ALMA MATER STUDIORUM-Università di Bologna
DEIS - Dipartimento di Elettronica, Informatica e Sistemistica, sede di Cesena
Dottorato di Ricerca in Ingegneria Elettronica, Informatica e delle Telecomunicazioni
Ciclo XX

Ingegneria di Sistemi Auto-organizzanti con il Paradigma Multiagente

Autore

Ing. Luca Gardelli

Coordinatore

Prof. Ing. Paolo Bassi

Relatore

Prof. Ing. Antonio Natali

Correlatori

Prof. Ing. Andrea Omicini
Dott. Ing. Mirko Viroli

Anno Accademico 2007/2008
Settore Scientifico Disciplinare: ING-INF/05

Abstract

Self-organisation is increasingly being regarded as an effective approach to tackle modern systems complexity. The self-organisation approach allows the development of systems exhibiting complex dynamics and adapting to environmental perturbations without requiring a complete knowledge of the future surrounding conditions.

However, the development of self-organising systems (SOS) is driven by different principles with respect to traditional software engineering. For instance, engineers typically design systems combining smaller elements where the composition rules depend on the reference paradigm, but typically produce predictable results. Conversely, SOS display non-linear dynamics, which can hardly be captured by deterministic models, and, although robust with respect to external perturbations, are quite sensitive to changes on inner working parameters.

In this thesis, we describe methodological aspects concerning the early-design stage of SOS built relying on the Multiagent paradigm: in particular, we refer to the A&A metamodel, where MAS are composed by agents and *artefacts*, i.e. environmental resources. Then, we describe an architectural pattern that has been extracted from a recurrent solution in designing self-organising systems: this pattern is based on a MAS environment formed by artefacts, modelling non-proactive resources, and environmental agents acting on artefacts so as to enable self-organising mechanisms. In this context, we propose a scientific approach for the early design stage of the engineering of self-organising systems: the process is an iterative one and each cycle is articulated in four stages, modelling, simulation, formal verification, and tuning. During the modelling phase we mainly rely on the existence of a self-organising strategy observed in Nature and, hopefully encoded as a design pattern. Simulations of an abstract system model are used to drive design choices until the required quality properties are obtained, thus providing guarantees that the subsequent design steps would lead to a correct implementation. However, system analysis exclusively based on simulation results does not provide sound guarantees for the engineering of complex systems: to this purpose, we envision the application of formal verification techniques, specifically *model checking*, in order to exactly characterise the system behaviours. During the tuning stage parameters are tweaked in order to meet the target global dynamics and feasibility constraints.

In order to evaluate the methodology, we analysed several systems: in this thesis, we only describe three of them, i.e. the most representative ones for each of the three years of PhD course. We analyse each case study using the presented method, and describe the exploited formal tools and techniques.

Contents

Italian Summary - Introduzione in Italiano

Contesto di Ricerca e Motivazioni

L'auto-organizzazione costituisce un approccio efficiente per affrontare la complessità dei sistemi moderni, come testimoniato dal crescente interesse della comunità scientifica. Un approccio auto-organizzante permette lo sviluppo di sistemi che esibiscono dinamiche complesse e che si adattano alle perturbazioni ambientali senza richiedere una conoscenza completa delle condizioni circostanti. Un sistema sviluppato seguendo i principi dell'auto-organizzazione produce pattern e dinamiche globali attraverso l'interazione locale dei suoi componenti [SFH+04, MMTZ06]. Molti sistemi biologici possono essere modellati efficacemente utilizzando un approccio auto-organizzante: esempi noti includono la ricerca di cibo e l'ordinamento delle larve nelle colonie di formiche, la costruzione di nidi nelle colonie di termiti, e pattern geometrici negli alveari di api [BDT99, CDF+01, SB06]. I meccanismi auto-organizzanti osservati in Natura hanno ispirato lo sviluppo di molti sistemi artificiali, ad esempio, per la coordinazione decentralizzata di veicoli autonomi (AGV) [WSHL05, PBS05], per la riduzione della congestione nelle reti a commutazione di pacchetto [SA94], per la pianificazione delle lavorazioni e il controllo della pittura di veicoli [CS04], infrastrutture peer-to-peer auto-organizzanti [BMM02], e localizzazione basata su feromone digitale supportato da una rete di tag RFID [MZ07]. Inoltre, i principi dell'auto-organizzazione sono attualmente al centro di progetti di ricerca che potrebbero avere rilevanza industriale in un futuro prossimo: esempi notevoli includono Amorphous Computing [AAC+00], Autonomic Computing [Hor01, KC03] e NASA Swarm Robotics [RHTR06].

