tuple space, so as to ease the process of tuple localisation by agents. In fact, retrieving a tuple might be not so trivial. First of all, if the tuple space resides far from where an agent is currently situated, obtaining a tuple requires possibly expensive network operations. Most importantly, if the existence of a tuple is known but its position is not, an agent has to observe different tuple spaces before finding the right one. Instead, each agent should always be aware of the position of the tuples it is interested in. If this is unfeasible, even just some kind of awareness could be extremely helpful.

From a general perspective, one of the main reasons for dealing with the issue of tuple organisation and distribution is to ease the strategies adopted by agents to find specific kinds of tuples in distributed networks. Indeed, providing tuple-space-based systems with specific patterns of tuple distribution can improve the efficiency of the coordinated agents. In this scenario, the use of traditional centralised approaches becomes unfeasible for today's application domains, which feature a high degree of unpredictability and dynamism. For instance, agents of a distributed tuple-space-based system usually insert, move, and retrieve tuples in unpredictable ways both from a temporal and spatial viewpoint. As a consequence, the adoption of the self-organising conceptual framework shaped in Chapter 2 becomes crucial for providing adaptiveness and pattern emergence in coordination settings where state changes occur in a dynamic and unpredictable way. Accordingly, a possible solution is to provide agents with a background and online service for organising tuples in the network.

The issue of tuple distribution has been already dealt with in Chapter 3, where an online, distributed self-organising service – called *collective sort* – for exact tuple sorting in flat sets of tuple spaces was presented. In particular, collective sort was considered in a scenario featuring N tuple kinds and N fully connected tuple spaces. In this context, the role of tuple kind is played by tuple template. According to this scenario, the goal of collective sort is to adaptively sort tuples per kind so as to come to a final organisation where each tuple space stores only tuples of the same kind (template). Furthermore, also SwarmLinda, discussed in Chapter 4, deals with tuple distribution, though the approach is not based on a background service for sorting tuples, but on a mechanism to make new tuples find the proper storage place as they get inserted in the network.

This chapter instead considers separately the issues of *data clustering* and *data sorting* and adopts the self-organising framework depicted in Chapter 2. While the main inspiration for dealing with the latter issue derives from *larval sorting* in ants [BDT99], the former is dealt with by taking inspiration from *corpse clustering* [BDT99]. Corpse clustering is a phenomenon observed in many species of ants: corpses randomly distributed in the (physical) space tend to be organised and clustered by roaming ants through an increasing and emergent *aggregation process* [BDT99]. On the other hand, in larval sorting, larvae are picked up by ants according to larvae size and dropped elsewhere so as to

form clusters organised per larvae size [BDT99]. As a consequence, larvae with a similar size tend to be laid down in the same area by an increasing and emergent aggregation and segregation process.

Drawing inspiration from these natural phenomena, we focus on the problem of *tuple organisation* in distributed tuple-space networks from a quite general perspective. Tuple organisation can be viewed from a twofold perspective: *tuple clustering* and *tuple sorting*, which are key for today's tuple-space-based coordination systems. In tuple clustering the main focus is on defining a strategy to spatially aggregate tuples of the same kind – i.e. tuples carrying information of the same class – in clusters that form on the network in an emergent way. On the other hand, tuple sorting can be regarded as a generalisation of tuple clustering: tuples are not only aggregated but also segregated with respect to their kind. Accordingly, tuples of the same kind are aggregated in the same cluster, but kept segregated from tuples of other kinds.

In the follows, the solutions to tuple clustering and tuples sorting are presented and experimental results provided. In particular these solution can be viewed in terms of the features defined for the conceptual framework of self-organising coordination.

Topology — Topology is represented by the way tuple spaces are connected with one another. In other words, the space is shaped according to the specific topology adopted for the tuple-space network. Here, a location is represented by a tuple space, which can be regarded as a coordination medium. Accordingly, the whole tuple-space network is a distributed coordination medium.

Locality — Interactions in the network occur between neighbouring tuple spaces, which enact coordination rules to move tuples according to local criteria.

On-line character — Tuple spaces are provided with coordination rules reactive to locally occurring interactions. Such rules are online as they need to react to the dynamic evolution of tuple distribution in the network. This is an essential feature for providing tuple organisation with a truly self-organising coordination mechanism.

Time — The reaction rules enacted by tuple spaces are timed, i.e. reactive to time elapsing. In other words, these rules are executed whenever a given time interval has passed so that the tuple-organisation service is provided at a rate dependent on the chosen time interval.

Probability — The reaction rules enacted by tuple spaces are probabilistic, which can give a high degree of adaptiveness to unpredictable state changes.

5.2 Self-Organising Tuple Clustering

This section presents a solution to tuple clustering in distributed tuple-space networks that takes inspiration from the model of corpse clustering introduced by Deneubourg et

al. [BDT99, DGF+91]. Each tuple space in the network is provided with coordination rules, whose goal is to locally move tuples to neighbouring tuple spaces. As described in Section 5.1, this leads to the emergence of global tuple clustering only by local observations and actions. In fact, the rules enacted by each tuple space have only a partial view of system state and act accordingly by moving, if so, tuples to neighbouring tuple spaces. As prescribed by the LINDA model [Gel85], access to tuple space state is granted by performing tuple readings (`rd` operations), retrievals (`in` operations), and insertions (`out` operations).

Moreover, in order to exploit the metaphors coming from corpse clustering, tuples need to be enriched with some additional meta-data. The first meta-data to be added is a binary flag describing if the tuple is still or moving. While a *still* value of the flag describes a tuple that has not recently been moved, a *moving* value is assigned to recently moved tuples. When a still tuple is chosen for being moved, the value of the flag becomes *moving*. On the other hand, when a moving tuple is stored in a tuple space, the respective flag value becomes *still*. The second meta-data is represented by a value containing the tuple concentration of the last tuple space visited by a moving tuple. This is key to facilitate the storage of a tuple in tuple spaces whose tuple concentrations are higher than the ones observed in the tuple spaces previously visited by the tuple.

Correspondingly, the coordination rules applied by each tuple space are as follows:

1. a `rd` operation is performed on local space L, yielding a tuple t;
2. **if** t is *still*
   - t gets *moving* by a probability P_{mobile}
     - **if** t gets *moving*: $conc(t) = conc(L)$
3. **else** (t is *moving*)
   - a random number r between 0 and 1 is drawn
   - **if** $r < e^{-(conc(t)/conc(L))}$
     - t gets *still*
   - **else**
     - a neighbouring tuple space R is randomly chosen
     - t is moved to R

After having randomly read a tuple t from its tuple space L, there are two possible actions depending on the state of t. If t is a still tuple, t gets moving by a probability P_{mobile}, which represents the probability of a still tuple to get moving. Furthermore, the concentration of tuples perceived by t ($conc(t)$) is updated to the total concentration of tuples in L

($conc(L)$). If t is a moving tuple, it may be stored on the local tuple space: this decision is made by a probability calculated as

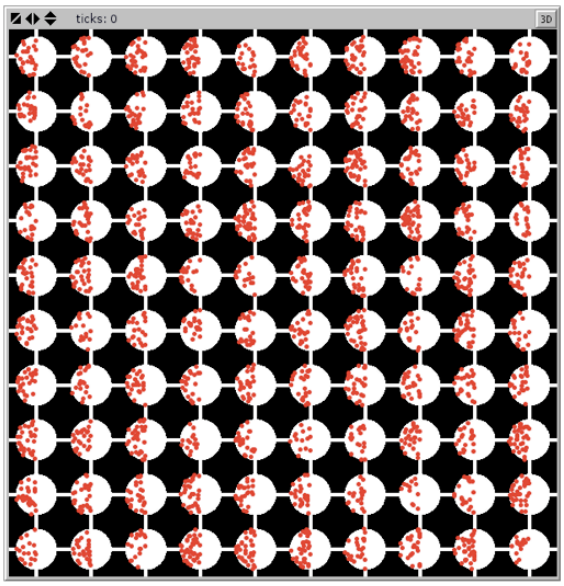
$$e^{-(conc(t)/conc(L))}$$

whose exponent is the relative concentration of tuples perceived by t during the last movement with respect to the total tuple concentration of L. In particular, a high value of relative concentration leads to a low probability of storing t in L, since we can suppose that tuple spaces previously visited by t feature higher concentration of tuples than L. On the other hand, low values of relative concentration increase the probability of storing t in L, as this is a clear clue that tuple spaces previously visited by t feature lower concentration of tuples than L.

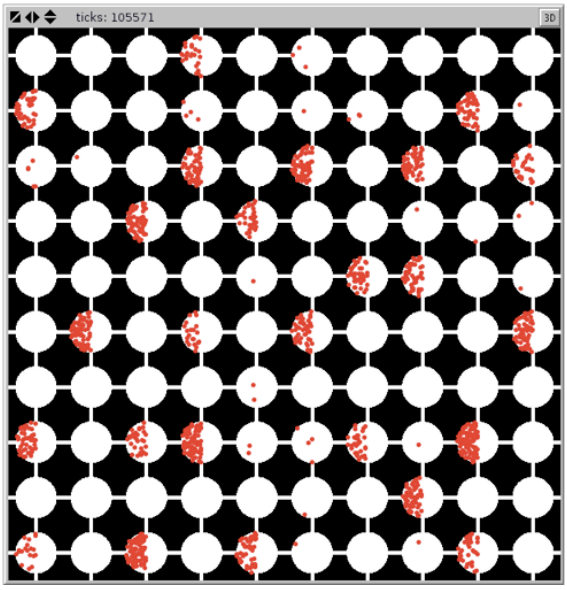
5.2.1 Experimental Results

This section shows the results obtained from experiments on tuple clustering run on a 100-tuple-space torus-topology network. The main peculiarity of a torus topology lies in the lack of a center in the network since even the nodes at the edge of the network are linked to the nodes on the opposite side. As a result of this topology, every node features the same connectivity degree so that a torus can also be thought of as a space wrapped around itself. One of the benefits of adopting such a topology lies in the chance of testing the effectiveness of the proposed solution on a network whose topology scarcely influences the way tuples aggregate. As a consequence, the way tuple organise in the network is mainly influenced by the enactment of the coordination rules and only barely by factors related to chosen topology.

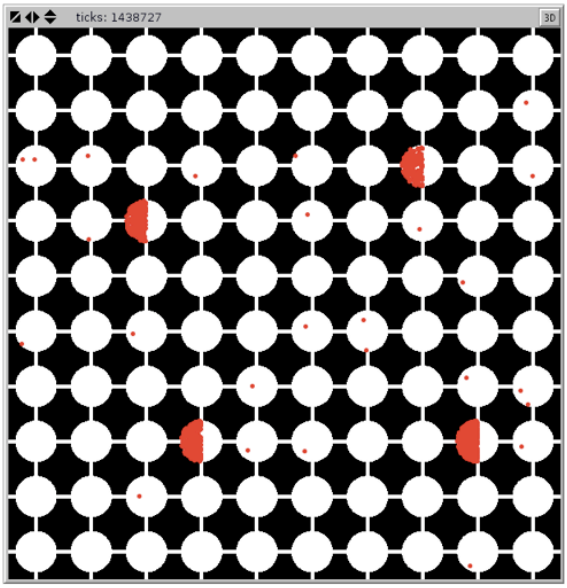
The simulation framework adopted for the experiments was NetLogo[1]. In the experiments we considered an initial configuration featuring 2500 tuples uniformly distributed in the network and $P_{mobile} = 0.2$, whose influence on the aggregation process is mainly related to the speed in cluster formation. The result of a simulation run by considering these initial conditions is shown in Figure 5.1. In particular, Figure 5.1 (a) reports the initial state featuring tuples uniformly distributed in the network, while Figure 5.1 (b) and Figure 5.1 (c) depict intermediate states of the simulation. Finally, Figure 5.1 (d) reports the final state of the simulation. As clearly visible throughout these figures, tuples tend to increasingly aggregate until coming to a complete clustering where all the tuples in the network get stored in the same tuple space: such a tuple space is not chosen a priori and in a global way but by emergence.



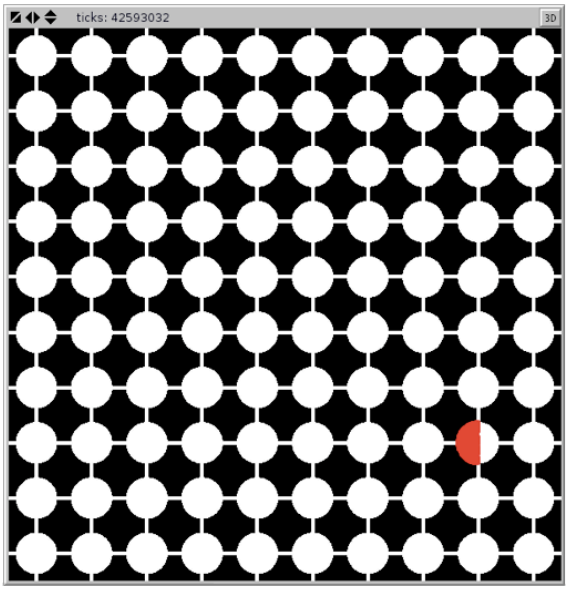
(a) Initial state: tuples are uniformly distributed.



(b) Intermediate state 1: tuples start aggregating.

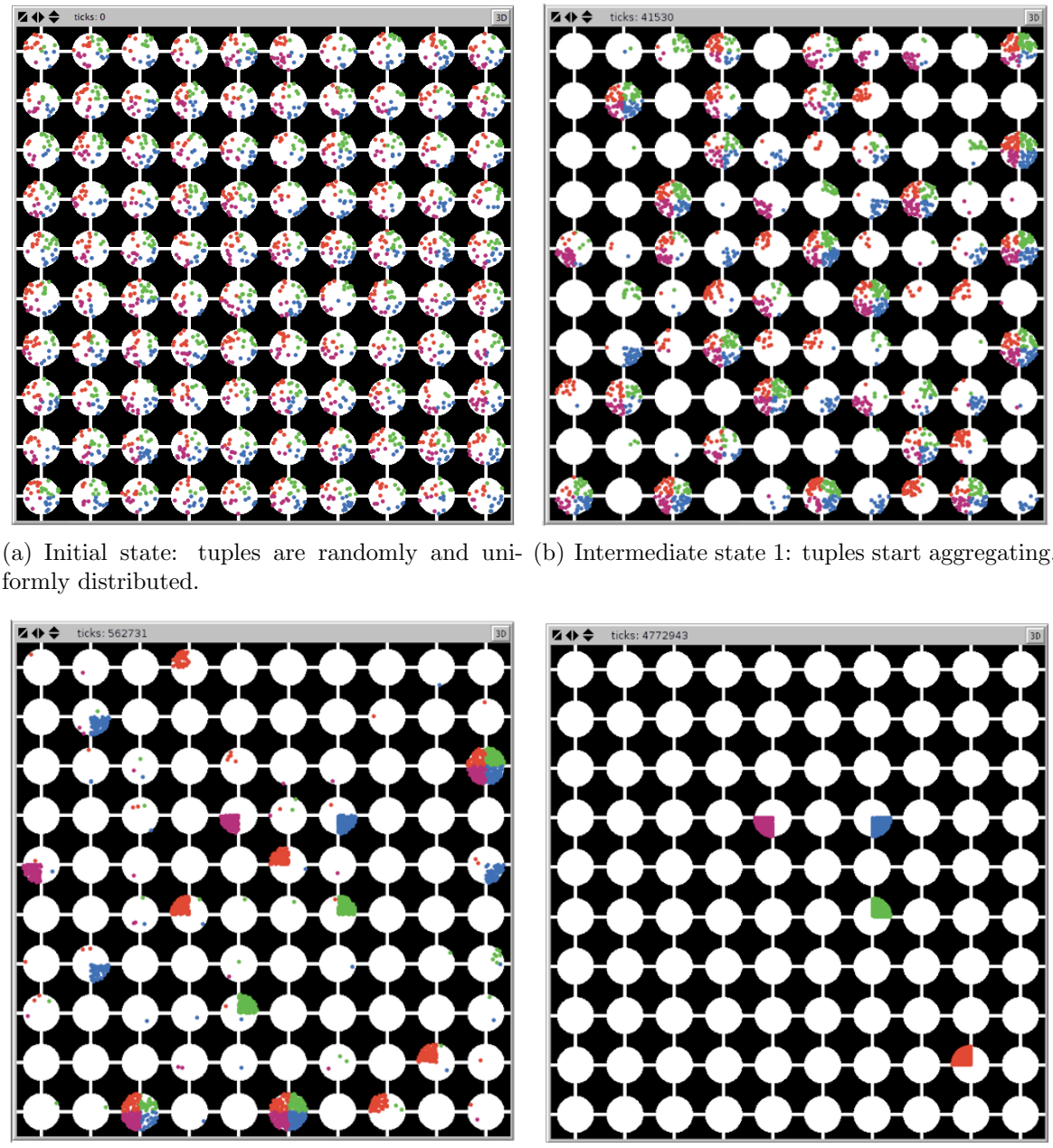


(c) Intermediate state 2: aggregation gets stronger.



(d) Final state: tuples are completely clustered in one tuple space.

Figure 5.1: Result of the execution of the tuple clustering algorithm on a 100-tuple-space torus.



(a) Initial state: tuples are randomly and uniformly distributed.

(b) Intermediate state 1: tuples start aggregating.

(c) Intermediate state 2: aggregation gets stronger and tuples start getting segregated per kind.

(d) Final state: tuples are completely aggregated and segregated: one cluster per tuple kind.

Figure 5.2: Result of the execution of the tuple sorting algorithm on a 100-tuple-space torus.

5.3 Self-Organising Tuple Sorting

In this section, a solution to the problem of tuple sorting in distributed networks is proposed by generalising the results obtained on tuple clustering. Now, instead of tuples

[1] `http://ccl.northwestern.edu/netlogo/`

of the same kind, we consider tuples of N different kinds, that is, tuples whose carried information belongs to N disjoint classes. The kind of a tuple is represented by the template of the tuple itself. The main inspiration for tuple sorting comes from the models of larval sorting proposed by Deneubourg et al. [BDT99, DGF+91] and Lumer and Faieta [BDT99, LF94]. The scenario of tuple sorting can be regarded as a generalisation of tuple clustering: indeed, here the goal is not only to aggregate tuples, but also provide a segregation mechanism to separate tuples belonging to different kinds so as to achieve tuple sorting. Accordingly, as a desired emergent behaviour, we aim at observing the formation in the network of clusters sorted per tuple kind.

As described in Section 5.2, access to tuple-space state is granted by exploiting the traditional LINDA primitives. However, since now we need to deal with different kinds of tuples, it is key to assume that `rd` operations work probabilistically in a uniform way, that is, each tuple matching a specified template is as likely to be retrieved as other tuples matching the same template—such a kind of operations is hereafter referred to as uniform read operations `urd`. Correspondingly, a uniform read operation that matches any tuple among N kinds is such that the higher the concentration of tuples of kind K, the higher the probability of reading a tuple of that kind. Therefore, this operation can be used to probabilistically observe which kind of tuples a space is mostly aggregating. The `urd` primitive makes tuple sorting truly self-organising and fully adaptive since it provides a mechanism for locally and probabilistically perceiving tuple-kind concentrations—as typically occurs in larval sorting.

The new rules enacted by each tuple space are reported in the following: such rules represent just a slight generalisation of the rules adopted for tuple clustering.

1. a `urd` operation is performed on local space L, yielding a tuple t_k of kind k
2. another `urd` operation is performed on L yielding a tuple t_m of a kind $m \neq k$ (if existing)
3. **if** t_m is *still*
   - let $mob(m, L) = 1 - conc_m(L)/conc(L)$
   - $P_{mobile} = \begin{cases} mob(m, L) & \text{if } mob(m, L) > 0 \\ C \text{ (constant)} & \text{otherwise} \end{cases}$
   - t_m gets *moving* by probability P_{mobile}
     - **if** t_m gets *moving*: $conc_m(t_m) = conc_m(L)$
4. **else** (t_m is *moving*)
   - a random number r between 0 and 1 is drawn
   - **if** $r < e^{-(conc(t_m)/conc_m(L))}$

  - t_m gets *still*
- **else**
  - a neighbouring tuple space R is chosen
  - t_m is moved to R

First, a tuple t_k of kind k is uniformly chosen in a probabilistic way by `urd`. Due to `urd` semantics, local space L is likely to be mostly aggregating tuples of kind k. Accordingly, since one of the goal is to segregate tuples of different kind, another `urd` is performed by considering only tuples whose kind is not K. The so-read tuple t_m is of kind $m \neq k$, whose tuple concentration is likely to be lower than concentration of kind k.