Lo sviluppo di sistemi auto-organizzanti (SOS) è guidato da principi diversi rispetto all'ingegneria tradizionale. Tipicamente, gli ingegneri progettano sistemi come composizione di elementi più semplici, sia nel caso di astrazioni software che di dispositivi fisici, dove le regole di composizione dipendono dal paradigma di riferimento, ma che comunque producono risultati predicibili. Al contrario, i sistemi auto-organizzanti mostrano dinamiche non-lineari, le quali possono essere difficilmente catturate da modelli deterministici, e che sebbene robusti nei confronti di perturbazioni esterne, sono piuttosto sensibili ai cambiamenti di parametri interni. In particolare, l'ingegneria di sistemi auto-organizzanti pone due grandi sfide [GVO08]: Come dobbiamo progettare le singole entità in modo tale da produrre il comportamento globale desiderato? Inoltre, come possiamo fornire garanzie sull'emergenza di pattern

specifici? Sebbene l'esistenza di questi problemi sia generalmente riconosciuta, la comunità scientifica non ha ancora fornito adeguati strumenti di supporto all'ingegneria di SOS—ad eccezione di alcune esplorazioni nel contesto dei sistemi Multiagente (MAS) [DW07, BCGP04, BCD$^+$06].

Panoramica e contributi

Mentre la simulazione è sfruttata efficacemente nell'analisi di sistemi complessi, le sue potenzialità nell'ingegneria del software sono tipicamente sottovalutate [Tic98]. Sebbene ci siano esempi dell'uso di simulazione nello sviluppo di software, questi approcci coinvolgono la simulazione dopo che lo sviluppo del software è già stato terminato: in sostanza le simulazioni sono impiegate per fornire una caratterizzazione statistica delle prestazioni. In questa tesi, si vuole promuovere l'utilizzo di tecniche di simulazione fin dai primi stadi del sviluppo software: in questa direzione di ricerca, esiste un insieme molto ridotto di contributi [vM93, Uhr02, DWHS06, BGP07]. La simulazione ci permette di osservare l'andamento qualitativo delle dinamiche di un sistema e di ottenere un primo insieme di parametri operativi primi di implementare il sistema. Se specifichiamo il modello utilizzando linguaggi formali possiamo eseguire ulteriori analisi utilizzando strumenti di verifica formale come il model checking [CGL94, RKNP04].

In particolare, in questa tesi, consideriamo aspetti metodologici riguardanti la fase preliminare di progettazione di sistemi auto-organizzanti costruiti utilizzando il paradigma multiagente: nel caso specifico, faremo riferimento al metamodello ad agenti e artefatti (A&A) [RVO06], dove gli artefatti rappresentano le risorse ambientali. Quindi, descriviamo un pattern architetturale che è stato sintetizzato da soluzioni ricorrenti nella progettazione di sistemi auto-organizzanti [GVO07a]: questo pattern si basa su un ambiente MAS formato da artefatti, che modellano risorse non proattive, e agenti ambientali che agendo sugli artefatti permettono la realizzazione di meccanismi auto-organizzanti. In questo contesto, proponiamo un approccio scientifico per condurre la fase preliminare della progettazione di sistemi auto-organizzanti: in particolare, l'approccio è articolato in quattro fasi, modellazione, simulazione, verifica formale e regolazione dei parametri. Durante la fase di modellazione si fa riferimento all'esistenza di strategie auto-organizzanti osservate in natura ed eventualmente codificate come pattern di progettazione [BCD$^+$06, GVO07a, DWH07]. Le simulazioni di un sistema astratto vengono utilizzate per guidare le scelte di progettazione fino a che si ottengono le proprietà richieste: in questo modo aumenta la probabilità che gli ulteriori passi porteranno ad una implementazione corretta.