Then, if t_m is *still*, it gets *moving* by a probability P_{mobile} whose value is calculated as $mob(m, L) = 1 - conc_m(L)/conc(L)$, where $conc_m(L)$ represents the concentration of tuples of kind m in local space L while $conc(L)$ is the total concentration of tuples in L. Correspondingly, ratio $conc_m(L)/conc(L)$ denotes the percentage of tuples of kind m in L. As a consequence, the higher the value of such a ratio, the lower the probability P_{mobile} of a tuple to get *moving*. This allows tuple of kinds not strongly aggregating on local space L to be moved away towards tuple spaces featuring higher concentration of tuples of that kind. On the other hand, if L does not store tuples of kinds other than k, the kind of the chosen tuple remains K so as to lead to $mob(m, L) = 0$. If that is the case, P_{mobile} is assigned a constant value C as for tuple clustering.

In case t_m is *moving*, the applied rules are the same as for tuple clustering. Here, the only difference lies in the exponent of e, which is calculated by taking into account only the concentration of tuples of kind m. In particular, $conc_m(t_m)$ represents the concentration of tuples of kind m perceived by t_m during the last movement.

5.3.1 Experimental Results

This section shows the results obtained from experiments on tuple sorting run on a 100-tuple-space torus-topology network by adopting Netlogo as in the experiments related to tuple clustering. We considered tuple of 4 different kinds, an initial configuration of 2500 tuples per kind uniformly distributed in the network, and $C = 0.2$—C is the value assigned to P_{mobile} when local space L aggregates only tuples of the same kind. The result of a simulation run by considering these initial conditions is shown in Figure 5.2. In particular, Figure 5.2 (a) reports the initial state featuring tuples uniformly distributed in the network, while Figure 5.2 (b) and Figure 5.2 (c) depict intermediate states of the simulation. Finally, Figure 5.2 (d) reports the final state of the simulation. As clearly visible throughout these figures, tuples tend to be increasingly aggregated and segregated until coming to complete sorting, which corresponds to the state where all the tuples of the same kind get stored in the same tuple space: as for tuple clustering, such a tuple space is not chosen a priori and in a global way but by emergence. It is remarkable that by slightly extending the rules adopted for tuple clustering, it is possible to achieve a final

tuple organisation featuring complete aggregation and segregation: indeed, the resulting tuple configuration is composed of one cluster per tuple kind.

5.4 Concluding Remarks

The self-organising approaches to tuple organisation presented here have some correlation with amorphous computing [AAC+00], in particular with pattern and shape formation where, starting from a high-level description of the desired pattern, agents forming a network resembling a lattice are assigned a program by exploiting biologically inspired primitives. Such agents, which have only a partial knowledge of system's state, locally enact the assigned program so as to make desired patterns emerge at the global level. While in pattern and shape formation a node of the network is represented by an agent, in our work a node is a tuple space. Amorphous computing has also been applied to sensor networks so as to define the global behaviour desired on a continuos space and the corresponding implementation on a sensor network that approximates the continuos space.

Inspired by previous works such as [MZ04, MT04, CGV07, DNLM05], the goal here was to show how models and techniques coming from the self-organisation community can be leveraged in the context of coordination models and languages to devise proper infrastructures to manage interaction in dynamic and unpredictable environments.

The presented solutions to tuple clustering and sorting were not developed considering performance as a main objective since the main goal was to show that cellular-automata-like approaches – where the decision on which tuple to be moved to a neighbour is based on a single championing operation – can actually result in the emergence of the desired clustering and sorting properties.

6

Self-Organising Coordination by TuCSoN and ReSpecT

This chapter first clarifies the details of the tuple-centre coordination model provided by TuCSoN and ReSpecT. Then, a suitable framework for enacting self-organising coordination services is shown that is based on the aforementioned technologies.

6.1 The Coordination Model of TuCSoN

The TuCSoN coordination infrastructure provides Java agents with coordination media spread over the network, called *tuple centres*. A tuple centre is a tuple space abstraction augmented with the possibility of programming coordination rules, which are fired as a response to interaction events, and which can transform the set of tuples as well as execute coordination primitives on other tuple centres in the neighbourhood. ReSpecT (Reaction Specification Tuples) is the logic-based language by which such coordination rules can be programmed [OD01].

Agents act on a ReSpecT tuple centre according to the original Linda [Gel85] tuple-space model: tuples can be inserted by primitive `out`, retrieved by primitives `in` and `inp`, and read by primitives `rd` and `rdp`. Readings and retrievals can be executed by specifying a *template*, which serves as an identifier of a set of tuples according to a *tuple-matching* mechanism. Some primitives are suspensive (`in` and `rd`), so that the requesting agent waits until a matching tuple is found, while others are non-suspensive (`inp`, `rdp`), so that the agent is immediately provided with a result that may be either a matching tuple or a failure. However, the matching mechanism is non-deterministic: if more than one tuple in a tuple centre matches the specified template, the tuple to be returned is non-deterministically chosen among the matching ones. ReSpecT tuple centres adopt a logic-based model for tuples: tuples and templates are first-order terms and the matching mechanism is based on unification.

While the behaviour of a standard tuple space in response to interaction events is fixed (the effect of coordination primitives follows LINDA model), the behaviour of a tuple centre is defined by a set of ReSpecT specification tuples called *reactions*, which determine how

a tuple centre reacts to incoming/outgoing events.

The version of ReSpecT we rely on adopts the A&A meta-model [ORV08]—the interested reader can refer to [COV08, Omi07] for a complete description of syntax and semantics. In ReSpecT, a reaction is specified by a *specification tuple* `reaction(E,G,R)`, which associates (modulo unification) a reaction body `R` to an event `E` if the guard `G` is satisfied: `E` expresses the agent interaction to be intercepted, `G` the condition that the event must satisfy (concerning status, source, target, and time of the event), and `R` is a list of goals. A goal specifies a basic computation, which could be *(i)* an insertion/retrieval of tuples in the local tuple space (which might itself fire new reactions leading to a Turing-equivalent chain of reaction executions); *(ii)* an insertion/retrieval of tuples in a remote tuple space; and *(iii)* a Prolog algorithmic computation.

As an example of a simple ReSpecT program, we consider the following rules, which change the behaviour of `in` primitive so that all tuples matching the template are removed:

```
reaction( in(X), (response, from-agent), (     %(1)
    out(remove(X))
)).
reaction( out(remove(X)), endo, (              %(2)
    in(X), in(remove(X)), out(remove(X))
)).
reaction( out(remove(X)), endo, (              %(3)
    no(X), in(remove(X))
)).
```

Whenever an agent asks to remove a tuple of a generic template `X`, the first reaction is triggered and executed: the guard predicate `response` is true when `in(X)` gets served—namely a tuple is found and about to be replied to the agent. The execution of reaction `(1)` leads to the insertion of a tuple `remove(X)` in the tuple space, where `X` identifies the template specified by the requesting agent. The insertion of `remove(X)` triggers reactions `(2)` and `(3)`—both reactions feature guard `endo`, which is true when the triggering event is generated by the tuple centre itself. Reaction `(2)` is successfully executed as long as there are tuples matching `X` (`in(X)` succeeds): in that case, other than removing `X`, the reaction is triggered again, recursively. When all the tuples matching `X` are removed, the execution of `(2)` fails while reaction `(3)` succeeds—reaction goal `no(X)` succeeds only if no tuple matching `X` can be found in the tuple space. Since each reaction is atomically executed, the resulting behaviour is such that all tuples matching `X` are removed.

6.2 Self-Organisation in Tuple Centres

TuCSoN can be shown to fit the properties of self-organising coordination as outlined in Chapter 2:

Topology — Assuming the network is organised in a topologically structured distributed system, TuCSoN allows one or more tuple centres to be created locally to any specific node on the network. Java agents, too, are supposed to be localised in a node of the network.

Locality — In TuCSoN, a coordination primitive can be executed over a tuple centre provided its identifier (name and host address) is known—and this happens both for agent-medium and medium-medium interactions. As such, to structure a multiagent system featuring self-organising coordination, we simply need agents and tuple centres to be aware of the list of tuple centre identifiers in the neighbourhood. For tuple centres, e.g., this simply means a tuple `neighbour(tc)` occurs in the space if tuple centre `tc` is in the neighbourhood.

On-line character and Time — ReSpecT supports timed reactions, namely reactions whose event `E` is of the kind `time(T)`. When the tuple centre time (expressed as Java milliseconds) reaches `T`, the corresponding reaction is fired. Moreover, a reaction goal can be of the kind `out_s(reaction(time(T),G,R))`, which inserts tuple `reaction(time(T),G,R)` in the space, thus triggering a new reaction. As a simple example, the following reactions are used to update the integer argument `N` of a tuple `tick(N)` in the space each second, starting from time `t`, so as to define a clock where the value of `N` represents the elapsed seconds:

```
reaction( time(t), true, (
  in(tick(N)), N1 is N+1, out(tick(N1)) )).
reaction( out(tick(X)), endo, (
  current_time(T), NewT is T+1000,
  out_s(reaction( time(NewT), true, in(tick(N)), N1 is N+1, out(tick(N1)))) )).
```

This mechanism can hence be used to realise either an on-line service that keeps transforming tuples as time passes, or time-dependent coordination primitives.

Probability — Probability of coordination rules is supported in TuCSoN in two ways. On the one hand, it is possible to draw random numbers in Prolog and use them to drive the reaction firing process, that is, tuple transformation by reactions can be intrinsically probabilistic. For instance, the following reactions insert either tuple `head` or `tail` in the space (with 50% probability):

```
reaction( ..., ..., (
  rand_float(X), out(rnd(X)) )).
reaction( out(rnd(X)), endo, (
  in(rnd(X)),
  ( X > 0.5, out(head)
    ;                 % ';' as logic disjunction
    out(tail) ) )).
```

On the other hand, tuple retrieval can be moved from the non-deterministic version to a probabilistic version by using a variant of the usual primitives, i.e. `urd` instead of `rd`. As described in [CGV07] and Section 3.3, this is called *uniform read* operation (and analogously for `uin`, `uinp`, and `urdp`): its semantics is such that among all tuples that match the given template, one is chosen equiprobabilistically. As an example, if the space has 100 tuples `t(red)`, and 50 tuples `t(blue)`, then operation `rdp(t(X))` yields `t(red)` with probability 66%. This can be extremely useful to exploit tuple retrieval as a sampling mechanism to partially observe tuple centre state.

6.3 Adaptive Tuple Distribution

Here two complementary application scenarios are presented, the former focussing on self-organisation through interactions *between* coordination media (presented in this section), the latter through interactions (of tuples) *inside* a coordination medium (presented in next section). The combination of these two approaches has the potential for leading to a full-featured self-organisation methodology for coordination—though deepening such implications is left for future investigations. However, both of them are interesting *per se* and also show the usefulness of self-organisation in coordination models and the suitability of TuCSoN as a supporting platform.

We start considering the self-organising approaches in data-centered network systems, which concern adaptive information distribution presented in Chapter 5. The problem is how and where information items – tuples in our case – are to be moved, copied, and transformed, so as to achieve certain patterns of distribution, like co-fields [MZ04], clustering, sorting, and the like [MMTZ06], which may be used to facilitate access to data and resources. Among the many cases that could be considered, we here focus on emergent tuple *clustering* in distributed networks as described in Section 5.2 and [CV08], which is considered as a simple and paradigmatic case.

6.3.1 Tuple Clustering

Drawing inspiration from natural phenomena like *corpse clustering* by ants [BDT99], we focus on the problem of *tuple clustering* in distributed tuple-space networks, which amounts at defining a strategy to spatially aggregate tuples of the same kind – i.e. tuples carrying information of the same class – in few and small clusters that form on the network in an emergent way (see Chapter 5 and [CV08])—ideally, one cluster made of one space. As such, each tuple space in the network is provided with coordination rules whose goal is to locally move tuples to neighbouring tuple spaces if this may increase clustering, ultimately leading to the emergence of global tuple clustering only by local observations and actions. As usual, access to tuple space state is granted by performing tuple readings (`rd` operations), retrievals (`in` operations), and insertions (`out` operations).

In order to exploit the metaphors coming from corpse clustering, tuples need to be enriched with some additional meta-data. A tuple can be either *still* or *moving*. A *still* tuple is one whose current position has been selected as a good candidate final position—it would represent an item on the ground according to the ant-based metaphor. On the other hand, a *moving* tuple is one which has not yet found a good candidate position—dually, it would represent an item picked up by an ant. A moving tuple additional carries a numerical value representing the concentration (i.e. number) of tuples in the space where the tuple started moving—it represents the memory the ant has of the place where the item was picked up. This is fundamental to facilitate the storage of a tuple in tuple spaces whose tuple concentration is higher than the one in the spaces previously visited by the tuple.

Correspondingly, the coordination rules applied by each tuple space are as follows, which recall those alredy presented in Chapter 5. Let t be a tuple observed by a `urd` operation; then

- if t is *still*, it moves to state *moving* (storing the local value of concentration) by probability P_{mobile}
- if t is *moving*, it moves to state *still* by probability P_{still}
- if t is *moving*, it moves to a neighbouring tuple space R randomly chosen by probability $1 - P_{still}$

P_{mobile} is a fixed value (0.2 in the experiments shown in Section 5.2.1) though this value is likely to depend on the size of the network and the number of tuples to be clustered. $P_{still} = e^{-(conc(t)/localconc)}$, where $conc(t)/localconc$ ($\in [0, +\infty]$) is the relative concentration of tuples perceived by t during the last movement, with respect to the local concentration. In particular, a high value of relative concentration leads to a low probability of storing t (the spaces previously visited by t had higher concentration of tuples); a low value of relative concentration increases the probability of storing t (the local space is a better candidate for storing w.r.t. previously visited ones). An example of a system evolution following this algorithm is shown in Figure 5.1: emergently, one cluster is eventually selected that stores all the tuples.

6.3.2 TuCSoN Implementation

We suppose that each node of the network hosts one tuple centre, in which the following tuples are stored: *(i)* `tuple(still,T)` represents the still tuple `T` to be clustered; *(ii)* `tuple(moving(C),T)` represents the moving tuple `T` to be clustered, where `C` is the concentration value stored in the tuple; *(iii)* `count(C)` means the space has currently `C` tuples to be clustered; *(iv)* `movingprob(P)` means that `P` is the moving probability; and finally *(v)* `neighbour(S)` means that `S` is the identifier of a space in the neighbourhood. We suppose that tuples are inserted in spaces in the still state, and that the count

value is automatically updated by the tuple centre as long as they are inserted, removed, and replaced. In order to implement clustering as a self-organised coordination problem in TuCSoN, it is then sufficient to inject in each tuple centre the following ReSpecT specification:

```
reaction( time(T), true, out(step) ).                %(1)
reaction( out(step), endo, (                         %(2)
    in(step),
    uin(tuple(K,T)),
    rand_float(Tao),
    out(go(K,T,Tao)),
    current_time(T), T2 is T+1000,
    out_s(reaction( time(T2), true, (out(step))))
)).
reaction( out(go(still,T,Tao)), endo, (              %(3)
    in(go(still,T,Tao)),
    rd(movingprob(P)),
    ( Tao < P, rd(count(C)), out(tuple(moving(C),T))
      ;
      out(tuple(still,T))
    )
)).
reaction( out(go(moving(C),T,Tao)), endo, (          %(4)
    in(go(moving(C),T,Tao)),
    rd(count(LC)),
    Level is exp(-C/LC),
    ( Tao < Level, out(tuple(still,T))
      ;
      urd(neighbour(TCR)),
      TCR ? out(tuple(moving(C),T))
    )
)).
```

Reaction `(1)` is used to fire the movement process in reaction to time, as usual. Reaction `(2)` randomly takes a tuple `T` to be possibly clustered and selects a random number `Tao` (to be used next); then, it inserts tuple `go` to proceed with the algorithm, and fires reaction `(1)` again after `1000` time units—this parameter is to be changed depending on the intended rate of clustering. Still tuples are managed by reaction `(3)`, which inserts the tuple back in the space, choosing the status according to moving probability. On the other hand, moving tuples are managed by reaction `(4)`, which (probabilistically) either inserts a still tuple or moves the tuple in a neighbouring space.

Although simple, this example paradigmatically shows how locality, probability, and time altogether play a crucial role in constructing a self-organising coordination service.

6.4 Chemical-like Coordination

When seeing self-organising coordination as enacted by rules that keep transforming and moving tuples in a distributed setting, it is quite natural to find a connection with chemistry and biology, where chemical laws similarly transform molecules floating in a chemical soup. Accordingly, we envision the idea of a "biological" coordination model, where chemical- and biological-oriented laws are used to coordinate agent behaviour—re-using concepts like oscillatory chemical systems, catalysts, membranes, DNA-related mechanisms and the like as coordination metaphors. Among the many sources, our work has been also inspired by previous approaches adopting chemical abstractions for coordination like the Chemical Abstract Machine [BB92] or Gamma [BLM96]. However, besides having no explicit interest in self-organisation, those early approaches never brought the chemical metaphor to its full realisation, while we here address fundamental issues like concentration/number of molecules, time of reactions, etc. so as to exactly reproduce chemical processes.

6.4.1 Chemical Reactions for Coordination

Chemical systems are the source scenario where the phenomenon of self-organisation has been widely studied [PS97]. The large-scale character of chemical systems, along with the locality of their inner interactions and the intrinsic chaotic behaviour they manifest, make us observe the emergence of global properties of reactants that cannot be straightforwardly ascribed to the shape of chemical laws, such as final concentrations, decay rates, periods of oscillatory behaviours, and so on. Then, by adding a topological structure to standard chemical systems – reactants diffuse to certain parts of space, molecules bind forming complex structures, membranes tend to form separate compartments – we move into the complexity of biology, where most self-organisation patterns of nature actually take place—and this can be achieved by combining with inter-space techniques like those discussed in previous section. It does not come as a surprise then, that many research activities were developed to build artificial computer-based systems exhibiting organised complexity and properties of chemical and biological systems.

As in many previous works, we start from the consideration that distributed systems might well be designed as chemical systems: mobile data-items spread in the network can be seen as floating molecules, and processes manipulate them so as to enact chemical laws that make data-items aggregate and transform into other data. Moving to the coordination setting, we foster the idea of a coordinated system as a system where the interaction space hosts a chemical dynamics and evolution: coordination-related data (shared knowledge, reified interactions, and the like) flow in it, and coordination rules manage such a data like chemical laws do. On the one hand, agents can get simply coordinated through this chemical space, by inserting new data and retrieving chemically transformed one. On the other hand, agents can truly influence the chemical system

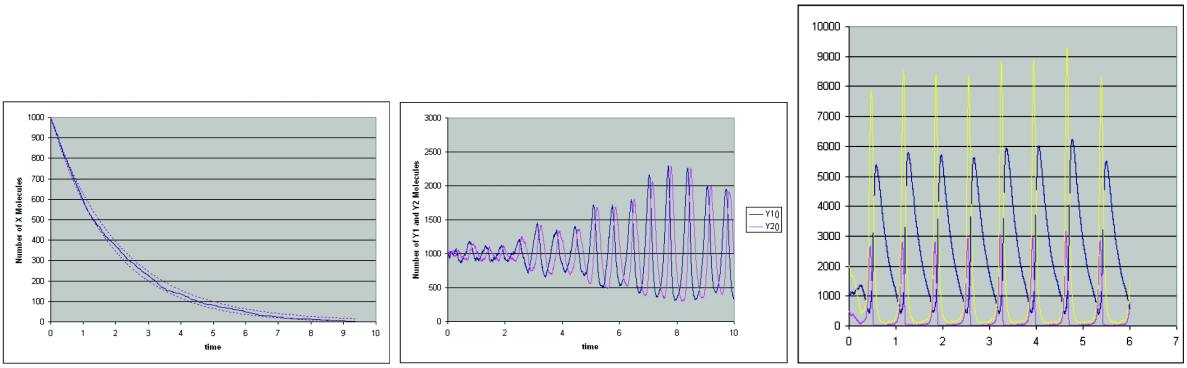


Figure 6.1: System evolutions for chemical reactions: (a) Decay, (b) Lotka reactions, (c) Oregonator.

and behaviour, by inserting e.g. data-items acting like catalysts for certain reactions. Moreover, the number of tuples of the same kind – namely the concentration of a chemical substance – can be used by an agent as a dynamic characterisation of its "activity" level: the higher such a concentration, the more likely the agent will coordinate with others through the tuple centre. Although it is out of the scope of this work to fully analyse the impact of this metaphor on coordination, we aim at providing some initial ideas and implementations, paving the way towards a deeper understanding and exploitation.