L'analisi di sistemi esclusivamente basata sui risultati di simulazioni non fornisce solide garanzie per l'ingegneria di sistemi complessi. A questo scopo, valutiamo l'uso di tecniche di verifica formale: in modo particolare *model checking* [CGL94] ci permette una caratterizzazione precisa dei comportamenti in esame. Data la specifica formale, un model checker probabilistico determina se una proprietà è verificata o meno oppure l'effettivo valore di probabilità: le proprietà sono specificate utilizzando differenti versioni di logiche temporali, in accordo con il tipo di modello, ad esempio, probabilistico, stocastico oppure nondeterministico [RKNP04]. Sfortunatamente, l'applicabilità di tecniche di model checking è limitata dal problema dell'*esplosione dello spazio degli stati* [CGL94]:

nonostante ciò, in quei casi, il model checking può comunque essere utilizzato per validare i risultati delle simulazioni su piccole istanze del problema.

Dalla sua prima formulazione [GVO05c], il metodo è stato enormemente evoluto grazie all'esplorazione di diversi sistemi. In questa tesi, descriviamo tre casi di studio che riteniamo rappresentativi per ognuno dei tre anni del corso di dottorato: in particolare, ogni caso di studio è stato analizzato utilizzando uno strumento differente.

Detection of Anomalous Behaviour Sebbene un insieme di azioni siano sicure quando eseguite individualmente su un sistema, combinazioni di queste azioni possono creare effetti dannosi. Questo caso di studio è il riferimento per le pubblicazioni relative al primo anno di dottorato, ad esempio [GVO06a, GVO06b]. È stato analizzato attraverso simulazioni stocastiche, utilizzando Stochastic π-Calculus [MPW92a, Pri95] e SPiM [PC04, Phi07]: inoltre, rappresenta lo stadio preliminare di sviluppo del metodo.

Collective Sorting Dato un ambiente con diversi tipi di oggetti, si richiede una strategia distribuita per raggruppare insieme oggetti simili, separandoli da quelli differenti. Questo è il caso di studio di riferimento per il secondo e parte del terzo anno di dottorato, ad esempio [CGV07, GVCO08]. È stato valutato utilizzando il nostro ambiente di simulazione stocastica sviluppato con MAUDE [Mau07]: inoltre, rappresenta uno stadio intermedio dell'evoluzione del metodo.

Plain Diffusion Dato un insieme di nodi interconnessi che ospitano dati, si vogliono distribuire i dati in modo omogeneo tra i nodi. Questo è il caso di studio di riferimento per le pubblicazioni relative a parte del terzo anno di dottorato [GVO08]. Il caso di studio è stato analizzato attraverso simulazione e model checking stocastico utilizzando lo strumento PRISM [PRI07, KNP04]: inoltre, rappresenta lo stadio corrente di evoluzione del metodo.

Rispetto al contesto di ricerca attuale, il contributo della tesi è duplice: (i) proponiamo un approccio sistematico per l'ingegneria di sistemi MAS auto-organizzanti[1] utilizzando pattern di progettazione, simulazione e strumenti formali nella fase preliminare di progettazione; (ii) abbiamo analizzato tre casi di studio fornendo soluzioni auto-organizzanti alternative a quelle esistenti.

Struttura della Tesi

Descriviamo ora la struttura della tesi: si noti che i contributi originali della tesi sono principalmente concentrati nei capitoli 3, 4 e 6.

Capitolo 1 - Introduction Si presentano una panoramica della tesi, il contesto di ricerca, le motivazioni e gli effettivi contributi.

Capitolo 2 - Background In modo da comprendere la restante parte della tesi, presentiamo alcuni concetti e definizioni fondamentali: in particolare, si discutono le definizioni di sistema auto-organizzante e emergenza e si introducono i sistemi multiagente ed il metamodello A&A.

[1] Sebbene la metodologia sia stata concepita per MAS auto-organizzanti, crediamo che possa essere applicata con successo ad altri paradigmi.

Capitolo 3 - Design Patterns for Self-Organising Systems Si discute il ruolo dei pattern di progettazione per lo sviluppo di sistemi auto-organizzanti. In particolare, introduciamo un pattern architetturale che è parte fondamentale nel nostro approccio. Si discutono anche alcuni semplici pattern in accordo con lo schema di riferimento.