A computational model for simulating chemical reactions is given by Gillespie in [Gil77], which basically provides a characterisation of discrete chemistry – number of molecules are considered instead of concentrations – as a continuous-time stochastic transition system à la Markov. Consider a chemical rule $\mathtt{A} + \mathtt{B} \xrightarrow{r} \mathtt{C} + \mathtt{D}$, meaning that a molecule `A` joins molecule `B`, forming molecules `C` and `D`. Rate r is called *reaction constant*, and expresses the likelyhood of the reaction between two pairs of molecules `A` and `B`. As shown in [Gil77], the global rate of the reaction is proportional to r and to the concentration of each reactant in the left-hand side. As a result, given n chemical laws, in a certain state we can determine the n global rates $r_1, \ldots, r_n$, one per available chemical law, with R being their sum. According to Markov's model, an exact simulation of the system evolution is obtained by a computation step whose duration is described by negative exponential distribution due to Markov property, and where one law is chosen probabilistically and applied (law i is chosen with probability r_i/R). This approach is in fact the standard one for stochastic computational models used to simulate biological systems, as in the case of stochastic π-calculus [Pri95].

In [Gil77] some examples of natural and synthetic chemical systems are described, which could be of some interest in coordination—Figure 6.1 reports corresponding simulation traces, as computed in the context of [Phi06].

Decay — Rule $\mathtt{X} \xrightarrow{r} 0$ describes the decay of `X` (irreversible isomerisation), which can be used as a mechanism to fade pheromone-like data-items—see Figure 6.1 (a).

Lotka Reactions — These reactions are used to model a prey-predator system, in which a prey species `Y1` feeds on an inexhaustible food source `X` to reproduce, and a predator species `Y2` feeds on `Y1` to reproduce while dying of natural causes. Three rules can describe this system: *(i)* $\texttt{X}+\texttt{Y1} \xrightarrow{c1} 2\texttt{Y1}+\texttt{X}$, *(ii)* $\texttt{Y1}+\texttt{Y2} \xrightarrow{c1} 2\texttt{Y2}$, and *(iii)* $\texttt{Y2} \xrightarrow{c3} 0$. With certain rates, populations of `Y1` and `Y2` develop an oscillatory behaviour: `Y2` population appearently self-adapts to the increase/decrease of `Y1` population—see Figure 6.1 (b). In a coordination setting, these kinds of rule can be used to keep requests `Y1` of accessing a resource `X` limited, thanks to a sort of automatically generated "controller item" `Y2`.

Oregonator — These reactions describe an artificial chemical system that enacts a stable oscillatory behaviour: *(i)* $\texttt{X1} + \texttt{Y2} \xrightarrow{c1} 2\texttt{X1} + \texttt{Y1}$, *(ii)* $\texttt{X} + \texttt{Y1} \xrightarrow{c2} 0$, *(iii)* $\texttt{X2} + \texttt{Y1} \xrightarrow{c3} \texttt{X2}+2\texttt{Y1}+\texttt{Y3}$, *(iv)* $2\texttt{Y1} \xrightarrow{c4} 0$, and *(v)* $\texttt{X3}+\texttt{Y3} \xrightarrow{c5} \texttt{X3}+\texttt{Y2}$. Under certain rates, reaction *(v)* indeed behaves like a time tick, fired at a very stable frequency—see high peaks in Figure 6.1 (c). This behaviour could be used in coordination to simulate time, e.g. to regulate a time-based scheduler for accessing resources—whose features can be altered by properly inserting/removing elements of species `X1`, `X2`, and so on.

6.4.2 TuCSoN Implementation

A TuCSoN tuple centre can be specialised to act as a chemical system where tuples model reactant molecules: they combine and transform over time as in true chemical systems, and agents perceive such transformations through usual tuple retrieval. Let `m` be a molecule kind, we assume the tuple centre holds one tuple of the kind `reactant(m,N)`, where `N` is the number of molecules. This is just a compact way of expressing the chemical state instead of using one tuple per molecule: agents keep inserting and removing molecules one a time, and a ReSpecT specification can be used to turn such operations into proper increment/decrease of molecule count. Moreover, we assume that chemical laws are expressed as tuples of the kind `law(InputList,Rate,OutputList)`. For example, Lotka reactions leading to the trace in Figure 6.1(b) would be expressed by tuples

```
law([x,y1],10,[y1,y1,x]).
law([y1,y2],0.01,[y2,y2]).
law([y2],10,[]).
reactant(x,1000).
reactant(y1,1000).
reactant(y2,1000).
```

and similarly for any other chemical system as e.g. shown in previous subsection. Algorithmic computations required to obtain rates and probabilities are expressed as a Prolog theory installed in the tuple centre. Predicate `chooseOne(-Law,-Dt)` encapsulates Gillespie's algorithm: assuming facts `reactant/2` and `law/3` properly populate the theory, it yields the law to be applied and the elapsed time—its implementation is not reported here for brevity.

By the following ReSpecT specification, the tuple centre can then be turned into a true engine for chemical systems:

```
reaction( time(Time), true, out(engine_trigger) ).  %(1)
reaction( out(engine_trigger), endo, (              %(2)
    in(engine_trigger),
    chooseOne(Law,Dt),
    current_time(Time),Time1 is Time + Dt,
    out_s(reaction(time(Time1),true,out(engine_trigger))),
    out(execution(Law))
)).
reaction( out(execution(law([C|T],_,O)), endo, (    %(3)
    in(execution(law([C|T],_,O)),
    in(reactant(C,N)), N2 is N-1,
    out(reactant(C,N2)),out(execution(law(T,_,O))
)).
reaction( out(execution(law([],_,[C|T])), endo, (   %(4)
    in(execution(law([],_,[C|T])),
    in(reactant(C,N)), N2 is N+1,
    out(reactant(C,N2)),out(execution(law([],_,T))
)).
reaction( out(execution(law([],_,[])), endo, (      %(5)
    in(execution(law([],_,[]))
)).
```

Reaction `(1)` is used to fire an execution step, as usual. Reaction `(2)` calls the Gillespie's engine obtaining the law to be executed and the elapsed time, re-triggering reaction `(1)`, and also firing reactions `(3,4,5)` aimed at applying the chosen chemical law. Reaction `(3)` drops one molecule per reactant; recursively, reaction `(4)` adds the molecules created by the chemical law, and finally reaction `(5)` concludes the reaction chain.

By this specification, the content of the tuple centre keeps evolving according to the chemical laws—in a sense, it indeed simulates the chemical system iteself. As described above, agents can perceive the current state of the system by executing primitives, or can affect the chemical behaviour by properly inserting/removing reactants.

7

Formal Verification of Self-Organising Coordination Systems

The global behaviour resulting from self-organising coordination systems can be regarded as an emergent property since it appears by a process emerging from local interactions among components. As clearly described throughout the thesis, the corresponding system dynamics is usually non-linear and complex so that the adoption of simulation and verification techniques in the early design stage becomes essential to carry out an effective design. As far as formal verification is concerned, this chapter discusses a hybrid approach relying on stochastic simulation and probabilistic model checking, showing an example of emergent-property check on the collective sort problem presented in Chapter 3. To this end, the PRISM probabilistic model checker is adopted as a concrete tool for analysing emergent properties on collective sort. A comprehensive discussion of the corresponding results is provided.

7.1 Introduction

As clearly remarked so far, the adoption of self-organisation approaches for system coordination leads to a shift in designer's focus towards a bottom-up-like design where the key point becomes to define the behaviour of system's single components rather than the overall behaviour of the system as a whole. As a consequence, the desired global behaviour of the system is achieved by *emergence* [CDF+01, Par97, BDT99].

Given the non-linear and complex dynamics of self-organising systems, the early design stage becomes essential for a successful design of such systems, e.g. as proposed in [GVO08], where a methodology for designing self-organising systems is presented. Similarly, even though not relying on a specific methodology, here it is shown how stochastic simulation and probabilistic model checking can be applied to simulation and verification of self-organising coordination systems in the early design stage. While simulation is useful to provide informal evidence of expected behaviour, verification becomes key for formal validation by automatically verifying properties on system models. In particular, since in self-organising systems desired patterns and behaviours are achieved by emergence, veri-

fying properties on a model means to automatically check whether such emergent patterns and/or behaviours may be guaranteed. Correspondingly, the chapter deals with verifying the probability of properties of self-organising coordination systems by adopting a hybrid approach, which combines stochastic simulation and probabilistic model checking so as to provide an approximated technique for self-organising-system verification.

As a reference example of a self-organising-coordination approach, the basic solution to the *collective sort* problem described in Chapter 3 (Section 3.3) is taken in order to show the suitability of stochastic simulation and probabilistic model checking for supporting the design stage. To this end, as a concrete tool, we relied on PRISM [PRI07], a probabilistic symbolic model checker developed by University of Oxford. PRISM was applied on collective sort in order to analyse the emergence of complete sorting, taken as an example of emergent property. In particular, complete sorting was analysed on several instances and initial configurations of the system.

7.2 Self-Organising System Design by Stochastic Simulation and Probabilistic Model Checking

7.2.1 Stochastic Simulation

Computer-based simulation has been widely adopted in the study of hardware systems over the past years. In addition, computer-based simulation is also becoming increasingly adopted as a suitable approach for analysing complex systems and supporting software design. Correspondingly, this chapter focusses on simulation as an important support for the early phases of software design, especially when self-organisation is adopted.

As far as self-organisation is concerned, the adoption of deterministic simulation approaches is unfeasible due to the intrinsic stochastic nature of self-organising systems. Accordingly, stochasticity becomes an important ingredient of any simulation tool for self-organising systems as it allows unpredictability to be brought to the model level, so that aleatoric system evolution can be explicitly modelled at the model level. Examples of stochastic simulation languages and engines are reported in [Phi06, Gil77, Pri95, CGV07, DPHW05]. In most of these, the evolution of a self-organising system is modelled in terms of Continuous-Time Markov Chains (CTMC) or Discrete-Time Markov Chains (DTMC). Both can be regarded as a stochastic transition system but, while in the latter, transitions are labelled with probabilities, the former features transitions labelled by rates, which represent the average frequency at which a transition occurs.

Once a model of the system has been defined, simulation is adopted as a tool for performing qualitative analysis and obtaining some clues about the dynamics of the system. In order to actually perform simulation, it is first necessary to provide a set of suitable working parameters for the model to be simulated, decide the initial states to be considered as test instances, and choose simulation parameters, e.g. the maximum number of

steps to be considered in the simulation run [GVO08].

7.2.2 Probabilistic Model-Checking

Model checking is with no doubt one of the most used techniques for automatic verification of properties on finite-state-system models [CGP00]. Models are usually expressed by formal languages in terms of transition systems and the resulting specification given to a model checker, which translates such a specification in a suitable internal representation, usually a Binary Decision Diagram (BDD) [CGP00]. Once the model has been loaded into the model checker, properties to be verified need to be specified: this is typically done by adopting temporal logics [CGP00]. The model checker can then perform automatic verification of the declared properties by a full exploration of every possible computational path resulting from exhaustive model execution. The resulting output is a boolean answer indicating whether the declared property holds for the model.

Since the focus here is on self-organising systems, the probabilistic extension of traditional model checking is considered that can be exploited to deal not only with probabilistic but also with stochastic aspects [KNP04, KNP07]. The main difference between probabilistic and traditional model checking lies in the logics adopted for property specification. In fact, probabilistic model checking exploits a linear temporal logic extended with operators for specifying probabilities. As a consequence, the output of a probabilistic model checker can be either a probability or a boolean value. For instance, the answer to a property like *"will the system reach state S with a probability greater than 80%?"* is a boolean value, while the answer to a property like *"Which is the probability for the system to reach state S?"* is a probability value.

A major drawback of model checking is known as *state-space-explosion problem*, which is mainly due to a quick increase of state space as system size grows. Even though some techniques have recently been developed to mitigate this problem, state-space-explosion remains the main problem preventing a wider applicability of model checking [Del02]. As a solution to this problem, some approaches have been developed that rely on approximated model checking. In particular, as regards probabilistic model checking, approximated techniques based on Monte Carlo simulation have been proposed in [HLMP04]. The hybrid approach discussed in the next section relies on such approximated techniques.

7.2.3 The Hybrid Approach

In this section we propose the adoption of a hybrid approach by exploiting stochastic simulation, probabilistic model checking, and approximated verification techniques so as to define a framework for supporting the design of self-organising approaches to coordination.

As described in [PRI07], approximated model checking relies on sampling. In other words, given a CTMC/DTMC model of the system and a property to be verified, a large number of simulation runs are executed. For each run, the result is evaluated with

respect to the property to be verified. Then, once the runs are completed, the final result is provided as the average of all the obtained values. This makes it possible to perform approximated verification even on models of large-scale systems, since model checkers usually do not need to translate the CTMC/DTMC model into an internal representation (like BDD), which is typically a memory-consuming process.

Let M be a CTMC/DTMC model of a self-organising system and p a property to be verified on the model, suppose that we are interested in obtaining the probability for p to be satisfied on M. In order to deliver meaningful results from the standpoint of quantitative analysis, it is fundamental to carefully choose *confidence* (δ) and *approximation* (ϵ) values. Let $\mu(x)$ be the exact probability value resulting from the execution of M on a probabilistic model checker, $M(x, \epsilon, \delta)$ be the probability value resulting from the execution of M with approximation ϵ, confidence δ, and initial state x, the equation

$$Prob[|M(x, \epsilon, \delta) - \mu(x)| \leq \epsilon] \geq 1 - \delta$$

is satisfied if the number N of runs performed on M with input x is at least equal to $4log(\frac{2}{\delta})/\epsilon^2$. Put it simply, the probability for $M(x, \epsilon, \delta)$ to be different from $\mu(x)$ of a factor less than or equal to ϵ is greater than $1 - \delta$ if $N \geq 4log(\frac{2}{\delta})/\epsilon^2$. As a consequence, the higher ϵ, the lower the number of runs required and the quality of the obtained result with regard to $\mu(x)$. Conversely, fixed ϵ, lower values of δ lead to a greater number of required simulations but at the same time guarantee a higher degree of confidence on the result. In other words, the lower the error and the greater the confidence required, the higher the number of runs to be performed.

Another important point is the choice of the length k of the path generated per simulation run. Indeed, even though not trivial, k should be chosen big enough in order to provide a result not influenced by the fact that some runs could not satisfy p due to an insufficient path length. To avoid this problem, simulation can be adopted as a means to deliver a value of k large enough.

In the next section, we show how stochastic simulation, probabilistic model checking, and approximated probabilistic model checking can be used for analysing emergent properties on the *collective sort problem*.

7.3 Collective Sort as a Case Study

7.3.1 Relevance and Complexity

The basic solution to the collective sort, described in Chapter 3 (Section 3.3), is here briefly reviewed so as to highlight some of the issues crucial to the verification phase.

Collective sort, as presented in Chapter 3, considers a fully connected set of N tuple spaces: accordingly, tuples are supposed to be classified at design time in N kinds. Complete sorting is achieved when each tuple space aggregates only tuples of the same kind.

Accordingly, complete sorting can be regarded as the emergent property we are interested to analyse on collective sort by adopting the hybrid approach previously outlined. The behaviour of collective sort is independent of the number of tuples, tuple spaces, initial tuple configuration, and small perturbations due to user agents randomly moving tuples. Furthermore, even though performance can be affected by the above aspects, the effectiveness of the algorithm remains unaffected. As shown in Section 3.3.3, a percentage of cases is characterised by a symmetry between the number of spaces and kinds that might cause the system evolution to be attracted by stable global states featuring positive values of entropy, where sorting is only locally achieved. A system configuration featuring a local minimum of entropy is $(100, 0, 0, 0), (0, 70, 0, 0), (0, 30, 0, 0), (0, 0, 100, 100)$—the content of each space is presented as a list of 4 values representing the concentration of tuples per kind. Even though a noise-based solution to this issue is discussed in Section 3.3.4, noise is not contemplated here since it would lead to a model whose complexity would make it difficult to keep the focus on the hybrid approach proposed for emergent-property analysis.

As probabilistic model checking is highly affected by the state-space-explosion problem, a preliminary analysis of collective sort complexity is important in order to set up an efficient process for analysing emergence of complete sorting. Let N be the size of a collective sort instance in terms of number of tuple spaces and number of tuple kinds, the resulting number of possible transitions for each iteration of the sorting agent in charge of managing a space is $(N-1)N^2$, so that the total number of permutations is $(N-1)N^3$. Indeed, chosen a sorting agent, the number of tuple spaces to be potentially chosen as target space is $N-1$. Then, after choosing the target space, the possible permutations as regards source and target kind are N^2. This applies for each sorting agent, so that the total number of permutations is $(N-1)N^3$. For instance, considering $N=3$, the total number of permutations is 54 and rapidly grows to 192 with $N=4$.

Furthermore, as regards complexity from the standpoint of total number of possible states, it is necessary to introduce K_i, representing the total concentration of tuples for tuple kind i. If we take into account only tuple kind i and consider every possible permutation of tuples of kind i around the N available spaces, the resulting number of possible states is $Tot_i = \binom{N+K_i-1}{K_i}$. Accordingly, the total number of states is $Tot = \prod_{i=1}^{n} Tot_i$. For instance, considering $N=3$ and $K_i=9$ for each i, the resulting space-state size is $166,000$ and becomes $753,000$ with $K_i=12$ for each i.

7.3.2 PRISM

To model and analyse collective sort, we decided to exploit PRISM, the Probabilistic Symbolic Model Checker developed by University of Oxford [PRI07]. PRISM comprises a set of tools and facilities ranging from a modelling language to a simulator and a probabilistic model checker.

PRISM models are organised in modules specified in a transition system fashion. The

tool makes it possible to model non-deterministic, probabilistic, as well as stochastic systems by using respectively Markov Decision Processes (MDP), Discrete Time Markov Chains (DTMC), and Continuous Time Markov Chains (CTMC) [KNP04, KNP07]. In other words, models are specified in terms of modules whose behaviour is expressed by transition rules and whose state is encoded into a set of variables. Transitions are associated a rate (in CTMC) or a probability (in DTMC), and can be labelled if synchronisation among modules is required. A transition follows the syntax:

```
 [] guard -> r : (update-variable);
```

where `guard` is a sequence of conditions that need to be satisfied to enable the transition, `r` represents either a probability (DTMC) or a rate (CTMC), and `update-variable` is a list of operations for updating (part of) the state of the system by modifying variables. As a trivial example, consider the following specification, which defines a stochastic system – modelled as a CTMC – switching between two states by two transitions featuring different rates:

```
ctmc
module swing
 state : boolean init false;
 [] state=false -> 10.0 : (state=true);
 [] state=true -> 100.0 : (state=false);
endmodule
```

PRISM also provides a simulator whose simulation engine makes it possible to perform either step-by-step or N-step simulations, where N can be specified by the user. Variable values are traced so as to facilitate model debugging. The most remarkable feature of PRISM concerns probabilistic model checking features. Probabilistic model checking can be performed by choosing one of three model checking engines—the choice is mainly driven by the size of the model to be verified as the engines differ from one another in terms of computational and memory costs. Properties are specified according to Probabilistic Computational Tree Logic (PCTL) for DTMC and Continuous Stochastic Logic (CSL) for CTMC. For instance, considering a DTMC-based model C featuring a variable s ranging from 0 to 10, a property like *"Which is the probability for s to reach value 10?"* is expressed by the PCTL formula:

```
P=? [true U s=10]
```

As far as collective sort is concerned, the main interest consists in analysing its behaviour and the resulting emergent patterns. As a consequence, we decided to abstract away the time dimension from the collective sort model and adopt DTMC.

7.3.3 Modelling Collective Sort by PRISM

Collective sort was modelled in PRISM as a DTMC by explicitly specifying all the $(N-1)N^3$ possible transitions resulting from an iteration of one sorting agent' process. For a specific value of N, the corresponding model can be automatically generated via a software program. To make the PRISM specification compact, we decided to model collective sort as composed of a single sorting agent alternatively working on each of the N available spaces. The complete specification of collective sort with $N = 3$ is reported in Figure 7.1. For each tuple space j, tuple concentrations are expressed by global variables `Kij` where `i` represents tuple kind `i`. On the other hand, total tuple concentration for each space j is modelled by formula `totj`. Figure 7.1 reports the definition of every transition that can occur between every permutation of 2 tuple spaces chosen as source and destination among the 3 available. Moreover, each transition refers to a specific permutation of source and destination kind that could be chosen over an iteration of the sorting agent's process. A transition is associated a probability depending on the probability of choosing 2 given tuple spaces as source and target ($\frac{1}{6}$ with $N = 3$) and the concentration of the specific kinds considered in the transition. For example, consider the transition

```
[] k11>0 & k22>0 & k21>0 & k22<n ->
  (1/6*k11/tot1*k22/tot2) :
  (k21'=k21-1) & (k22'=k22+1);
```

which models the case where kind 1 is chosen as a reference kind on space 1 and kind 2 as a target kind on space 2. Accordingly to the sorting agent's protocol described in Section 3.3, this implies that a tuple of kind 2 (if any) is moved from space 1 to space 2. The probability for this situation to occur is modelled as a joint probability `(1/6*k11/tot1*k22/tot2)`, where `1/6` is the probability of choosing tuple space 1 and 2 as source and target space respectively, while `k11/tot1` and `k22/tot2` are the probabilities of choosing kind 1 on space 1 and kind 2 on space 2. In particular, the latter probabilities model the semantics of the `urd` primitive.

The choice of the same kind on both source and destination space is not explicitly modelled as it does not lead to any change of state in the model. In other words, the choice of the same kind on both spaces would never influence the achievement of complete sorting, so that it is pointless to explicitly model such a situation by a transition. Furthermore, for each of the $N(N-1)$ possible couples of source and target space, every possible tuple kind permutation is modelled by a transition

With regard to scalability in N, the decision of explicitly model all the possible permutations occurring from process execution could appear unreasonable since the number of permutations to be modelled as transitions grows as $(N-1)N^3$. However, providing a model efficient from the standpoint of scalability is out of the scope of the work and will be matter of future investigation.

```
dtmc
const int NK;
global k11 : [0..n] init NK; global k12 : [0..n] init NK; global k13 : [0..n] init NK;
global k21 : [0..n] init NK; global k22 : [0..n] init NK; global k23 : [0..n] init NK;
global k31 : [0..n] init NK; global k32 : [0..n] init NK; global k33 : [0..n] init NK;
formula tot1 = k11 + k21 + k31; formula tot2 = k12 + k22 + k32;
formula tot3 = k13 + k23 + k33; formula n = NK * 3;
module ts
    //***T1 --> T2****
    //k1
    [] k11>0 & k22>0 & k21>0 & k22<n -> (1/6 * k11/tot1 * k22/tot2) : (k21'=k21-1) & (k22'=k22+1);
    [] k11>0 & k32>0 & k31>0 & k32<n -> (1/6 * k11/tot1 * k32/tot2) : (k31'=k31-1) & (k32'=k32+1);
    //k2
    [] k21>0 & k12>0 & k11>0 & k12<n -> (1/6 * k21/tot1 * k12/tot2) : (k11'=k11-1) & (k12'=k12+1);
    [] k21>0 & k32>0 & k31>0 & k32<n -> (1/6 * k21/tot1 * k32/tot2) : (k31'=k31-1) & (k32'=k32+1);
    //k3
    [] k31>0 & k12>0 & k11>0 & k12<n -> (1/6 * k31/tot1 * k12/tot2) : (k11'=k11-1) & (k12'=k12+1);
    [] k31>0 & k22>0 & k21>0 & k22<n -> (1/6 * k31/tot1 * k22/tot2) : (k21'=k21-1) & (k22'=k22+1);
    //*****************
    //***T2 --> T1****
    //k1
    [] k12>0 & k21>0 & k22>0 & k21<n -> (1/6 * k12/tot2 * k21/tot1) : (k22'=k22-1) & (k21'=k21+1);
    [] k12>0 & k31>0 & k32>0 & k31<n -> (1/6 * k12/tot2 * k31/tot1) : (k32'=k32-1) & (k31'=k31+1);
    //k2
    [] k22>0 & k11>0 & k12>0 & k11<n -> (1/6 * k22/tot2 * k11/tot1) : (k12'=k12-1) & (k11'=k11+1);
    [] k22>0 & k31>0 & k32>0 & k31<n -> (1/6 * k22/tot2 * k31/tot1) : (k32'=k32-1) & (k31'=k31+1);
    //k3
    [] k32>0 & k11>0 & k12>0 & k11<n -> (1/6 * k32/tot2 * k11/tot1) : (k12'=k12-1) & (k11'=k11+1);
    [] k32>0 & k21>0 & k22>0 & k21<n -> (1/6 * k32/tot2 * k21/tot1) : (k22'=k22-1) & (k21'=k21+1);
    //*****************
    //***T2 --> T3****
    //k1
    [] k12>0 & k23>0 & k22>0 & k23<n -> (1/6 * k12/tot2 * k23/tot3) : (k22'=k22-1) & (k23'=k23+1);
    [] k12>0 & k33>0 & k32>0 & k33<n -> (1/6 * k12/tot2 * k33/tot3) : (k32'=k32-1) & (k33'=k33+1);
    //k2
    [] k22>0 & k13>0 & k12>0 & k13<n -> (1/6 * k22/tot2 * k13/tot3) : (k12'=k12-1) & (k13'=k13+1);
    [] k22>0 & k33>0 & k32>0 & k33<n -> (1/6 * k22/tot2 * k33/tot3) : (k32'=k32-1) & (k33'=k33+1);
    //k3
    [] k32>0 & k13>0 & k12>0 & k13<n -> (1/6 * k32/tot2 * k13/tot3) : (k12'=k12-1) & (k13'=k13+1);
    [] k32>0 & k23>0 & k22>0 & k23<n -> (1/6 * k32/tot2 * k23/tot3) : (k22'=k22-1) & (k23'=k23+1);
    //*****************
    //***T3 --> T1****
    //k1
    [] k13>0 & k21>0 & k23>0 & k21<n -> (1/6 * k13/tot3 * k21/tot1) : (k23'=k23-1) & (k21'=k21+1);
    [] k13>0 & k31>0 & k33>0 & k31<n -> (1/6 * k13/tot3 * k31/tot1) : (k33'=k33-1) & (k31'=k31+1);
    //k2
    [] k23>0 & k11>0 & k13>0 & k11<n -> (1/6 * k23/tot3 * k11/tot1) : (k13'=k13-1) & (k11'=k11+1);
    [] k23>0 & k31>0 & k33>0 & k31<n -> (1/6 * k23/tot3 * k31/tot1) : (k33'=k33-1) & (k31'=k31+1);
    //k3
    [] k33>0 & k11>0 & k13>0 & k11<n -> (1/6 * k33/tot3 * k11/tot1) : (k13'=k13-1) & (k11'=k11+1);
    [] k33>0 & k21>0 & k23>0 & k21<n -> (1/6 * k33/tot3 * k21/tot1) : (k23'=k23-1) & (k21'=k21+1);
    //*****************
    //***T3 --> T2****
    //k1
    [] k13>0 & k22>0 & k23>0 & k22<n -> (1/6 * k13/tot3 * k22/tot2) : (k23'=k23-1) & (k22'=k22+1);
    [] k13>0 & k32>0 & k33>0 & k32<n -> (1/6 * k13/tot3 * k32/tot2) : (k33'=k33-1) & (k32'=k32+1);
    //k2
    [] k23>0 & k12>0 & k13>0 & k12<n -> (1/6 * k23/tot3 * k12/tot2) : (k13'=k13-1) & (k12'=k12+1);
    [] k23>0 & k32>0 & k33>0 & k32<n -> (1/6 * k23/tot3 * k32/tot2) : (k33'=k33-1) & (k32'=k32+1);
    //k3
    [] k33>0 & k12>0 & k13>0 & k12<n -> (1/6 * k33/tot3 * k12/tot2) : (k13'=k13-1) & (k12'=k12+1);
    [] k33>0 & k22>0 & k23>0 & k22<n -> (1/6 * k33/tot3 * k22/tot2) : (k23'=k23-1) & (k22'=k22+1);
    //*****************
endmodule
```

Figure 7.1: The PRISM program modelling collective sort with $N = 3$.

7.4 Simulating and Verifying Collective Sort

Collective Sort was analysed by adopting the model shown in Section 7.3.3. As an interesting emergent property to be analysed, we focussed on *complete sorting*. Complete sorting was investigated on different PRISM specifications of the collective sort modelling the case with $N = 3$, $N = 4$, $N = 5$, and $N = 6$. To this end, we adopted both exact and approximated probabilistic model checking techniques as explained in Section 7.2.3.

As regards collective sort with $N = 3$, we performed analysis on complete sorting by considering different tuple concentrations. Furthermore, we considered initial states characterised by tuples uniformly distributed per kind among the available spaces: e.g. considering 6 tuples per kind, the resulting initial configuration for $N = 3$ would be $(2, 2, 2), (2, 2, 2), (2, 2, 2)$. The property *"Which is the probability for collective sort to reach complete sorting?"* is expressed as a PCTL formula by the PRISM syntax

```
P=? [true U "complete-sorting"]
```

where `U` is the unbounded until operator and `complete-sorting` is a label expressed as

```
label "complete-sorting" =
            ok1 & ok2 & ok3 &
            tot1=n & tot2=n & tot3=n;

formula ok1 = k11=n | k21=n | k31=n;
formula ok2 = k12=n | k22=n | k32=n;
formula ok3 = k13=n | k23=n | k33=n;

formula tot1 = k11 + k21 + k31;
formula tot2 = k12 + k22 + k32;
formula tot3 = k13 + k23 + k33;
```

In particular, `okj` is a boolean formula that gets `true` if a tuple kind completely aggregates on space `j`—`n` represents the total number of tuples per tuple kind. Accordingly, label `complete-sorting` gets `true` when every space aggregates all the tuples of exactly one kind and the total number of tuples per space is equal to `n`. For the sake of brevity, the probability of achieving complete sorting is hereafter referred to as *probability of convergence* P_c.

Probability of convergence for collective sort with $N = 3$ was analysed for values of total concentration of tuples N_t ranging from 3 to 30 tuples per kind. Exact model checking was adopted for values of N_t ranging from 3 to 12. As a reference value of the dimension of the internal MTBDD model generated by PRISM, the model corresponding to the instance with $N_t = 12$ is defined by $746,983$ states and $9,603,897$ transitions. The total time taken for model construction is around 300 seconds on a computer equipped with a 2.5 GHz P4 processor, 2 GB DDR RAM, and 800 MHz System Bus. For this

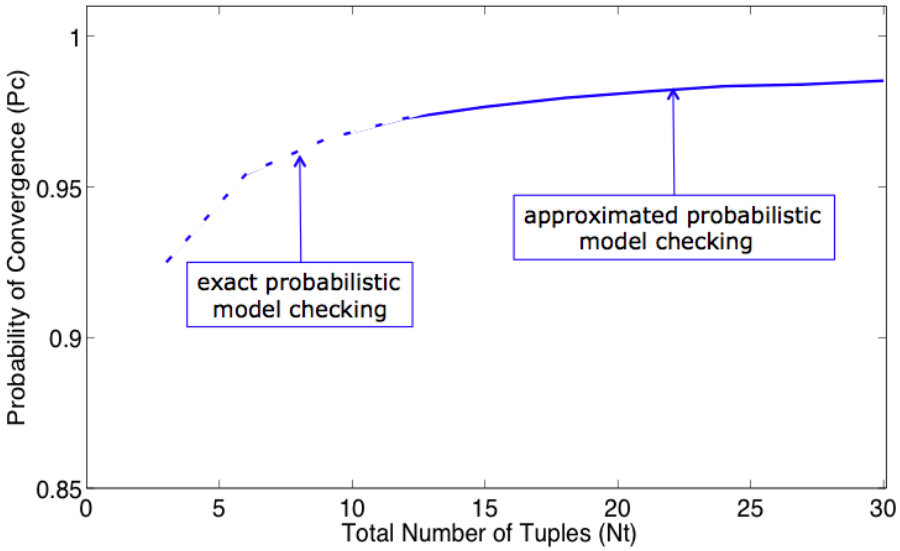


Figure 7.2: Trend of probability of convergence with $N = 3$ for N_t ranging from 3 to 30.

reason, complete sorting on instances with $N_t > 12$ was analysed by using approximated probabilistic model checking. In particular, this approach was adopted for instances with N_t in the range $[15, 30]$ by performing $16,481$ simulation runs (also known as samples) for each value of N_t in the specified range. This allowed to obtain values of P_c featuring an approximation $\epsilon = 0.05$ and a confidence $\delta = 10^{-10}$.

Figure 7.2 shows the trend of P_c for N_t ranging from 3 to 30. Exact probabilistic model checking was used for values of N_t up to 12 tuples. For subsequent values of N_t, given the amount of memory and computational resources needed to build the internal MTBDD model, approximated model checking was adopted. Figure 7.2 shows that P_c ranges from 0.92 for $N_t = 3$ to 0.98 for $N_t = 30$. Such a growing tendency is due to the fact that, as tuple concentration increases, moving a tuple from one space to another produces less noise in system evolution. In other words, the importance of moving a tuple is relative to the total concentration of tuples in the system: moving a tuple in a system with a low tuple concentration has a stronger influence than moving a tuple in a system with a higher tuple concentration.

The same analysis was also performed considering $N = 4$ and $20 \leq N_t \leq 1200$. Since this case is characterised by a much higher complexity in term of model size – 1771^4 states with $N_t = 20$ – we relied only on approximated model checking by adopting the same values of ϵ and δ as for $N = 3$. The corresponding results are reported in Figure 7.3, which shows the trend of P_c over N_t covering the aforementioned range.

As a further analysis, the value of N_t was fixed at 100 tuples and the trend of P_c analysed for different values of N in the range $(3, 6)$ by relying again on approximated

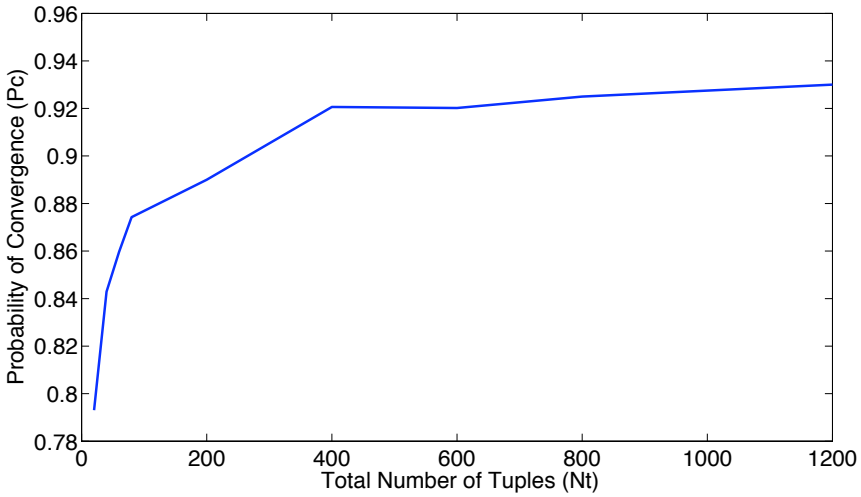


Figure 7.3: Trend of probability of convergence over N_t with $N = 4$.

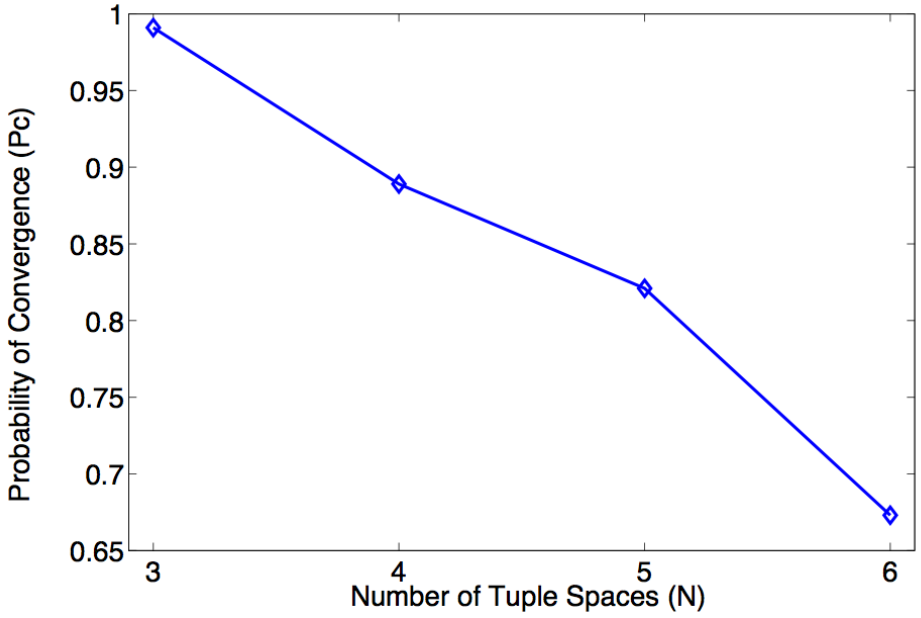


Figure 7.4: Trend of probability of convergence with $N_t = 100$ for N ranging from 3 to 6.

probabilistic model checking. The corresponding results are presented in Figure 7.4, which clearly shows that P_c decreases as N grows. In fact, as N increases, the number of potential local minimum states increases as well. As a consequence, the probability of achieving complete sorting is lower for larger collective sort instances.

The results presented throughout the chapter make it clear that probability of obtaining complete sorting decreases as *(i)* tuple concentration decreases and/or *(ii)* N grows. An extension of collective sort for increasing the probability of complete sorting based on simulated annealing is proposed in Section 3.3.4, and will be matter of future analysis by adopting the hybrid approach discussed here.

According to the presented results, the adoption of probabilistic model checking, either in its exact or approximated version, becomes essential to formally validate the behaviour of systems where self-organisation is exploited like collective sort. In fact, such systems can often be tested by simulation only, which however can just provide informal evidence of expected behaviour. We also believe that probabilistic model checking will become essential also in the tuning stage that has usually to be undertaken since the early stage of self-organising-system design. As a matter of fact, tuning the working parameters of a self-organising system is indeed crucial to deliver a behaviour meeting the expected functional requirements and dynamics. Accordingly, formal verification plays a key role in defining a methodology to efficiently and rapidly find adequate values of the working parameters from the standpoint of desired behaviour.

8

Related Work

This chapter discusses related works, focussing especially on coordination. The last years have witnessed a lot of works on self-organisation for software system engineering and computer science in general. Some of the computer-science-related fields where the adoption of self-organisation has been investigated are cited in [MMTZ06], including grid computing, pervasive systems, database organisation, security issues, robotic systems, mobile ad-hoc networks, sensor networks, and amorphous computing. Several conference series on these topics have as well taken place: the most relevant are the International Workshop on Self-Adaptive Software (IWSAS) held in 2000 and 2001, the International Workshop on Self-* Properties in Complex Information Systems (Self-Star) held in 2004, the International Workshop on Engineering Self-Organizing Applications (ESOA) held from 2003 to 2006, the International Workshop on Self-Managed Networks, Systems and Services (SelfMan) held in 2005 and 2006, the International Workshop on Self-Organizing Systems (IWSOS) held since 2006, and the IEEE Conference on Self-Adaptive and Self-Organizing Systems (SASO) whose first edition dates back to 2007. In particular SASO, resulting from the fusion of ESOA, SelfMan, Self-Star, and IWSAS, is now the major event in the field of self-organisation applied to software systems, acting as a main reference for many computer science fields [sas07, sas08]. As regards journals, the ACM Transactions on Autonomous and Adaptive Systems is with no doubt the most important reference in self-organisation. However, providing a complete discussion of the many fields where self-organisation has been applied to is not only out of the scope, but would also require a disproportionate effort to the actual focus of the thesis. As a consequence, this chapter mainly deals with describing works regarding self-organisation approaches in coordination and experiments concerning techniques inspired by swarm intelligence. The decision of including swarm intelligence comes from the consideration that most of the coordination services described here – in particular SwarmLinda and tuple sorting – rely on swarm intelligence.

8.1 Existing Self-Organising Approaches to Coordination

Traditional coordination models like LINDA [Gel85], REO [Arb04], and their many derivatives, always enact coordination rules that are *predictable*, namely, whose impact on system interactions is known, and fully re-producible, though forms of non-determinism are possible. This is a natural consequence of coordination models and languages starting as a branch of traditional software engineering, which promotes methods of software design with the goal of building correct, efficient, and predictable applications.

However, as highlighted in Chapter 1, the emergence of new application scenarios – like distributed artificial intelligence, multi-agent systems, self-organising systems, and pervasive computing – is pushing towards the development of radically new approaches to coordination [ZP03, MMTZ06].

There still are few works on coordination models and languages, that relate somehow to self-organisation. For instance, TOTA [MZ04] is an infrastructure of tuple spaces where tuples get copied and spread in neighbouring nodes along with a fading mechanism so as to form so-called *computational fields* (co-fields), which can be used by agents to find each other (and retrieve data items) in spite of their mobility and of changes in the network topology. In this infrastructure, *topology* and *locality* of interactions play a crucial role since a distributed data structure of tuples (the co-field) is created on a step-by-step basis.

Moreover, few formal models have been introduced to tackle self-organisation-related aspects in coordination, such as stochasticity. In STOKLAIM [DNLM05], a formal model extending LINDA is described where agents insert and retrieve tuples in a stochastic way, namely by specifying an operation rate that affects timing and probability of the corresponding primitive execution. Similarly in [BGLZ04, BGLZ05], formal underpinnings for probabilistic extensions of LINDA are studied, featuring the ability to specify a probability (or a rate) for the execution of primitives—e.g. for retrieving certain tuples instead of others.