Capitolo 4 - A Systematic Approach for Engineering SOS Si descrive il metodo per l'ingegneria di MAS auto-organizzanti: l'approccio è di tipo iterativo ed ogni ciclo è composto da quattro fasi, precisamente, modellazione, simulazione, verifica e regolazione dei parametri. Si noti che, dalla prima formulazione, l'approccio è evoluto enormemente: infatti, mentre inizialmente consisteva principalmente nell'uso di simulazione, ora include anche aspetti di verifica formale e di pattern di progettazione.

Capitolo 5 - Formal Languages and Tools Prima di discutere i casi di studio, si effettua una panoramica dei linguaggi e strumenti formali impiegati, precisamente, SPiM, MAUDE e PRISM. In particolare, questi strumenti forniscono funzionalità differenti e sono stati utilizzati in momenti diversi dello sviluppo del metodo.

Capitolo 6 - Case Studies Si discutono i casi di studio che hanno caratterizzato i tre anni di dottorato. In particolare, si discute un caso di studio per ogni anno di corso, precisamente, Detection of Anomalous Behaviour, Collective Sorting e Plain Diffusion: ogni caso di studio è stato analizzato per mezzo di uno strumento differente ed è rappresentativo di un diverso momento dello sviluppo del metodo. Tutti i casi di studio sono strategie distribuite ed utilizzano differenti meccanismi auto-organizzanti.

Capitolo 7- Related Works Si discutono i lavori correlati all'ingegneria di sistemi auto-organizzanti, principalmente da un punto di vista metodologico: inoltre, si valuta l'uso in letteratura di strumenti formali per l'analisi o progettazione di sistemi auto-organizzanti.

Capitolo 8 - Conclusion and Future Works Si conclude riassumendo i contenuti, i contributi, le limitazioni e i possibili sviluppi futuri della tesi.

Chapter 1

Introduction

1.1 Research Context and Motivation

Self-organisation is increasingly being regarded as an effective approach to tackle modern systems complexity. The self-organisation approach allows the development of systems exhibiting complex dynamics and adapting to environmental perturbations without requiring a complete knowledge of the future surrounding conditions. A system developed according to the self-organisation principles produce global patterns and dynamics via local interaction of its components [SFH+04, MMTZ06]. Many biological systems can be effectively modelled using a self-organisation approach: well known examples include food foraging in ant colonies, nest building in termites societies, comb pattern in honeybees, brood sorting in ants [BDT99, CDF+01, SB06]. Self-organising mechanisms observed in Nature have inspired the development of many artificial systems, such as decentralised coordination for automated guided vehicles (AGV) [WSHL05, PBS05], congestion avoidance in circuit switched telecommunication networks [SA94], manufacturing scheduling and control for vehicle painting [CS04], self-organising peer-to-peer infrastructures [BMM02], and pheromone-based localization supported by a network of RFID tags [MZ07]. Furthermore, principles of self-organisation are currently investigated in several research initiatives that may have industrial relevance in a near future: notable examples include Amorphous Computing [AAC+00], Autonomic Computing [Hor01, KC03] and NASA Swarm Robotics [RHTR06].

However, the development of self-organising systems (SOS) is driven by different principles with respect to traditional engineering. For instance, engineers typically design systems as the composition of smaller elements, being either software abstractions or physical devices, where composition rules depend on the reference paradigm (e.g. the object-oriented one), but typically produce predictable results. Conversely, SOS display non-linear dynamics, which can hardly be captured by deterministic models, and, although robust with respect to external perturbations, are quite sensitive to changes on inner working parameters. In particular, engineering SOS poses two big challenges [GVO08]: How do we design the individual entities to produce the target global behaviour? And, can we provide guarantees of any sort about the emergence of specific patterns? Even though the existence of these issues is generally acknowledged,

few efforts have been devoted to the study of an engineering support either from methodologies and tools—except for a few explorations in the MAS (multiagent system) community [DW07, BCGP04, BCD+06].

1.2 Overview and Contributions

While simulation is effectively exploited in complex systems analysis, its potentialities in software engineering are typically overlooked [Tic98]: although there are examples of simulation in software development, these approaches involve simulation after the software has been already designed and developed, that is simulations are performed afterwards for profiling purposes. Conversely, we promote the use of simulation techniques at the early stages of software development, and a few works in this direction exist [vM93, Uhr02, DWHS06, BGP07]. Simulation allows us to preview overall qualitative system dynamics and devise a coarse set of working parameters before actually implementing the system. If we specify a model using formal languages we can then perform further analysis resorting to formal verification techniques such as model checking [CGL94, RKNP04].

Specifically, in this thesis, we describe methodological aspects concerning the early-design stage of SOS built relying on the agent-oriented paradigm: in particular we refer to the A&A metamodel [RVO06], where MAS are composed by agents and *artefacts*, i.e. environmental resources. Then, we describe an architectural pattern that has been extracted from a recurrent solution in designing self-organising systems [GVO07a]: this pattern is based on a MAS environment formed by artefacts, modelling non-proactive resources, and environmental agents acting on artefacts so as to enable self-organising mechanisms. In this context, we propose a scientific approach for the early design stage of the engineering of self-organising systems: in particular, the approach is articulated in four stages, modelling, simulation, formal verification, and tuning. During the modelling phase we mainly rely on the existence of a self-organising strategy observed in nature and, hopefully encoded as a design pattern [BCD+06, GVO07a, DWH07]. Simulations of an abstract system model are used to drive design choices until the required quality properties are obtained, thus providing guarantees that the subsequent design steps would lead to a correct implementation. However, system analysis exclusively based on simulation results does not provide sound guarantees for the engineering of complex systems: to this purpose, we envision the application of formal verification techniques, specifically *model checking* [CGL94], in order to exactly characterise the system behaviours. Given a formal specification a probabilistic model checker determines whether a specific property is satisfied or not or the actual likelihood value: properties are specified using different flavours of temporal logic, depending on the model type, e.g. probabilistic, stochastic or non-deterministic [RKNP04]. Unfortunately, the applicability of model checking techniques is hindered by the explosion of state space [CGL94]: nonetheless, in those cases model checking still a valuable tool for validating simulation results on small problem instances.

From its first formulation [GVO05c], the method has been greatly evolved through the exploration of several case studies and tools. In this thesis, we describe three case studies that we feel representative for each of the three years

of PhD course: in particular, each case study has been analysed using a different tool which supported the method in its state of advancement.

Detection of Anomalous Behaviour Despite a set of actions may be safe when executed individually upon a system, combinations of these actions may create unexpected damaging effects. This is the reference case study for the publications related to the PhD activities of the first year, e.g. see [GVO06a, GVO06b]. It has been analysed via stochastic simulation, relying on Stochastic π-Calculus [MPW92a, Pri95] and SPiM [PC04, Phi07]: furthermore, it represents the early development stage of our methodology.

Collective Sorting Given an environment with several kinds of items, we want to devise a distributed algorithm for clustering together similar items while separating different ones. This is the reference case study for the publications related to the PhD activities of the second and part of the third year, e.g. see [CGV07, GVCO08]. It has been analysed using our stochastic simulation framework developed on top of the MAUDE tool [Mau07]: furthermore, it represents an intermediate evolution of our methodology.

Plain Diffusion Given a networked set of nodes hosting information, we have to homogeneously distribute the information across the nodes. This is the reference case study for the publications related to the PhD activities of part of the third year [GVO08]. It has been analysed via simulation and model checking using the PRISM tool [PRI07, KNP04]: furthermore, it represents the current stage of the methodology.

With respect to the current research setting, the main contribution of the thesis is twofold: (i) we describe systematic approach for the engineering self-organising MAS[1] using design patterns, simulation and formal tools in the early design phase of the development of self-organising systems; (ii) we analyse three case studies providing alternative self-organising solutions.

1.3 Structure of the Thesis

We now provide an overview of the thesis structure and summaries for each chapter. It is worth noting that the original contributions of this thesis are mainly concentrated in Chapters 3, 4 and 6.

Chapter 1 - Introduction We provide an overview of the thesis, set the research context, describe the motivations and the actual contributions.