Finally, we note that several works outside the core coordination community are actually addressing the problem of mediated interaction through a self-organising environment. A key example is that of multi-agent system environments inspired by *stigmergy* [HM99], a technique whereby ants coordinate their behaviour. The main idea of this paradigm is that agents leave pheromone-like data-items on the local environment, that as time passes *(i)* distribute in neighbouring nodes, *(ii)* aggregate, and *(iii)* fade. By properly exploiting such data items spread in the environment, agents can self-organise their behaviour, e.g. by adaptively creating and maintaining a path towards a resource [PBS02]. The potential feasibility of tuple-based coordination infrastructures to the aforementioned framework is evident—as witnessed by some recent experiment with TOTA.

All these works show that there is an increasing interest in the application of self-organising approaches to coordination models and languages, aimed at achieving adaptivity properties in the management of interactions, as required e.g. by today's pervasive

computing scenarios.

8.2 Swarm Intelligence

Initially developed in chemistry and physics, *self-organisation theory* describes the *emergence* of macroscopic patterns generated from microscopic interactions of system's components. In this theory, individual complexity is not excluded, but at some level of description it is possible to provide an explanation of a complex behaviour in terms of simple entities [BDT99]. This point is stressed also in some of the self-organising coordination services presented in the thesis: indeed, the proposed solutions often rely on very simple agents as well.

Self-organisation theory encompasses biological systems: in particular, it is suitable to describe the dynamics observed in insect colonies. Social insects are well known for their ability to collectively solve problems despite the individual limited perception and action capabilities. The wide repertoire of behaviours exhibited by insect colonies ranges from cooperative transport to complex structures building [BDT99, CDF+01]. In particular, sorting and clustering phenomena are observed in various forms and across several insect species. Those phenomena have also been the main inspiration for the self-organising coordination services proposed throughout the thesis. Clustering involves gathering items scattered into the environment and organizing them in piles [DGF+91, BDT99], while sorting involves more complex patterns and shapes like concentric rings [FSF92]. As an example of clustering, consider pile formation in termite colonies: termites wander in the environment, pick up scattered wood chips and arrange them in clusters. Resnick proposed a simple model able to recreate this collective dynamics [Res97]: individual termite behaviour is specified by two simple rules: *(i)* if nothing is carried, pick up an item as one is encountered; *(ii)* if carrying an item, drop it when encountering another one. Observations across several ant species reported that corpses are clustered in small cemeteries in order to clean up nests [BDT99]. Similarly, broods of the same size are clustered while broods of different size are sorted into concentric annuli. Concentric patterns have been observed in hive organization by honeybees: central area is occupied by broods and surrounded by pollen and honey placed in concentric rings [Cam91, CDF+01].

Although the actual mechanisms regulating these behaviours are not fully understood yet, researchers agree on the fundamental role of local density perception. In literature there are several models able to replicate the observed dynamics: e.g. collective robotics has produced several notable results through experimentation with physical robots.

8.2.1 Collective Robotics

Collective robotics is characterised by scenarios involving multiple autonomous robots coordinating one another by using local sensing and actions as well as limited communication. In the last decade, collective robotics has probably been the most active field

involved in developing artificial systems inspired by social insects behaviour. For instance, the main goal of the European project SWARM-BOTS has been to study novel approaches for developing self-organising and self-assembling swarms of robots [DTM+05]. However, among the several scenarios investigated in that specific context, none is strictly related to sorting behaviours.

Instead, other works in collective robotics deal with the problem called *spatial sorting* which is closely related to the coordination services presented in Chapters 3-5, but solved by mechanisms more similar to what observed in social insects. A taxonomy of spatial sorting problems reflecting actual insect behaviours is provided in [MHH98]:

clustering — the only available type of item is grouped in a small fraction of the available space;

segregation — each type of item is clustered but contiguous to other clusters;

patch sorting — each type of item is clustered but distant from other clusters;

annular sorting — clusters of different types of item are arranged in concentric rings.

Applying this classification to tuple organisation allows a better understanding of the problem. Let nt be the number of tuple spaces in the network, and nk the number of tuple kinds:

- when $nk = 1$ we deal with a problem of clustering;
- the case featuring $nk > 1$ and $nk < nt$ resembles patch sorting;
- when $nk > 1$ and $nk \geq nt$, we have segregation;

Annular sorting requires a specific notion of topology we have not taken into account so far, but would be an interesting approach to consider as well. For instance, collective sort focusses on the case $nk = nt$, tuple clustering deals with the case $nk = 1$, while SwarmLinda and tuple sorting are aimed at solving the general case where $nk > 1$.

In [DGF+91], Deneubourg et al. proposed a probabilistic model for brood clustering, based on two probabilities P_p and P_d which depend on estimating local density f of items: in [DGF+91] f is evaluated as the number of items encountered in time interval T. While researchers agree on the fact that ants actually perceive local density when picking up an item, whether they do so when dropping an item is still matter of debate [MSFS+06]. In the struggle to find a minimal set of rules, as pointed out in [MHH98], it is possible to achieve spatial sorting by exploiting techniques similar to self-sorting, i.e. sorting that occurs under environmental forces like gravity. In [LF94] it has been proposed a generalisation based on a similarity function for exploratory data analysis [BDT99]. The Lumer Faieta's function replaces f: basically, such a function minimises intra-cluster distance with respect to inter-cluster distance. Results coming from this research could be applied to evaluate an improved version of the solutions discussed in the thesis.

Besides, it should be noted that collective sort, SwarmLinda, and tuple sorting are only apparently related to *data clustering* as defined in [JMF99], which is indeed a different problem than spatial clustering in robotics. Indeed data clustering only defines techniques for associating patterns to groups of related data items. However, data clustering could be applied at design-time to the set of tuples an application has to handle, in order to identify N clusters of information that will be used as kinds to be grouped.

9 Conclusion

This chapter summarises the work presented in the thesis, highlighting the corresponding contributions and shortcomings, and finally tracing a feasible path for future work.

9.1 Contributions

The work presented in this thesis is one of the really few attempts at understanding the role of self-organisation for software system coordination, resulting in a concrete proposal in terms of self-organising coordination strategies. Besides defining quite a general abstract framework for self-organising coordination systems, this thesis also showed a set of self-organising coordination services that range from strategies for solving coordination issues in specific domains, to proposals of middleware extending traditional coordination models by self-organising principles. In particular, *collective sort* was the first concrete step in this direction, devising a self-organising strategy for tuple organisation in distributed sets of fully connected tuple spaces: the main results of this work are reported in [CGV07, VCG07, CVG09]. Then, the main focus was moved to middleware, realising a swarm-intelligence-based extension of the traditional LINDA coordination model. The result of this activity is SwarmLinda, that proposed an extended semantics of the LINDA `out` operation leading to a self-organising tuple distribution, whereby it is possible to improve the scalability of systems built on top of the middleware: the most remarkable results of this work are shown in [CMVT07b, CMVT07a]. Finally, as a result of the generalisation of the scenario underlying collective sort, a new strategy to tuple clustering and sorting was provided, that is again inspired by swarm intelligence. In particular, the generalisation consists in making no assumption about the topology of the tuple-space networks. The early results of this work are reported in [CVS08, CV08].

As a minor contribution, this work also dealt with stochastic simulation as a means for analysing and prototyping self-organising coordination services. To this end, a stochastic simulation framework was developed in MAUDE and summarised in [GVCO07]. The work on simulation resulted also in a proposal of a methodology for the design of self-organising systems, centred around formal modelling, stochastic simulation, and verification as reported in [GVCO08]. There, the importance of verification is also clearly remarked,

though really few works on verifying self-organising systems can be found in literature. Accordingly, the last year of the PhD course was in part spent working on verification for self-organising systems: in particular, probabilistic model checking was investigated as a suitable technique to automatic verification of systems showing probabilistic and stochastic behaviours, as typically occurs in self-organising systems. Among the tools experimented with, the PRISM model checker was chosen and exploited to verify the emergent property of complete sorting on collective sort, as presented in [CV09a, CV09b].

9.2 Main Shortcomings

The engineering of self-organisation techniques into software systems is still a largely unknown problem, since it is not clear how to link the very need of software engineering to find a meaningful set of functional requirements with the fundamental essence of self-organisation that instead promotes the emergence of global behaviour – which is that expected to fulfil functional requirements of the system at hand – as a response to local interactions among system components. As a consequence, the resulting engineering complexity of self-organising systems is due to the fact that traditional reductionist approaches are not feasible when dealing with self-organisation. Probably the first attempt at solving this issue is represented by the work reported in [GVCO08], where the problem is addressed by proposing the use of formal modelling, simulation, and verification tools within a clearly defined design methodology, as a way to cope with the intrinsic complexity of self-organising systems.

Another limitation of this work regards verification, and lies in the fact that automatic techniques like probabilistic model checking are still largely unfeasible when applied to self-organising systems, due to their typically large-scale nature. As a consequence, the model to be verified often results extremely large in terms of variables required for specifying the dynamics of the system, definitely leading to the state-space-explosion problem, which is a well-known issue in model-checking, making memory requirements rapidly grow. So far, even strategies for building parallel and distributed model checkers seem just a partial solution to this issue.

9.3 Future Work

There are several works that would be compelling to pursue in future research activities. First of all, as regards self-organising coordination services, it would be interesting to capitalise the devises coordination techniques by proposing a comprehensive novel coordination middleware based on self-organisation. We are currently experimenting with a novel model of self-organising coordination based on biochemistry, where the nodes of a distributed network can be though of as cell compartments, while the management of interaction among the processes in the network is handled in terms of chemical reactions

acting on data, playing the role of chemical reactants. An early proposal of the middleware can be found in [VZCM09].

As far as coordination languages are concerned, we have also started experimenting with the ReSpecT language to actually implement some of the coordination services proposed in the thesis: in the end, the adoption of ReSpecT for devising self-organising coordination services will play an important role in future research activities.

Last but not least, probabilistic model checking needs to be more deeply investigated so as to find a set of more effective techniques to be applied to self-organising system verification.

A

Maude Specification of the Stochastic Simulation Engine

```
mod RANDOM-UTILITIES is
    pr COUNTER .
    pr RANDOM .
    pr CONVERSION .

    op randrange : Nat -> Nat .     *** A RANDOM NUMBER IN A RANGE
    op rand : -> [Float] .          *** A RANDOM FLOAT IN  0 - 1

    *** IMPLEMENTATION
   eq rand = float(random(counter)/ 4294967295) .
   eq randrange( N:Nat ) = floor( rat (rand) * N:Nat ) .
endm

mod STOCHASTIC-SELECTION is
   pr RANDOM-UTILITIES .
   pr LIST{Float} .

   sort Event .

   op now : -> Float .  *** CONSTANT RATE FOR INSTANTANEOUS ACTIONS

   *** AN EVENT
   op @ : [Nat] [Float] -> Event [ctor] .

   *** RANDOM EVENT GENERATION
   op next : List{Float} -> Event .
endm

mod STOCHASTIC-SELECTION-IMPLEMENTATION is
    pr STOCHASTIC-SELECTION .
```

```
    *** INTERNALS
    vars L L'       : List{Float} .
    vars F F' F''   : Float .
    var N           : Nat .

    eq now = 1000000000.0 .

    eq next(nil) =  @( -1 , 0.0 ) .
    ceq next(L) = @( $sample(L,F), $dtime(F) )
                            if F := $sum(L) /\ F =/= 0.0 [owise] .
    eq next(L) = @( -1 , 0.0 ) [owise] .

    op $sample : List{Float} Float -> Nat .
    ceq $sample( L , F ) = $ssample(rand, F' ,0, L' )
                            if (F' L') := $normalize(L,F) .

    op $dtime  : Float -> Float .
    eq $dtime( F ) = (1.0 / F) * log( 1.0 / rand ) .

    op $ssample : Float Float Nat List{Float} -> Nat .
    ceq $ssample ( F , F' , N , L ) = N  if F < F' .
    ceq $ssample ( F , F' , N , nil ) = s N if F >= F' .
    eq $ssample ( F , F' , N , (F'' L) ) =
                            $ssample( F , F' + F'', s N , L ) [owise] .

    op $sum : List{Float} -> Float .
    eq $sum( nil ) = 0.0 .
    eq $sum( F L ) = F + $sum (L) .

    op $normalize : List{Float} Float -> List{Float} .
    eq $normalize( nil , F ) = nil .
    eq $normalize( (F' L) , F ) =
                            ((F' / F) $normalize( L , F ) ) [owise] .
endm

set clear rule off .

fmod STANDARD-CARRIER is
    pr FLOAT .
    pr BOOL .
    pr NAT .

    sort State Action States Effect Effects Observation .
```

```
    subsort State < States .
    subsort Effect < Effects .

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

    op nil : -> Effects .
    op _;_ : Effects Effects -> Effects [ ctor assoc id: nil ] .
    op _#_->[_] : Action Float States -> Effect [ctor] .

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

    op obs : Nat State Float -> Observation .
endfm

fth CARRIER is
    pr FLOAT .
    pr BOOL .
    pr NAT .

    sort State Action States Effect Effects Observation .
    subsort State < States .
    subsort Effect < Effects .

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

    op nil : -> Effects .
    op _;_ : Effects Effects -> Effects [ ctor assoc id: nil ] .
    op _#_->[_] : Action Float States -> Effect [ctor] .

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

    op obs : Nat State Float -> Observation .
endfth

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

    sort Step Observations Trace Steps Evt Evts .
    subsort X$Observation < Observations .
    subsort Step < Steps .
```

```
    subsort Evt < Evts .

    op [_:_@_] : Nat X$State Float -> Step [ctor format (ni d d d d d d)] .
    op evt(_,_,_) : Nat X$Observation Float -> Evt
                                          [ctor format (ni d d d d d d d)] .
    op _,_ : Observations Observations -> Observations
                                    [ctor assoc id: empty format (d d n d)] .
    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)].
    op nil : -> Steps .
    op _+_ : Steps Steps -> Steps [ctor assoc id: nil ] .
    op nil : -> Evts .
    op __ : Evts Evts -> Evts [ctor assoc id: nil ] .
endm


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

    op activities : X$Effects -> List{Float} .
    op newState : Nat X$Effects -> X$State .
    op one : X$States -> X$State .
    op evalEffects : [X$Effects] -> X$Effects .

*** INTERNALS
    var S : X$State .
    var LS : X$States .
    var Es : X$Effects .
    var E : X$Effect .
    var A : X$Action .
    vars N N1 : Nat .
    vars F F1 : Float .

    eq evalEffects(Es) = Es .

    op $size : X$States -> Nat .
    op $get : X$States Nat -> X$States .

    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].

    eq one ( S ) = S .
    eq one ( LS ) = $get( LS , randrange($size(LS)) ) [owise] .

    eq $size ( S ) = 1 .
    eq $size ( S LS ) = s $size ( LS ) [owise] .

    eq $get ( S , 0 ) = S .
    eq $get ( S LS , 0 ) = S .
    eq $get ( S LS , s N ) = $get ( LS , N ) [owise] .
endm

mod STOCHASTIC-TRACES-ENGINE{ X :: CARRIER } is
    pr STOCHASTIC-TRACES-FUNCTIONS{X} .

    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 f : X$State -> X$State .
    eq f(S) = newState(0,evalEffects(S ==>)) .

    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(0,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(0 , O , F )
                        if not temp(S)
                        /\ O := obs(0,S,F) .

    op series : Nat Step -> Evts .
    eq series ( 0 , P ) = nil .
    eq series ( s N , P ) = last( P < empty > ) series( N , P ) .
endm
```

B

Maude Specification of the Collective Sort Strategy

```
mod COLLECTIVE-SORTING-TYPES is
  pr CONVERSION .
  pr QID .

  var F : Float .

  ops rl rh rho : -> Float .
  eq rl = 0.025 .
  eq rh = 0.25 .
  eq rho = 0.1 .

  op decrease : Float -> Float .
  ceq decrease(F) = F - rho * (rh - rl) if F - rho * (rh - rl) > rl .
  eq decrease(F) = rl [owise] .

  sort TupleType Tuple TupleMSet Space QList Items Rate DataSpace .
  subsort Qid < TupleType .

  op ? : -> TupleType .

  *** TUPLES
  op _[_] : TupleType Nat -> Tuple [ctor] .
  subsort Tuple < TupleMSet .
  op empty : -> TupleMSet [ctor] .
  op _|_ : TupleMSet TupleMSet -> TupleMSet [ctor assoc comm id: empty prec 6] .

  *** TUPLE SPACE
  op <_@_> : Nat TupleMSet -> Space [ctor format (n d d d d d) ] .

  *** QID LIST, THAT IS, TUPLE KINDS
```

```
  subsort  TupleType < QList .
  op nilql : -> QList [ctor] .
  op _,_ : QList QList -> QList [ctor assoc id: nilql prec 7] .

  sort Total .
  subsort Total < Tuple .
  op tot : Nat -> Total [ctor] .

  *** STATE-ITEMS
  op init : -> Items [ctor] .
  op [_] : Nat -> Items [ctor] .
  op [_] : TupleType -> Items [ctor] .
  op {_} : Nat -> Items [ctor] .
  op #_# : Nat -> Items [ctor] .
  op #_# : Float -> Items [ctor] .

  op {_} : DataSpace -> Items [ctor] .
  op _;_ : Items Items -> Items [ctor assoc prec 8] .

  *** RATES
  op r : Nat Float -> Rate [ctor] .

  *** DATASPACE
  subsort Items Space < DataSpace . *** Rate
  op empty : -> DataSpace [ctor] .
  op _|_ : DataSpace DataSpace -> DataSpace [ctor assoc comm prec 10] .

endm

mod COLLECTIVE-SORTING-FUNCTIONS is
  pr COLLECTIVE-SORTING-TYPES .
  pr STOCHASTIC-SELECTION-IMPLEMENTATION .
  pr LIST{Nat} .

  var Q Q1 : TupleType .
  var QL QL1 : QList .
  var N N': Nat .
  var MT : TupleMSet .
  var F QT T : Float .
  var L : List{Float} .
  var LN : List{Nat} .

  op one : Nat -> Nat .
  eq one( N:Nat ) = randrange(N:Nat).
```

```
op size : QList -> Nat .
eq size( nilql ) = 0 .
eq size( Q , QL ) = s size( QL ) .

op length : List{Float} -> Nat .
eq length( nil ) = 0 .
eq length( F L ) = s length( L ) .

op get : QList Nat -> Qid .
eq get ( (Q , QL ) , 0 ) = Q .
eq get ( (Q , QL ) , s N:Nat ) = get ( QL , N:Nat) .
eq get(nilql, N:[Nat]) = 'noKind [owise] .

op choose : QList -> Qid .
eq choose ( Q ) = Q .
eq choose ( QL ) = get( QL , one(size(QL))) [owise] .

op occurringTuples : TupleMSet -> QList .
eq occurringTuples (tot(N:Nat)) = nilql .
eq occurringTuples ( ( Q [ 0 ] ) | MT ) = occurringTuples( MT ) .
eq occurringTuples ( ( Q [ N:Nat ] ) | MT ) =
               ( Q , occurringTuples( MT ) ) [owise] .

op noNoise : QList -> QList .
eq noNoise( (? , QL) ) = noNoise(QL) .
eq noNoise( (Q , QL) ) = (Q, noNoise(QL)) .
eq noNoise( nilql ) = nilql .

op sample : List{Float} -> [Nat] .
ceq sample( L ) = $sample( L , F ) if F := $sum(L) /\ F =/= 0.0 .
eq sample( L ) = one( length ( L ) ) [owise] .

op quantities : QList TupleMSet -> List{Float} .
eq quantities( nilql, MT ) = nil .

eq quantities( (Q, QL) , (Q [ N:Nat ]) | MT ) =
            ( float(N:Nat) quantities( QL , MT ) ) .
eq quantities( (Q, QL) , MT ) = ( 0.0 quantities( QL , MT ) ) [owise] .

op log2 : Float -> Float .
eq log2( F ) = log( F ) / log( 2.0 ) .

op info : Float Float -> Float .
```

```
eq info( QT , F ) = ( ( F / QT ) * log2( QT / F ) ) .

op entropy : Float List{Float} -> Float .
eq entropy( QT , nil ) = ( 0.0 ) .
ceq entropy( QT , F L ) = info( QT , F ) + entropy( QT , L ) if F > 0.0 .
eq entropy( QT , F L ) = 0.0 + entropy( QT , L ) [owise] .

op sp-entropy : List{Float} -> Float .
eq sp-entropy( F L ) =
          entropy( $sum( F L ) , F L ) / log2( float (length( F L )) ) .

op occursQ : TupleType QList -> Bool .
eq occursQ( Q , nilql ) = false .
eq occursQ( Q , (Q , QL) ) = true .
eq occursQ( Q , (Q1 , QL) ) = occursQ ( Q , QL ) [owise] .


op out : DataSpace -> Bool .
eq out( S:DataSpace ) = out ( S:DataSpace , nilql) .

op out : DataSpace QList -> Bool .

ceq out ( < N:Nat @ MT > | S:DataSpace , QL ) = false
            if  QL1 := noNoise(occurringTuples(MT)) /\
               size(QL1) >= 2 .
ceq out ( < N:Nat @ MT > | S:DataSpace , QL ) = false
            if Q1 := noNoise(occurringTuples(MT)) /\
                occursQ( Q1, QL ) .
ceq out ( < N:Nat @ MT > | S:DataSpace , QL ) =
        out ( S:DataSpace , (Q1 , QL) )
            if Q1 := noNoise(occurringTuples(MT)) [owise] .
eq out ( < N:Nat @ MT > | S:DataSpace , QL ) =
        out ( S:DataSpace , QL ) [owise] .
eq out ( S:DataSpace , QL ) = true [owise] .

op ts-is-ok : List{Float} Float -> Bool .
eq ts-is-ok(nil,T) = true .
eq ts-is-ok(0.0 L,T) = ts-is-ok(L,T) .
ceq ts-is-ok(F L,T) = ts-is-ok(L,T)
     if F == T /\
        F > 0.0 .
eq ts-is-ok(F L,T) = false [owise]

op out-new : DataSpace -> Bool .
```

```
  ceq out-new ( S:DataSpace | # T # ) = true
      if N:Nat := count-kind(S:DataSpace) /\
         float(N:Nat) == T .
  eq out-new ( S:DataSpace | # T # ) = false [owise] .

  op count-kind : DataSpace -> Nat .
  eq count-kind(< N':Nat @ MT | tot(N:Nat) > | S:DataSpace) =
          N:Nat + count-kind(S:DataSpace) .
  eq count-kind(S:DataSpace) = 0 [owise] .
endm


mod COLLECTIVE-SORTING is
  pr COLLECTIVE-SORTING-FUNCTIONS .
  pr STANDARD-CARRIER .
  pr NAT .
  pr LIST{Nat} .

  vars F F0 F1 F2 F3 : Float .
  vars N N' N'' N''' N1 N2 N3 Tot Tot' : Nat .
  var NN : [Nat] .
  vars Q Q1 Q2 : TupleType .
  var QQ : [TupleType] .
  vars MT MT1 MT2 MT3 : TupleMSet .
  vars QL : QList .
  vars DS DS' : DataSpace .

  ***  SYNTAX OF ACTIONS AND STATES
  op choose : -> Action .
  op chooseO : -> Action .
  op ttype : -> Action .
  op in1 : -> Action .
  op in2 : -> Action .
  op move : -> Action .

  subsort DataSpace < State .

  *** SEMANTICS
  eq  (init | DS | {N}) ==>  =
              ( choose # 1.0 -> [ [one(N)] | DS | {N}] ) .

  *** CHOOSING OTHER SPACE
  eq ([N1]     | DS | {N})   ==>  =
          ( chooseO # now -> [ ([N1];[one(N - 1)]) | DS | {N} ] ) .
```

```
eq (([N1];[N1])  | DS | {N})  ==>  =
        ( chooseO # now -> [ ([N1];[ N - 1 ])   | DS | {N} ] ) .

*** CHOOSING A TUPLE FROM N1 :
ceq ([N1];[N2]     | < N1 @ MT > | DS ) ==> =
        ( in1 # now -> [ ([N1];[N2];[QQ] | < N1 @ MT > | DS ) ] )
      if QL := occurringTuples(MT) /\
         QQ := get( QL , sample (quantities(QL, MT))) [owise].

*** CHOOSING A TUPLE FROM N2
ceq ([N1];[N2];[Q]   | < N2 @ MT > | DS ) ==> =
        ( in2 # now -> [ ([N1];[N2];[Q];[QQ] | < N2 @ MT > | DS ) ] )
     if QL := occurringTuples(MT) /\
        QQ := get( QL , sample (quantities(QL, MT))) [owise] .

*** MOVING OR DISCARDING
op pred _ : Nat -> Nat .
eq pred 0 = 0 .
eq pred s N = N .

op move : TupleType TupleType -> Bool .
ceq move (Q1,Q2) = true if Q1 =/= Q2 .
eq move (Q1,Q2) = false [owise] .

op noise : TupleType TupleType Nat Bool -> Nat .
ceq noise(Q1,Q2,N,false) = N + 1
        if Q1 == Q2 /\ Q1 =/= ? /\
           Q2 =/= ? /\ Q1 =/= 'noKind /\
           Q2 =/= 'noKind .
ceq noise(Q1,Q2,N,true) = N - 1
        if Q1 =/= Q2 /\ N > 1 /\
           Q1 =/= ? /\ Q2 =/= ? /\
           Q1 =/= 'noKind /\ Q2 =/= 'noKind  .
ceq noise(?,Q2,N,true) = N - 1
           if N > 1 /\ Q2 =/= ? /\ Q2 =/= 'noKind .
ceq noise('noKind,Q2,N,true) = N - 1
           if N > 1 /\ Q2 =/= ? /\ Q2 =/= 'noKind .
ceq noise(Q2,?,N,true) = N
           if Q2 =/= ? /\ Q2 =/= 'noKind .
ceq noise(Q2,'noKind,N,true) = N - 1
           if N > 1 /\ Q2 =/= ? /\ Q2 =/= 'noKind .
eq noise(Q1,Q2,N,B:Bool) = N [owise] .

var Mov : Bool .
```

```
ceq (  [N1];[N2];[Q1];[Q2] | # N3 #
   | < N1 @ (Q2 [ s N ]) | (? [ N'' ]) | MT | tot(Tot) >
   | < N2 @ (Q2 [ N'  ]) | MT1 | tot(Tot') > | DS ) ==> =
          ( move # now -> [ (   init | # s N3 #
   | < N1 @ (Q2 [   N ]) | (? [ N''' ]) | MT |
         tot(updateNum((Q2 [ N ]) | MT)) > |
     < N2 @ (Q2 [ s N']) | MT1 |
           tot(updateNum((Q2 [ s N']) | MT1)) > | DS ) ] )
        if Mov := move(Q1,Q2) /\ Mov == true /\
           N''' := noise(Q1,Q2,N'',Mov) .

ceq (  [N1];[N2];[Q2];[?] | # N3 #
   | < N1 @ (Q2 [ s N ]) | (? [ N'' ]) | MT | tot(Tot) >
   | < N2 @ (Q2 [ N'  ]) | MT1 | tot(Tot') > | DS ) ==> =
           ( move # now -> [(   init | # s N3 #
   | < N1 @ (Q2 [   N ]) | (? [ N''' ]) | MT |
           tot(updateNum((Q2 [ N ]) | MT)) >
   | < N2 @ (Q2 [ s N']) | MT1 |
           tot(updateNum((Q2 [ s N']) | MT1)) > | DS ) ] )
        if Mov := move(Q2,?) /\ Mov == true /\
           N''' := noise(Q2,?,N'',Mov) .

ceq (  [N1];[N2];[?];[Q2] | # N3 #
   | < N1 @ (Q2 [ s N ]) | (? [ N'' ]) | MT | tot(Tot) >
   | < N2 @ (Q2 [ N'  ]) | MT1 | tot(Tot') > | DS ) ==> =
           ( move # now -> [(   init | # s N3 #
   | < N1 @ (Q2 [   N ]) | (? [ N''' ]) | MT |
           tot(updateNum((Q2 [ N ]) | MT)) >
   | < N2 @ (Q2 [ s N']) | MT1 |
           tot(updateNum((Q2 [ s N']) | MT1)) > | DS ) ] )
        if Mov := move(?,Q2) /\ Mov == true /\
           N''' := noise(?,Q2,N'',Mov) .

ceq (  [N1];[N2];[?];[Q2] | # N3 #
   | < N1 @ (Q2 [ 0 ]) | (? [ N'' ]) | MT | tot(Tot) >
   | < N2 @ (Q2 [ N'  ]) | MT1 | tot(Tot') > | DS ) ==> =
           ( move # now -> [(   init | # N3 #
   | < N1 @ (Q2 [   0 ]) | (? [ N''' ]) | MT | tot(Tot) >
   | < N2 @ (Q2 [ N']) | MT1 | tot(Tot') > | DS ) ] )
       if Mov := move(?,Q2) /\ Mov == true /\ N''' := N'' .

ceq (  [N1];[N2];[Q1];['noKind] | # N3 #
   | < N1 @ (Q1 [ s N ]) | (? [ N'' ]) | MT | tot(Tot) >
   | < N2 @ (Q1 [ N'  ]) | MT1 | tot(Tot') > | DS ) ==> =
```

```
            ( move # now -> [(   init | # s N3 #
   | < N1 @ (Q1 [   N ]) | (? [ N''' ]) | MT |
            tot(updateNum((Q1 [ N ]) | MT)) >
   | < N2 @ (Q1 [ s N']) | MT1 |
            tot(updateNum((Q1 [ s N']) | MT1)) > | DS ) ] )
      if Q1 =/= 'noKind /\ N''' := noise(Q1,'noKind,N'',true) .

ceq (  [N1];[N2];[Q1];[Q2] | # N3 #
   | < N1 @ (Q2 [ 0 ]) | (? [ N'' ]) | MT >
   | < N2 @ (Q2 [ N'  ]) | MT1 > | DS ) ==> =
            ( move # now -> [(   init | # N3 #
   | < N1 @ (Q2 [ 0 ]) | (? [ N''' ]) | MT >
   | < N2 @ (Q2 [ N']) | MT1 > | DS ) ] )
      if move(Q1,Q2) /\ N''' := N'' .

ceq (  [N1];[N2];[Q1];[Q2] | < N1 @ (? [ N'' ]) | MT > | DS ) ==> =
            ( move # now -> [ (   init
   | < N1 @ (? [ N''' ]) | MT > | DS ) ] )
      if Mov := move(Q1,Q2) /\
         Mov == false /\ N''' := noise(Q1,Q2,N'',Mov) .

op updateNum : TupleMSet -> Nat .
ceq updateNum((Q2 [ N ]) | MT ) = 1 + updateNum(MT)
      if N > 0 /\ Q2 =/= ? .
eq updateNum(empty) = 0 .
eq updateNum((? [ N ]) | MT ) = 0 + updateNum(MT) .
ceq updateNum((Q2 [ 0 ]) | MT ) = 0 + updateNum(MT) if Q2 =/= ? .


*** TEMP
eq quit( N, DS, F ) = out-new(DS) .

eq temp( init | DS ) = false .
eq temp( DS ) = true [owise] .

*** OBS
subsort State < Observation .

sorts TSView-List TSView .
subsort TSView < TSView-List .

op nilTSList : -> TSView-List .
op _,_ : TSView-List TSView-List -> TSView-List [ctor assoc id: nilTSList] .
```

```
  op ts : Nat Nat Nat -> TSView .
  op t : Nat TSView-List -> Observation .
  op o : State -> Observation .
   eq o ( < N' @ T:TupleMSet | (?[N'']) > | S:State | # N #  ) =
           t(N,getPattern(< N' @ T:TupleMSet | (?[N'']) > | S:State )) .


  eq obs (N:Nat , S:State, F:Float ) = o(S:State) .

  op getPattern : State -> TSView-List .
  eq getPattern(< N' @ T:TupleMSet | (?[N'']) | tot(N) > | S:State) =
           ts(N',N,N''), getPattern(S:State) .
  eq getPattern(S:State) = nilTSList [owise] .

  op SS-4 : -> State .
  eq SS-4 = ( init | {4} |
          < 0 @ ('a[25])|('b[25])|('c[25])|('e[25])|(?[0]) | tot(4)  > |
          < 1 @ ('a[25])|('b[25])|('c[25])|('e[25])|(?[0]) | tot(4)  > |
          < 2 @ ('a[25])|('b[25])|('c[25])|('e[25])|(?[0]) | tot(4)  > |
          < 3 @ ('a[25])|('b[25])|('c[25])|('e[25])|(?[0]) | tot(4)  > |
             # 0 # | # 4.0 # ) .


  op SS-4-local-min : Nat -> State .
  eq SS-4-local-min(N) = ( init | {4} |
          < 0 @ ('a[50])|('b[0])  |('c[0])  |('e[0])  |(?[N]) | tot(1)  > |
          < 1 @ ('a[50])|('b[0])  |('c[0])  |('e[0])  |(?[N]) | tot(1)  > |
          < 2 @ ('a[0]) |('b[100])|('c[100])|('e[0])  |(?[N]) | tot(2)  > |
          < 3 @ ('a[0]) |('b[0])  |('c[0])  |('e[100])|(?[N]) | tot(1)  > |
             # 0 # | # 4.0 # ) .

endm


view COLLECTIVE-SORTING from CARRIER to COLLECTIVE-SORTING is
endv

mod CS-STOCHASTIC-TRACES is
  pr STOCHASTIC-TRACES-ENGINE{COLLECTIVE-SORTING} .
endm

set print format on .

***SAMPLE SIMULATION RUNS
```

```
rewrite series( 20 , [20000 : SS-4-local-min(1) @ 0.0] ) .
rewrite series( 100 , [200000 : SS-4-local-min(2) @ 0.0] ) .
rewrite series( 100 , [20000 : SS-4-local-min(5) @ 0.0] ) .
rewrite series( 100 , [20000 : SS-4-local-min(10) @ 0.0] ) .
rewrite series( 100 , [20000 : SS-4-local-min(20) @ 0.0] ) .
rewrite series( 100 , [20000 : SS-4-local-min(50) @ 0.0] ) .
```