Chapter 2 - Background In order to understand and follow the rest of the thesis, we provide some necessary background information: in particular, we discuss definitions and examples of both self-organisation and emergence, and provide an introduction to multiagent systems and the A&A metamodel.

Chapter 3 - Design Patterns for Self-Organising Systems Here we discuss the role of design patterns for the development of self-organising

[1] Although the methodology has been conceived for self-organising MAS, we believe it might be successfully applied to other paradigms as well.

systems. In particular, we introduce an architectural pattern which is a fundamental element in our approach. Furthermore, we discuss a few simple design patterns with respect to our reference pattern scheme.

Chapter 4 - A Systematic Approach for Engineering SOS We describe our approach to the engineering of self-organising MAS: the approach is an iterative one and each cycle is articulated in four steps, namely, modelling, simulation, verification and tuning. It is worth noting that, from its first formulation, the approach has been evolved and refined to its current form: indeed, while initially mainly relied on simulation techniques, now it involves also formal verification techniques.

Chapter 5 - Formal Languages and Tools Before discussing the case studies, we provide an overview of the formal languages and tools used to analyse them, namely SPiM, MAUDE and PRISM. In particular, these tools provide different facilities and have been used at different stages of development of the method.

Chapter 6 - Case Studies We discuss the case studies that have characterised the three years of PhD activity. In particular, we discuss a case study for each year, namely, Detection of Anomalous Behaviour, Collective Sorting and Plain Diffusion: each case study has been analysed with a different tool and is representative of a particular stage of development of the method. While all the case studies are distributed strategies, they are fundamentally unrelated and uses different self-organising mechanisms.

Chapter 7- Related Works We discuss the works related to the engineering of self-organising systems, mainly from a methodological viewpoint: furthermore, we survey the uses of formal methods for the analysis of self-organising systems.

Chapter 8 - Conclusion and Future Works We conclude summarising the thesis, highlighting the contributions and the limitations, and listing future works.

Chapter 2

Background

In this chapter, we introduce the fundamental concepts and definitions that will be used throughout the thesis.

First, we provide a description of self-organisation and emergence, starting from historical roots and discussing current definitions. Then, we continue by introducing the multiagent paradigm, where systems are conceived in terms of agents and environmental abstractions: in particular, we describe the Agents & Artefacts metamodel, which is our reference metamodel when designing and modelling self-organising MAS.

2.1 Self-Organisation and Emergence

In this section, we point out the main conceptual elements of self-organisation and emergence, as well as provide a brief historical background: this section is mainly based on the following works [Gol99, DWH05].

2.1.1 The Definition of Self-Organisation

The term self-organisation suggests the idea of internal processes creating and supporting organisation: the actual definition of self-organisation is not so far from this intuitive explanation. The first explicit formulation of the idea that order and structure can spontaneously arise is due to the French philosopher and mathematician René Descartes: quoting from [Des37]

> [Consider] what would happen in a new world, if God were now to create somewhere in the imaginary spaces matter sufficient to compose one, and were to agitate variously and confusedly the different parts of this matter, so that there resulted a chaos as disordered as the poets ever feigned, and after that did nothing more than lend his ordinary concurrence to nature, and allow her to act in accordance with the laws which he had established. [..] Thereafter, I showed how the greatest part of the matter of this chaos must, in accordance with these laws, **dispose and arrange itself** in such a way as to present the appearance of heavens;

Although the basic concept was already contained in Descartes writings, the first appearance of the term self-organisation seems to be dated three centuries later, specifically in a 1947 paper by the English psychiatrist William Ross Ashby. Ashby defined self-organisation as the *ability of a system to change its own internal organisation, rather being changed from an external force* [Ash47].

In this definition the focus is placed on the *self* aspect, that is the control flow driving the system must be internal: unfortunately nothing is said about the distribution of control across system components. Sometime, when it is possible to identify a component that controls the re-organisation of components, it is called *weak* self-organisation. Conversely, when the control is distributed across multiple entities it is called *strong* self-organisation. This distinction between weak and strong self-organisation still in debate and literature provide different viewpoints. Although the differentiation between weak and strong self-organisation might seem appealing, in our opinion weak self-organisation can easily degenerate in the kind of pan-ism allowing to infer that almost every system is self-organising. Indeed, starting from a passive system it is sufficient to couple it with another system acting as a controller to have weak self-organisation: it is worth noting that every automatic system falls into this definition. From now on, when talking about self-organisation, we will implicitly refer to the case of internal distributed control.