C

Maude Specification of SwarmLinda

```
mod SWARMLINDA-TYPES is
   pr TERMS-SYNTAX .
   pr NAT .
   pr QID .
   pr FLOAT .
   pr LIST{Nat} .

   sort MultiTerm .
   subsort Term < MultiTerm .
   op nilMT : -> MultiTerm [ctor] .
   op _|_ : MultiTerm MultiTerm -> MultiTerm [ctor assoc comm id: nilMT] .

   sorts TS MultiTS .
   subsort TS < MultiTS .
   op nilMTS : -> MultiTS [ctor] .
   op _|_ : MultiTS MultiTS -> MultiTS [ctor assoc comm id: nilMTS] .

   sorts Scent MultiScent .
   subsort Scent < MultiScent .
   op nilS : -> MultiScent [ctor] .
   op _|_ : MultiScent MultiScent -> MultiScent [ctor assoc comm id: nilS] .
   op {_,_} : Term Float -> Scent [ctor] .

   sorts Item MultiItem .
   subsort Item < MultiItem .
   op nilI : -> MultiItem [ctor] .
   op _|_ : MultiItem MultiItem -> MultiItem [ctor assoc comm id: nilI] .
   op [_@_] : Qid MultiTerm -> Item [ctor] .
   op [_@_] : Qid MultiScent -> Item [ctor] .
   op [_@_] : Qid Float -> Item [ctor] .
   op [_@_] : Qid List{Nat} -> Item [ctor] .
```

```
   op <_@_> : Nat MultiItem -> TS [ctor] .

endm


mod SWARMLINDA-FUNCTIONS is
   pr SWARMLINDA-TYPES .
   pr STOCHASTIC-SELECTION-IMPLEMENTATION .
   pr LIST{Float} .

   op similarity : TermList TermList -> Float [comm] .
   vars N1 N2 : Term .
   vars X Y : TermVar .
   vars L1 L2 : TermList .
   ceq similarity((N1(L1)),(N2(L2))) = 0.0 if N1 =/= N2 .
   eq similarity((N1(L1)),(N1(L2))) = 0.6 + similarity(L1,L2) .
   eq similarity((N1(L1)),(N1)) = 0.6 + similarity(L1,nil) .
   eq similarity((N1,L1),(N2,L2)) = similarity(L1,L2) .
   eq similarity(nil,(N1,L1)) = -0.1 + similarity(nil,L1) .
   eq similarity(nil,nil) = 0.0 .

   op f : Term MultiScent Float -> Float .
   var MS : MultiScent .
   var S : Scent .
   vars F F' Tot : Float .
   eq f(N2,({N1,F} | MS),F') = f(N2,MS,(F' + (F * similarity(N1,N2)))) .
   eq f(N2,nilS,F') = F' .

   op addMem2F : Float List{Float} -> Float .
   var Mem : List{Float} .
   eq addMem2F(F,nil) = F .
   eq addMem2F(F,F' Mem) = addMem2F((F + F'),Mem) .

   op calcNeighbsScent : Term List{Nat} MultiTS -> List{Float} .
   var LN : List{Nat} .
   var MTS : MultiTS .
   var TS : TS .
   var MI : MultiItem .
   var N : Nat .
   var T : Term .
   eq calcNeighbsScent(T,(N LN),(< N @ ['scents @ MS] | MI > | MTS)) =
               f(T,MS,0.0)  calcNeighbsScent(T,(LN),(MTS)) .
   eq calcNeighbsScent(T,(nil),(MTS)) = nil .
```

```
op choose : List{Float} -> Nat .
vars L : List{Float} .
ceq choose( L ) = $sample( L , F ) if F := $sum(L) /\ F =/= 0.0 .
eq choose( L ) = one( length ( L ) ) [owise] .

op one : Nat -> Nat .
eq one( N:Nat ) = randrange(N:Nat).

op length : List{Float} -> Nat .
eq length( nil ) = 0 .
eq length( F L ) = s length( L ) .

op chooseNeighbs : List{Nat} List{Float} -> Nat .
eq chooseNeighbs(LN,L) = getEl(LN,choose(L)) .

op isEveryQueueEmpty : MultiTS -> Bool .
var MQ : MultiTerm .
eq isEveryQueueEmpty( < N @ ['queue @ nilMT] | MI > | MTS ) =
                                                isEveryQueueEmpty( MTS ) .
ceq isEveryQueueEmpty( < N @ ['queue @ MQ] | MI > | MTS ) = false
                                                if MQ =/= nilMT .
eq isEveryQueueEmpty( MTS ) = true [owise] .

op updateEntropy : MultiTS -> MultiTS .
eq updateEntropy(nilMTS) = nilMTS .
ceq updateEntropy( < N @ ['scents @ MS] | ['entropy @ F] | MI > | MTS) =
       < N @ ['scents @ MS] | ['entropy @ F'] | MI > | updateEntropy(MTS)
if F' := calcEntropy(MS) .

op calcEntropy : MultiScent -> Float .
ceq calcEntropy(MS) = sp-entropy(L) if L := convert-MS2LF(MS) .

op convert-MS2LF : MultiScent -> List{Float} .
eq convert-MS2LF(nilS) = nil .
eq convert-MS2LF({N1,F} | MS) = F convert-MS2LF(MS) .

op log2 : Float -> Float .
eq log2( F ) = log( F ) / log( 2.0 ) .

op info : Float Float -> Float .
var QT : Float .
eq info( QT , F ) = ( ( F / QT ) * log2( QT / F ) ) .

op entropy : Float List{Float} -> Float .
```

```
   eq entropy( QT , nil ) = ( 0.0 ) .
   ceq entropy( QT , F L ) = info( QT , F ) + entropy( QT , L ) if F > 0.0 .
   eq entropy( QT , F L ) = 0.0 + entropy( QT , L ) [owise] .

   op sp-entropy : List{Float} -> Float .
   eq sp-entropy( F L ) =
              entropy( $sum( F L ) , F L ) / log2( float (length( F L )) ) .

endm


mod SWARMLINDA is
   pr STANDARD-CARRIER .
   pr SWARMLINDA-FUNCTIONS .
   pr NAT .
   pr TERMS-SUBS .

   subsort MultiTS < State .

   op chooseNode : -> Action .
   op choose : -> Action .
   op init : -> Action .
   op simil : Nat -> Action .
   op store : -> Action .
   op move : -> Action .
   op move1 : -> Action .
   op move1-a : -> Action .
   op done : -> Action .

   sort Status .subsort Status < MultiTS .
   op [_] : Qid -> Status [ctor] .
   op [_] : Nat -> Status [ctor] .
   op [_] : Float -> Status [ctor] .
   op [_] : Term -> Status [ctor] .
   op [_,{_}] : Term List{Float} -> Status [ctor] .
   op [_,_] : Term Nat -> Status [ctor] .
   op [_,_] : Term Int -> Status [ctor] .
   op [_,_] : Term Float -> Status [ctor] .


   vars TS' TS : MultiTS .
   vars N N1 TMem MemSz : Nat .
   var MI : MultiItem .
   vars MT SC : MultiTerm .
```

```
vars T T' : Term .
var MS : MultiScent .
vars F F' Steps Pstore K Tot Fe Ke : Float .
var BL : BindList .
var Neighbs LN : List{Nat} .
vars Mem Mem' NeighbsScents L : List{Float} .


op convert-MTS2LN : MultiTS -> List{Nat} .
var S : Status .
eq convert-MTS2LN(nilMTS) = nil .
eq convert-MTS2LN(< N @ MI > | TS) = N convert-MTS2LN(TS) .
eq convert-MTS2LN(S | TS) = convert-MTS2LN(TS) .
op convert-MTS2LF : MultiTS -> List{Float} .
eq convert-MTS2LF(nilMTS) = nil .
eq convert-MTS2LF(< N @ MI > | TS) = 1.0 convert-MTS2LF(TS) .
eq convert-MTS2LF(S | TS) = convert-MTS2LF(TS) .

op chooseNode : MultiTS -> Nat .
ceq chooseNode(TS) = chooseNeighbs(LN,L) if LN := convert-MTS2LN(TS) /\
                                            L := convert-MTS2LF(TS) .

  *** SWARMLINDA SEMANTICS
 ceq  (['calc-entropy] | ['steps,K] | ['mem,TMem] | ['k,Ke] |
               ['f,Fe] | ['chosen,(-1)] | TS ) ==>  =
      ( init # 1.0 -> [['chooseNode] | ['chosen,(-1)] | ['k,Ke] |
               ['f,Fe] | ['steps,K] | ['mem,TMem] | TS'] )
 if TS' := updateEntropy(TS) .

 eq  (['chooseNode] | ['steps,K] | ['chosen,(-1)] | TS ) ==>  =
       ( chooseNode # 1.0 -> [['init] |
                              ['chosen,(chooseNode(TS))] |
                              ['steps,K] | TS ] ) .

 eq  (['init] | ['steps,K] | ['chosen,N] |
                < N @ ['queue @(('out(T))| MT)]| MI > | TS ) ==>  =
   ( choose # 1.0 -> [['choose1] | ['chosen,(N)] |
                       ['steps,K] | [K] | [N] | [('out(T)),{nil}] | TS |
                < N @ ['queue @ MT] | MI >] ) .

 eq  (['init] | ['steps,K] | ['chosen,N] |
                    < N @ ['queue @(nilMT)]| MI > | TS ) ==>  =
    ( init # 1.0 -> [['chooseNode] | ['chosen,(- 1)] | ['steps,K] | TS |
                  < N @ ['queue @ (nilMT)] | MI >] ) .
```

```
ceq  (['choose1] | ['k,Ke] | ['f,Fe] | [K] | [N] | [('out(T)),{Mem}] |
                      < N @ ['scents @ MS] | MI > | TS ) ==>  =
 ( store # Pstore -> [['store] | ['k,K] | ['f,F] | [N] | [('out(T))] |
                      < N @ ['scents @ MS] | MI > | TS] );
 ( move # (1.0 - Pstore) -> [['choose2] | ['k,Ke] | ['f,Fe] | [(K - 1.0)] |
                           [N] | [('out(T)),{Mem F}] |
                      < N @ ['scents @ MS] | MI > | TS] )
if F := f(T,MS,0.0) /\  F' := addMem2F(F,Mem) /\
   Pstore := (F' / (F' + K)) * (F' / (F' + K)) [owise] .

eq  (['choose1] | [0.0] | ['k,Ke] | ['f,Fe] | [N] | [('out(T)),{Mem}] |
                      < N @ ['scents @ MS] | MI > | TS ) ==>  =
  ( store # 1.0 -> [['store] | ['k,0.0] | ['f,- 1.0] | [N] | [('out(T))] |
                      < N @ ['scents @ MS] | MI > | TS] ) .

eq  (['choose2] | ['k,Ke] | ['f,Fe] | [0.0] | [N] | [('out(T)),{Mem}] |
                      < N @ ['scents @ MS] | MI > | TS ) ==>  =
  ( store # 1.0 -> [['store] | ['k,0.0] | ['f,- 1.0] | [N] | [('out(T))] |
                      < N @ ['scents @ MS] | MI > | TS] ) .

ceq  (['choose2] | ['mem,TMem] | [K] | ['steps,Steps] | [N] |
   [('out(T)),{Mem}] | < N @ ['neighbors @ Neighbs] | MI > | TS ) ==>  =
  ( move1 # 1.0 -> [['update] | ['mem,TMem] | [K] | ['steps,Steps] |
   [chooseNeighbs(Neighbs,calcNeighbsScent(T,Neighbs,TS))] |
   [('out(T)),{Mem}] | < N @ ['neighbors @ Neighbs] | MI > | TS] )
if (size(Mem) < TMem) or (size(Mem) == TMem) .

 eq  (['update] | ['mem,TMem] | [K] | ['steps,Steps] |
                     [N] | [('out(T)),{Mem}] |
       < N @ ['neighbors @ Neighbs] | ['mov @ Mov] | MI > | TS ) ==>  =
  ( move1-a # 1.0 -> [['choose1] | ['mem,TMem] | [K] | ['steps,Steps] |
                     [N] | [('out(T)),{Mem}] |
       < N @ ['neighbors @ Neighbs] | ['mov @ (Mov + 1.0)] | MI > | TS] ) .

 ceq  (['choose2] | ['mem,TMem] | [K] | ['steps,Steps] | [N] |
                [('out(T)),{Mem}] |
       < N @ ['neighbors @ Neighbs] | MI > | TS ) ==>  =
  ( move1 # 1.0 -> [['update] | ['mem,TMem] | [K] | ['steps,Steps] |
                [chooseNeighbs(Neighbs,calcNeighbsScent(T,Neighbs,TS))] |
                [('out(T)),{Mem'}] |
       < N @ ['neighbors @ Neighbs] | MI > | TS] )
 if (size(Mem) > TMem) /\ Mem' := tail(Mem) .
```

```
   ceq  (['store] | [N] | ['chosen,N1] | [('out(T))] |
    < N @ ['scents @ {T',K} | MS] | ['tot @ Tot] | MI > | TS ) ==>  =
            ( done # 1.0 -> [['calc-entropy] | ['chosen,(- 1)] |
    < N @  ['scents @ {T',(K + 1.0)} | MS] |
           ['tot @ (Tot + 1.0)] | MI > | TS] )
    if  BL := T // T' /\ BL :: BindList .

   eq quit( N, TS, F ) = isEveryQueueEmpty(TS) .
   eq temp( ['chooseNode] | TS ) = false .
   eq temp( TS ) = true [owise] .

   sort Plottable .
   subsort Plottable < Observation .
   subsort List{Float} < Plottable .

   op nonP : -> Plottable [ctor] .
   op _@_ : Plottable Plottable -> Plottable [ctor assoc comm id: nonP] .

   op o : Float State -> Observation .
   op o : Float -> Observation .
   op o : State -> Observation .

   eq obs (N:Nat , S:State, F:Float ) = o(S:State) .
   var Mov : Float .
   eq o( < N @ ['scents @ MS] | ['entropy @ Tot] | ['mov @ Mov] | MI > |
           TS ) = float(N) getPlottable(MS) Tot Mov @ o( TS ) .
   eq o( TS ) = nonP [owise] .

   op getPlottable : MultiScent -> List{Float} .
   eq getPlottable( {T',(K)} | MS ) = K getPlottable(MS) .
   eq getPlottable( nilS ) = nil .

    ***SAMPLE INSTANCE
    op S-star-2 : -> State .
    eq S-star-2 = (['init] | ['steps,10.0] |
< 0 @ ['queue @(('out('a(v('X)))) | ('out('b(v('X)))) | ('out('a(v('X)))) |
           ('out('b(v('X)))) | ('out('a(v('X)))) | ('out('b(v('X)))) |
           ('out('b(v('X)))) | ('out('a(v('X)))) | ('out('b(v('X))))))]|
           ['tot @ 0.0] | ['scents @ ({'a(v('X)),0.0} | {'b(v('X)),0.0})] |
           ['neighbors @ (1)] > |
< 1 @ ['queue @(('out('a(v('X)))) | ('out('b(v('X)))) | ('out('a(v('X)))) |
            ('out('b(v('X)))) | ('out('a(v('X)))) | ('out('b(v('X)))) |
            ('out('a(v('X)))) | ('out('b(v('X)))) | ('out('a(v('X))))] |
           ['tot @ 0.0] | ['scents @ ({'a(v('X)),0.0} | {'b(v('X)),0.0})] |
```

```
             ['neighbors @ (0 2)] > |
 < 2 @ ['queue @(('out('a(v('X')))) | ('out('b(v('X')))) | ('out('a(v('X')))) |
              ('out('b(v('X')))) | ('out('a(v('X')))) | ('out('b(v('X')))) |
              ('out('b(v('X')))) | ('out('a(v('X')))) | ('out('b(v('X'))))] |
             ['tot @ 0.0] | ['scents @ ({'a(v('X')),0.0} | {'b(v('X')),0.0})] |
             ['neighbors @ (1 3 5)] > |
 < 3 @ ['queue @(('out('a(v('X')))) | ('out('b(v('X')))) | ('out('a(v('X')))) |
               ('out('b(v('X')))) | ('out('a(v('X')))) | ('out('b(v('X')))) |
               ('out('b(v('X')))) | ('out('a(v('X')))) | ('out('b(v('X'))))] |
             ['tot @ 0.0] | ['scents @ ({'a(v('X')),0.0} | {'b(v('X')),0.0})] |
             ['neighbors @ (2 4)] > |
 < 4 @ ['queue @(('out('a(v('X')))) | ('out('b(v('X')))) | ('out('a(v('X')))) |
                ('out('b(v('X')))) | ('out('a(v('X')))) | ('out('b(v('X')))) |
                ('out('b(v('X')))) | ('out('a(v('X')))) | ('out('b(v('X'))))] |
              ['tot @ 0.0] | ['scents @ ({'a(v('X')),0.0} | {'b(v('X')),0.0})] |
              ['neighbors @ (3)] > |
 < 5 @ ['queue @(('out('a(v('X')))) | ('out('b(v('X')))) | ('out('a(v('X')))) |
                ('out('b(v('X')))) | ('out('a(v('X')))) | ('out('b(v('X')))) |
                ('out('b(v('X')))) | ('out('a(v('X')))) | ('out('b(v('X'))))] |
              ['tot @ 0.0] | ['scents @ ({'a(v('X')),0.0} | {'b(v('X')),0.0})] |
              ['neighbors @ (2)] > ) .
endm


view SWARMLINDA from CARRIER to SWARMLINDA is
endv

mod SWARMLINDA-STOCHASTIC-TRACES is
    pr STOCHASTIC-TRACES-ENGINE{SWARMLINDA} .
endm

set print format on .

***SAMPLE SIMULATION RUN
rewrite trace( [10000 : S-star-2 @ 0.0] < empty >) .
```

D
List of Publications

2009

- Matteo Casadei and Mirko Viroli. An experience on probabilistic model checking and stochastic simulation to design self-organizing systems. In *IEEE Congress on Evolutionary Computation, 2009 (CEC 2009).*, 18–21 May 2009

- Mirko Viroli, Matteo Casadei, and Andrea Omicini. A framework for modelling and implementing self-organising coordination. In Sung Y. Shin, Sascha Ossowski, Ronaldo Menezes, and Mirko Viroli, editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, Honolulu, Hawai'i, USA, 8–12 March 2009. ACM

- Matteo Casadei and Andrea Omicini. Situated tuple centres in ReSpecT. In Sung Y. Shin, Sascha Ossowski, Ronaldo Menezes, and Mirko Viroli, editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, Honolulu, Hawai'i, USA, 8–12 March 2009. ACM

- Matteo Casadei and Mirko Viroli. Using probabilistic model checking and simulation for designing self-organizing systems. In Sung Y. Shin, Sascha Ossowski, Ronaldo Menezes, and Mirko Viroli, editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, Honolulu, Hawai'i, USA, 8–12 March 2009. ACM

- Mirko Viroli, Franco Zambonelli, Matteo Casadei, and Sara Montagna. A biochemical metaphor for developing eternally adaptive service ecosystems. In Sung Y. Shin, Sascha Ossowski, Ronaldo Menezes, and Mirko Viroli, editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, Honolulu, Hawai'i, USA, 8–12 March 2009. ACM

- Matteo Casadei, Mirko Viroli, and Luca Gardelli. On the collective sort problem for distributed tuple spaces. *Science of Computer Programming*, 2009. In press

2008

- Matteo Casadei and Mirko Viroli. Applying self-organizing coordination to emergent tuple organization in distributed networks. In Sven Brueckner, Paul Roberson, and Umesh

Bellur, editors, *2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'08)*, pages 213–222, Venice, Italy, 20–24 October 2008. IEEE Computer Society

- Elena Nardini, Matteo Casadei, Andrea Omicini, and Pietro Gaffuri. A conceptual framework for collaborative learning systems based on agent technologies. In *The 2008 International Conference on the Interactive Computer Aided Learning (ICL 2008)*, pages RT–2:3(1–9), Villach, Austria, EU, 24–26 September 2008. Kassel University Press. Electronic Proceedings

- Matteo Casadei, Mirko Viroli, and Marco Santarelli. Collective sort and emergent patterns of tuple distribution in grid-like networks. In *17th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises. Workshop on Coordination Models and Applications (CoMA 2008)*, 23–25 June 2008

- Matteo Casadei and Andrea Omicini. Situating AA ReSpecT for pervasive environment applications. In Lyndon Nixon, Manfred Bortenschlager, Elena Simperl, and Robert Tolksdorf, editors, *17th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises. Workshop on Coordination Models and Applications (CoMA 2008)*, IEEE WETICE 2008, Rome, Italy, 23–25 June 2008

- Matteo Casadei, Andrea Omicini, and Mirko Viroli. Prototyping A&A ReSpecT in Maude. *Electronic Notes in Theoretical Computer Science*, 194(4):93–109, April 2008. 6th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'07), CONCUR'07, Lisbon, Portugal, 8 September 2007. Proceedings

- Luca Gardelli, Mirko Viroli, Matteo Casadei, and Andrea Omicini. Designing self-organising environments with agents and artefacts: A simulation-driven approach. *International Journal of Agent-Oriented Software Engineering*, 2(2):171–195, 2008. Special Issue on Multi-Agent Systems and Simulation

2007

- Matteo Casadei, Ronaldo Menezes, Mirko Viroli, and Robert Tolksdorf. Self-organized over-clustering avoidance in tuple-space systems. In *IEEE Congress on Evolutionary Computation, 2007 (CEC 2007).*, pages 1408–1415. IEEE Computer Society, 25–28 September 2007

- Matteo Casadei, Ronaldo Menezes, Mirko Viroli, and Robert Tolksdorf. Using ant's brood sorting to increase fault tolerance in Linda's tuple distribution mechanism. In Matthias Klusch, Koen Hindriks, Mike Papazoglou, and Mike Sterling, editors, *Cooperative Information Agents XI*, volume 4676 of *LNCS*, pages 255–269. Springer, September 2007. 11th International Workshop on Cooperative Information Agents (CIA 2007), Delf, The Netherland, 19–21 September 2007. Proceedings

- Matteo Casadei, Ronaldo Menezes, Mirko Viroli, and Robert Tolksdorf. A self-organizing approach to tuple distribution in large-scale tuple-space systems. In Davis Hutchison and Randy Katz, editors, *Self-Organizing Systems*, volume 4725 of *LNCS*, pages 146–160. Springer, August 2007. 2nd International Workshop on Self-Organizing Systems (IWSOS 2007), The Lake District, UK, 11–13 September 2007. Proceedings

- Matteo Casadei, Ronaldo Menezes, Robert Tolksdorf, and Mirko Viroli. On the problem of over-clustering in tuple-based coordination systems. In Ozalp Babaoglu and Howard Shrobe, editors, *1st IEEE Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, pages 303–306, Boston, Massachusetts, USA, 9–11 July 2007. IEEE Computer Society

- Matteo Casadei, Andrea Omicini, and Mirko Viroli. Prototyping A&A ReSpecT in Maude. In Carlos Canal, Pascal Poizat, and Mirko Viroli, editors, *6th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'07)*, pages 133–148, CONCUR 2007, Lisbon, Portugal, 8 September 2007. Proceedings

- Matteo Casadei, Luca Gardelli, and Mirko Viroli. Simulating emergent properties of coordination in maude: the collective sort case. *Electronic Notes in Theoretical Computer Science*, 175(2):59–80, June 2007. 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 August 2006. Post-proceedings

- Mirko Viroli, Matteo Casadei, and Luca Gardelli. A self-organising solution to the collective sort problem in distributed tuple spaces. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC 2007)*, pages 354–359, Seoul, Korea, 11–15 March 2007. ACM. Special Track on Coordination Models and Languages

- Luca Gardelli, Mirko Viroli, Matteo Casadei, and Andrea Omicini. Designing self-organising MAS environments: The collective sort case. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for MultiAgent Systems III*, volume 4389 of *LNAI*, pages 254–271. Springer, May 2007. 3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers

2006

- Matteo Casadei, Luca Gardelli, and Mirko Viroli. A case of self-organising environment for mas: the collective sort problem. In Andrea Omicini, Barbara Dunin-Keplicz, and Julian Padget, editors, *4th European Workshop on Multi-Agent Systems (EUMAS 2006)*, number 223 in CEUR Workshop Proceedings, Lisbon, Portugal, 14–15 December 2006. Sun SITE Central Europe, RWTH Aachen University

- Matteo Casadei, Luca Gardelli, and Mirko Viroli. Collective sorting tuple spaces. In Flavio De Paoli, Antonella Di Stefano, Andrea Omicini, and Corrado Santoro, editors, *WOA 2006*

– *Dagli oggetti agli agenti: sistemi grid, p2p e self-**, CEUR Workshop Proceedings, pages 173–180, Catania, Italy, 26–27 September 2006. Sun SITE Central Europe, RWTH Aachen University

- Luca Gardelli, Mirko Viroli, and Matteo Casadei. On engineering self-organizing environments: Stochastic methods for dynamic resource allocation. In *Atti Congresso Annuale AICA 2006*, volume 1, pages 96–101, Cesena, Italy, 21–22 September 2006. Associazione Italiana per l'Informatica ed il Calcolo Automatico, Alinea Editrice, Firenze, Italy

- Antonio Natali, Antonio Del Cinque, and Matteo Casadei. L' uso dei web service nella catena del valore della logistica integrata. In *Atti Congresso Annuale AICA 2006*, volume 2, pages 461–471, Cesena, Italy, 21–22 September 2006. Associazione Italiana per l'Informatica ed il Calcolo Automatico, Alinea Editrice, Firenze, Italy

- Matteo Casadei, Luca Gardelli, and Mirko Viroli. Simulating emergent properties of coordination in maude: the collective sort case. In Carlos Canal and Mirko Viroli, editors, *5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06)*, pages 57–75, CONCUR 2006, Bonn, Germany, 31 August 2006. Proceedings

- Luca Gardelli, Mirko Viroli, and Matteo Casadei. On engineering self-organizing environments: Stochastic methods for dynamic resource allocation. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *3rd International Workshop "Environments for Multi-Agent Systems" (E4MAS 2006)*, pages 96–101, AAMAS 2006, Hakodate, Japan, 8 May 2006

Bibliography

[AAC+00] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Jr. Thomas F. Knight, Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Amorphous computing. *Commun. ACM*, 43(5):74–82, 2000.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[Ash47] William Ross Ashby. Principles of self-organizing dynamic systems. *Journal of General Psychology*, 37:125–128, 1947.

[BA99] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286:509–512, October 1999.

[Bal00] Sadoun Balqies. Applied system simulation: a review study. *Inf. Sci. Inf. Comput. Sci.*, 124(1-4):173–192, 2000.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[BDF+07] M. Balazinska, A. Deshpande, M.J. Franklin, P.B. Gibbons, J. Gray, S. Nath, M. Hansen, M. Liebhold, A. Szalay, and V. Tao. Data management in the worldwide sensor web. *Pervasive Computing, IEEE*, 6(2):30–40, April-June 2007.

[BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Santa Fe Institute Studies in the Sciences of Complexity. Oxford University Press, 198 Madison Avenue, New York, NY 10016, USA, 1999.

[BGLZ04] Mario Bravetti, Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. Probabilistic and prioritized data retrieval in the Linda coordination model. In *Coordination Models and Languages*, volume 2949 of *LNCS*, pages 55–70. Springer, 2004.

[BGLZ05] Mario Bravetti, Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. Quantitative information in the tuple space coordination model. *Theoretical Computer Science*, 346(1):28–57, 2005.

[BGP06] Carole Bernon, Marie Pierre Gleizes, and Gauthier Picard. Enhancing self-organising emergent systems design with simulation. In Gregory M. P. O'Hare, Alessandro Ricci, Michael J. O'Grady, and Oguz Dikenelli, editors, *ESAW*, volume 4457 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2006.

[BLM96] Jean-Pierre Bonâtre and Daniel Le Métayer. Gamma and the chemical reaction model: Ten years after. In *Coordination Programming*, pages 3–41. Imperial College Press London, UK, 1996.

[Cam91] Scott Camazine. Self-organizing pattern formation on the combs of honey bee colonies. *Behavioral Ecology and Sociobiology*, 28(1):61–76, January 1991.

[CDE+05] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual*. Department of Computer Science University of Illinois at Urbana-Champaign, version 2.2 edition, December 2005. Version 2.2 is available online at http://maude.cs.uiuc.edu.

[CDF+01] Scott Camazine, Jean-Louis Deneubourg, Nigel R. Franks, James Sneyd, Guy Theraulaz, and Eric Bonabeau. *Self-Organization in Biological Systems*. Princeton Studies in Complexity. Princeton University Press, 41 William Street, Princeton, NJ 08540, USA, 2001.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000.

[CGV06a] Matteo Casadei, Luca Gardelli, and Mirko Viroli. A case of self-organising environment for mas: the collective sort problem. In Andrea Omicini, Barbara Dunin-Keplicz, and Julian Padget, editors, *4th European Workshop on Multi-Agent Systems (EUMAS 2006)*, number 223 in CEUR Workshop Proceedings, Lisbon, Portugal, 14–15 December 2006. Sun SITE Central Europe, RWTH Aachen University.

[CGV06b] Matteo Casadei, Luca Gardelli, and Mirko Viroli. Collective sorting tuple spaces. In Flavio De Paoli, Antonella Di Stefano, Andrea Omicini, and Corrado Santoro, editors, *WOA 2006 – Dagli oggetti agli agenti: sistemi grid, p2p e self-**, CEUR Workshop Proceedings, pages 173–180, Catania, Italy, 26–27 September 2006. Sun SITE Central Europe, RWTH Aachen University.

[CGV06c] Matteo Casadei, Luca Gardelli, and Mirko Viroli. Simulating emergent properties of coordination in maude: the collective sort case. In Carlos Canal and Mirko Viroli, editors, *5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06)*, pages 57–75, CONCUR 2006, Bonn, Germany, 31 August 2006. Proceedings.

[CGV07] Matteo Casadei, Luca Gardelli, and Mirko Viroli. Simulating emergent properties of coordination in maude: the collective sort case. *Electronic Notes in Theoretical Computer Science*, 175(2):59–80, June 2007. 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 August 2006. Post-proceedings.

[CLZ98] Antonio Corradi, Letizia Leonardi, and Franco Zambonelli. Strategies and protocols for highly parallel Linda servers. *Software Practice and Experience*, 28(14):1493–1517, December 1998.

[CMP07] Pietro Ciciriello, Luca Mottola, and Gian Pietro Picco. Efficient routing from multiple sources to multiple sinks in wireless sensor networks. In *Wireless Sensor Networks*, volume 4486 of *LNCS (Tutorial Volume)*, pages 34–50. Springer, 2007.

[CMTV07] Matteo Casadei, Ronaldo Menezes, Robert Tolksdorf, and Mirko Viroli. On the problem of over-clustering in tuple-based coordination systems. In Ozalp Babaoglu and Howard Shrobe, editors, *1st IEEE Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, pages 303–306, Boston, Massachusetts, USA, 9–11 July 2007. IEEE Computer Society.

[CMVT07a] Matteo Casadei, Ronaldo Menezes, Mirko Viroli, and Robert Tolksdorf. Self-organized over-clustering avoidance in tuple-space systems. In *IEEE Congress on Evolutionary Computation, 2007 (CEC 2007).*, pages 1408–1415. IEEE Computer Society, 25–28 September 2007.

[CMVT07b] Matteo Casadei, Ronaldo Menezes, Mirko Viroli, and Robert Tolksdorf. A self-organizing approach to tuple distribution in large-scale tuple-space systems. In Davis Hutchison and Randy Katz, editors, *Self-Organizing Systems*, volume 4725 of *LNCS*, pages 146–160. Springer, August 2007. 2nd International Workshop on Self-Organizing Systems (IWSOS 2007), The Lake District, UK, 11–13 September 2007. Proceedings.

[CMVT07c] Matteo Casadei, Ronaldo Menezes, Mirko Viroli, and Robert Tolksdorf. Using ant's brood sorting to increase fault tolerance in Linda's tuple distribution mechanism. In Matthias Klusch, Koen Hindriks, Mike Papazoglou, and Mike Sterling, editors, *Co-operative Information Agents XI*, volume 4676 of *LNCS*, pages 255–269. Springer, September 2007. 11th International Workshop on Cooperative Information Agents (CIA 2007), Delf, The Netherland, 19–21 September 2007. Proceedings.

[CO08] Matteo Casadei and Andrea Omicini. Situating AA ReSpecT for pervasive environment applications. In Lyndon Nixon, Manfred Bortenschlager, Elena Simperl, and Robert Tolksdorf, editors, *17th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises. Workshop on Coordination Models and Applications (CoMA 2008)*, IEEE WETICE 2008, Rome, Italy, 23–25 June 2008.

[CO09] Matteo Casadei and Andrea Omicini. Situated tuple centres in ReSpecT. In Sung Y. Shin, Sascha Ossowski, Ronaldo Menezes, and Mirko Viroli, editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, Honolulu, Hawai'i, USA, 8–12 March 2009. ACM.

[COV07] Matteo Casadei, Andrea Omicini, and Mirko Viroli. Prototyping A&A ReSpecT in Maude. In Carlos Canal, Pascal Poizat, and Mirko Viroli, editors, *6th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'07)*, pages 133–148, CONCUR 2007, Lisbon, Portugal, 8 September 2007. Proceedings.

[COV08] Matteo Casadei, Andrea Omicini, and Mirko Viroli. Prototyping A&A ReSpecT in Maude. *Electronic Notes in Theoretical Computer Science*, 194(4):93–109, April 2008. 6th International Workshop on Foundations of Coordination Languages and

Software Architectures (FOCLASA'07), CONCUR'07, Lisbon, Portugal, 8 September 2007. Proceedings.

[CV08] Matteo Casadei and Mirko Viroli. Applying self-organizing coordination to emergent tuple organization in distributed networks. In Sven Brueckner, Paul Roberson, and Umesh Bellur, editors, *2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'08)*, pages 213–222, Venice, Italy, 20–24 October 2008. IEEE Computer Society.

[CV09a] Matteo Casadei and Mirko Viroli. An experience on probabilistic model checking and stochastic simulation to design self-organizing systems. In *IEEE Congress on Evolutionary Computation, 2009 (CEC 2009).*, 18–21 May 2009.

[CV09b] Matteo Casadei and Mirko Viroli. Using probabilistic model checking and simulation for designing self-organizing systems. In Sung Y. Shin, Sascha Ossowski, Ronaldo Menezes, and Mirko Viroli, editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, Honolulu, Hawai'i, USA, 8–12 March 2009. ACM.

[CVG09] Matteo Casadei, Mirko Viroli, and Luca Gardelli. On the collective sort problem for distributed tuple spaces. *Science of Computer Programming*, 2009. In press.

[CVS08] Matteo Casadei, Mirko Viroli, and Marco Santarelli. Collective sort and emergent patterns of tuple distribution in grid-like networks. In *17th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises. Workshop on Coordination Models and Applications (CoMA 2008)*, 23–25 June 2008.

[DDF+06] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, 2006.

[Del02] Giorgio Delzanno. An overview of msr(c): A clp-based framework for the symbolic verification of parameterized concurrent systems. *Electronic Notes in Theoretical Computer Science*, 76, 2002.

[Des37] René Descartes. *A Discourse of a Method for the Well Guiding of Reason and the Discovery of Truth in the Sciences.* 1637. Available online at http://www.gutenberg.org/.

[DGF+91] J.L. Deneubourg, S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, and L. Chrétien. The dynamics of collective sorting: Robot-like ants and ant-like robots. In Jean-Arcady Meyer and Stewart W. Wilson, editors, *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 356–363. MIT Press, Cambridge, MA 02142, USA, February 1991.

[DNLM05] Rocco De Nicola, Diego Latella, and Mieke Massink. Formal modeling and quantitative analysis of KLAIM-based mobile systems. In *2005 ACM Symposium on Applied Computing (SAC 2005)*, pages 428–435. ACM Press, 2005.

[DPHW05] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Continuous-time probabilistic KLAIM. *Electronic Notes in Theoretical Computer Science*, 128(5):27–38, May 2005.

[DTM+05] Marco Dorigo, Elio Tuci, Francesco Mondada, Stefano Nolfi, Jean-Louis Deneubourg, Dario Floreano, and Luca M. Gambardella. The SWARM-BOTS project. *Künstliche Intelligenz*, 4/05:32–35, 2005. Also available at http://www.swarm-bots.org as IRIDIA Technical Report No. TR/IRIDIA/2005-018.

[DWHS05] Tom De Wolf, Tom Holvoet, and Giovanni Samaey. Development of self-organising emergent applications with simulation-based numerical analysis. In *Engineering Self-Organising Systems*, pages 138–152, 2005.

[FSF92] Nigel R. Franks and Ana B. Sendova-Franks. Brood sorting by ants: distributing the workload over the work-surface. *Behavioral Ecology and Sociobiology*, 30(2):109–123, March 1992.

[Gar08] Luca Gardelli. *Engineering Self-organising Systems with the Multiagent Paradigm.* PhD thesis, Alma Mater Studiorum-Universitá di Bologna, 2008.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, 1992.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[Gel89] David Gelernter. Multiple tuple spaces in Linda. In *Parallel Architectures and Languages Europe (PARLE'89)*, volume II: Parallel Languages, pages 20–27. Springer-Verlag, 1989.

[Gil77] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[GVC06a] Luca Gardelli, Mirko Viroli, and Matteo Casadei. On engineering self-organizing environments: Stochastic methods for dynamic resource allocation. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *3rd International Workshop "Environments for Multi-Agent Systems" (E4MAS 2006)*, pages 96–101, AAMAS 2006, Hakodate, Japan, 8 May 2006.

[GVC06b] Luca Gardelli, Mirko Viroli, and Matteo Casadei. On engineering self-organizing environments: Stochastic methods for dynamic resource allocation. In *Atti Congresso Annuale AICA 2006*, volume 1, pages 96–101, Cesena, Italy, 21–22 September 2006. Associazione Italiana per l'Informatica ed il Calcolo Automatico, Alinea Editrice, Firenze, Italy.

[GVCO07] Luca Gardelli, Mirko Viroli, Matteo Casadei, and Andrea Omicini. Designing self-organising MAS environments: The collective sort case. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for MultiAgent Systems III*, volume 4389 of *LNAI*, pages 254–271. Springer, May 2007. 3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers.

[GVCO08] Luca Gardelli, Mirko Viroli, Matteo Casadei, and Andrea Omicini. Designing self-organising environments with agents and artefacts: A simulation-driven approach. *International Journal of Agent-Oriented Software Engineering*, 2(2):171–195, 2008. Special Issue on Multi-Agent Systems and Simulation.

[GVO06] Luca Gardelli, Mirko Viroli, and Andrea Omicini. On the role of simulations in engineering self-organising MAS: The case of an intrusion detection system in TuCSoN. In Sven A. Brueckner, Giovanna Di Marzo Serugendo, David Hales, and Franco Zambonelli, editors, *Engineering Self-Organising Systems*, volume 3910 of *LNAI*, pages 153–168. Springer, 2006. 3rd International Workshop (ESOA 2005), Utrecht, The Netherlands, 26 July 2005. Revised Selected Papers.

[GVO08] Luca Gardelli, Mirko Viroli, and Andrea Omicini. Combining simulation and formal tools for developing self-organizing MAS. In Danny Weyns and Adelinde M. Uhrmacher, editors, *Agents, Simulation and Applications*, pages (5)1–34. Taylor & Francis, 2008.

[HJ94] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:512–535, 1994.

[HLMP04] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*. Springer, 2004.

[HM99] Owen Holland and Chris Melhuis. Stigmergy, self-organization, and sorting in collective robotics. *Artificial Life*, 5(2):173–202, 1999.

[JMF99] A.K. Jain, M.N. Murty, and P.J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, September 1999.

[KE01] J. Kennedy and R. C. Eberhart. *Swarm Intelligence*. Morgan Kaufmann, 2001.

[KEW02] Bhaskar Krishnamachari, Deborah Estrin, and Stephen B. Wicker. The impact of data aggregation in wireless sensor networks. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 575–578, Washington, DC, USA, 2002. IEEE Computer Society.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[KNP04] Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic symbolic model checking with prism: a hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128–142, 2004.

[KNP07] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007.

[Lew04] George Henry Lewes. *Problems of Life and Mind*. Kessinger Publishing, 2004.

[LF94] Erik D. Lumer and Baldo Faieta. Diversity and adaptation in populations of clustering ants. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adpative Behavior*, pages 501–508. MIT Press, Cambridge, MA 02142, USA, August 1994.

[May93] Anneliese von Mayrhauser. The role of simulation in software engineering experimentation. In *Proceedings of the International Workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions*, pages 177–179, London, UK, 1993. Springer-Verlag.

[MHH98] Chris Melhuish, Owen Holland, and Steve Hoddell. Collective sorting and segregation in robots with minimal sensing. In Rolf Pfeifer, Bruce Blumberg, Jean-Arcady Meyer, and Stewart W. Wilson, editors, *From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior*, pages 465–470. MIT Press, Cambridge, MA 02142, USA, 1998.

[MMTZ06] Marco Mamei, Ronaldo Menezes, Robert Tolksdorf, and Franco Zambonelli. Case studies for self-organization in computer science. *Journal of Systems Architecture*, 52(8-9):443–460, 2006.

[MOM02] Narciso Martí-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[MSFS+06] Chris Melhuish, Ana B. Sendova-Franks, Sam Scholes, Ian Horsfield, and Fred Welsby. Ant-inspired sorting by robots: the importance of initial clustering. *Journal of the Royal Society Interface*, 3(7):235–242, April 2006.

[MT03] Ronaldo Menezes and Robert Tolksdorf. A new approach to scalable Linda-systems based on swarms. In *Symposium on Applied Computing (SAC'03)*, pages 375–379, New York, NY, USA, March 2003. ACM Press.

[MT04] Ronaldo Menezes and Robert Tolksdorf. Adaptiveness in Linda-based coordination models. In *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*, volume 2977 of *LNAI*, pages 212–232. Springer, 2004.

[MZ04] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *PerCom 2004*, pages 263– 273. IEEE, March 2004.

[MZL03] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Tuples on the air: A middleware for context-aware computing in dynamic networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 342. IEEE Computer Society, 2003.

[NCOG08] Elena Nardini, Matteo Casadei, Andrea Omicini, and Pietro Gaffuri. A conceptual framework for collaborative learning systems based on agent technologies. In *The 2008 International Conference on the Interactive Computer Aided Learning (ICL 2008)*, pages RT–2:3(1–9), Villach, Austria, EU, 24–26 September 2008. Kassel University Press. Electronic Proceedings.

[NDCC06] Antonio Natali, Antonio Del Cinque, and Matteo Casadei. L' uso dei web service nella catena del valore della logistica integrata. In *Atti Congresso Annuale AICA 2006*, volume 2, pages 461–471, Cesena, Italy, 21–22 September 2006. Associazione Italiana per l'Informatica ed il Calcolo Automatico, Alinea Editrice, Firenze, Italy.

[NPL75] John S. Nicolis, E. Protonotarios, and E. Lianos. Some views on the role of noise in "self"-organizing systems. *Biological Cybernetics*, 17(4):183–193, 1975.

[OD01] Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, November 2001.

[OM06] Sascha Ossowski and Ronaldo Menezes. On coordination and its significance to distributed and multi-agent systems. *Concurrency and Computation: Practice and Experience*, 18(4):359–370, 2006.

[Omi07] Andrea Omicini. Formal ReSpecT in the A&A perspective. *Electronic Notes in Theoretical Computer Science*, 175(2):97–117, June 2007. 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 August 2006. Post-proceedings.

[ORV05] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Time-aware coordination in ReSpecT. In Jean-Marie Jacquet and Gian Pietro Picco, editors, *Coordination Models and Languages*, volume 3454 of *LNCS*, pages 268–282. Springer-Verlag, April 2005.

[ORV07] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Timed environment for Web agents. *Web Intelligence and Agent Systems*, 5(2):161–175, August 2007.

[ORV08] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3), December 2008. Special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems.

[OZ99] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Journal of Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999.

[Par97] H. Van Dyke Parunak. "go to the ant": Engineering principles from natural agent systems. *Annals of Operation Research*, 75:69–101, 1997.

[PBS02] H. Van Dyke Parunak, Sven Brueckner, and John Sauter. Digital pheromone mechanisms for coordination of unmanned vehicles. In *1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, pages 449–450. ACM Press, 2002.

[Phi06] Andrew Phillips. The Stochastic Pi Machine (SPiM), 2006. Version 0.042 available online at http://www.doc.ic.ac.uk/~anp/spim/.

[Plo91] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, AArhus University, Denmark, 1991.

[Pri95] Corrado Priami. Stochastic pi-calculus. *The Computer Journal*, 38(7):578–589, 1995.

[PRI07] PRISM. PRISM: Probabilistic symbolic model checker, October 2007. URL http://www.prismmodelchecker.org/.

[PS97] I. Prigogine and I. Steingers. *The End of Certainty: Time, Chaos, and the New Laws of Nature*. Free Press, 1997.

[rep06] Recursive porous agent simulation toolkit (repast), 2006. Available online at http://repast.sourceforge.net/.

[Res97] Mitchel Resnick. *Turtles, termites, and traffic jams: explorations in massively parallel microworlds*. MIT Press, Cambridge, Massachusetts 02142, USA, January 1997.

[RKNP04] J.J. M.M. Rutten, Marta Kwiatkowska, Gethin Norman, and David Parker. *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems*, volume 23 of *CRM Monograph*. American Mathematical Society, 201 Charles Street, Providence, Rhode Island 02904-2294, United States of America, 2004.

[sas07] *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, Boston, MA, USA, July 9-11, 2007*. IEEE Computer Society, 2007.

[sas08] *Proceedings of the Second International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2008, Venice, Italy, October 20-24, 2008*. IEEE Computer Society, 2008.

[SD03] Eugene Spafford and Richard DeMillo. Grand research challenges in information systems. Technical report, Computing Research Association, 2003. http://www.cra.org/reports/gc.systems.pdf.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27, 1948.

[Str01] S. H. Strogatz. Exploring complex networks. *Nature*, 410(6825):268–276, March 2001.

[swa06] Swarm, 2006. Available online at http://www.swarm.org/.

[TM03] Robert Tolksdorf and Ronaldo Menezes. Using swarm intelligence in linda systems. In Andrea Omicini, Paolo Petta, and Jeremy Pitt, editors, *Proceedings of the Fourth International Workshop Engineering Societies in the Agents World (ESAW'03)*, volume 3071 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, October 2003.

[Tol98] Robert Tolksdorf. Laura — A service-based coordination language. *Science of Computer Programming*, 31(2–3):359–381, July 1998.

[UK 04] UK Computing Research Committee. Grand challenges in computer research. Technical report, British Computing Society, 2004. http://www.bcs.org/upload/pdf/gcresearch.pdf.

[VCG07] Mirko Viroli, Matteo Casadei, and Luca Gardelli. A self-organising solution to the collective sort problem in distributed tuple spaces. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC 2007)*, pages 354–359, Seoul, Korea, 11–15 March 2007. ACM. Special Track on Coordination Models and Languages.

[VCO09] Mirko Viroli, Matteo Casadei, and Andrea Omicini. A framework for modelling and implementing self-organising coordination. In Sung Y. Shin, Sascha Ossowski, Ronaldo Menezes, and Mirko Viroli, editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, Honolulu, Hawai'i, USA, 8–12 March 2009. ACM.

[VZCM09] Mirko Viroli, Franco Zambonelli, Matteo Casadei, and Sara Montagna. A biochemical metaphor for developing eternally adaptive service ecosystems. In Sung Y. Shin, Sascha Ossowski, Ronaldo Menezes, and Mirko Viroli, editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, Honolulu, Hawai'i, USA, 8–12 March 2009. ACM.

[Wil08] U. Wilensky. *NetLogo 1.1 User Manual*. Northwestern University, 2008.

[WMLF98] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998. Special Issue on Java Technology.

[Yan97] Bar-Yam Yaneer. *Dynamics of Complex Systems*. Studies in Nonlinearity. Westview Press, 1997.

[YLB08] K. Yuen, Ben Liang, and Li Baochun. A distributed framework for correlated data gathering in sensor networks. *Vehicular Technology, IEEE Transactions on*, 57(1):578–593, Jan. 2008.

[YN08] Daniel Yamins and Radhika Nagpal. Automated global-to-local programming in 1-d spatial multi-agent systems. In *7th International Joint Conference on Agents and Multi-Agent Systems (AAMAS-08)*, pages 615–622, Estoril, Portugal, 12–16May 2008. IFAAMAS.

[ZP03] Franco Zambonelli and H. Van Dyke Parunak. Towards a paradigm change in computer science and software engineering: A synthesis. *The Knowledge Engineering Review*, 18(4):329–342, December 2003.