For what it concerns the organisation part there are different viewpoints: organisation may be interpreted as spatial arrangement of components, system statistical entropy, differentiation of tasks among system components, the establishment information pathways, and the like. It can be observed that all this cases include some sort of relation between components either, topological, structural or functional.

Hence, we call a system self-organising if *it is able to re-organise itself by managing the relations between components, either topological, structural or functional, upon environment perturbations solely via the interactions of its components, with no requirement of external forces.* This definition implies four key features found in every self-organising system

Autonomy As previously discussed for the term self, the control must be located within the system and should be shielded from environmental forces: it is worth noting that here we do not mean closed systems, since most of the systems will rely on the environment resources in the shape of information, energy, and matter.

Organisation In the literature self-organising systems are often described as increasing their own organisation: actually, we see no point for this continual increase. Indeed, it is often the case that self-organising systems stabilises in sub-optimal solutions and their organisation varies over time both decreasing and increasing. Hence, we prefer just to say that there should be some re-organisation, not necessarily towards a better solution.

Dynamic Self-organisation should always be intended as a process and not as a final state.

Adaptive Perturbations may happen within the system as well as in the environment: a self-organising system should be able to compensate for these

perturbations. Centralized systems characterized by having a single-point-of-failure can hardly be considered self-organising because of the high sensitivity with respect to perturbations.

2.1.2 The Definition of Emergence

Intuitively, the notion of emergence is linked to novelty and an abstraction gap between the system components and the observed property. For instance, according to the definition discussed in [MÖ4], a phenomenon is emergent if and only if we have

> (i) a system of entities in interaction whose expression of the states and dynamics is made in an ontology or theory D, (ii) the production of a phenomenon, which could be a process, a stable state, or an invariant, which is necessarily global regarding the system of entities, (iii) the interpretation of this global phenomenon either by an observer or by the entities themselves via an inscription mechanism in another ontology or theory D'.

Although the origin of the term emergence can be traced back to Greeks, the first scientific investigation of emergence is due to philosophers of the British Emergentism in the 1870s, including J. S. Mill, but flourished only in the 1920s with the work of Alexander and Broad. Mill (1872) recognized that the notion of emergence was highly relevant to chemistry: indeed, chemistry and Biochemistry provide a wide range of examples involving emergent phenomena, indeed many chemical compounds exhibit properties that cannot be inferred from the individual components. For instance, consider the aromatic character of the benzene molecule: this property cannot be found in any of the components, but it arises as the product of the particular atoms configuration [Lui06]. The English philosopher G. H. Lewes (1875) distinguished in chemical reactions between resultants and emergents: while resultants were reducible to their reactants, emergents displayed properties not explainable in terms of the components.

Sometimes emergence is explained resorting to *holism*, and hence used in contraposition with *reductionism*. While, it is true that emergent properties are not reducible to the individual components, but can be understood considering the system as a whole, i.e. analysing the relations and interactions between parts, emergence is not only about that, but a more comprehensive concept. Furthermore, while holism and reductionism are concerned with system structure, emergence deals with system properties, that is an orthogonal dimension to structure [Lui06]. Hence, assimilating emergence to holism can be a source of confusion and should therefore treated with care.

2.1.3 Self-Organisation Vs. Emergence

In the literature there are plenty of misleading definitions of self-organisation and emergence: the most common misconception is about mixing self-organisation and emergence together in the same definition. This probably happens because there are many systems that are both self-organising and exhibit emergent properties [DWH05, DW07]. From previous discussion, it should be evident that self-organisation and emergence are different concepts, although often appearing together [DWH05].

Now we analyse two definitions, one about self-organisation and the other about emergence in order to highlight common misconceptions. For instance, consider the following definition of emergence provided in [Gol99]:

> Emergence [..] refers to the arising of novel and coherent structures, patterns, and properties during the process of **self-organisation** in complex systems. Emergent phenomena are conceptualized as occurring on the macro level, in contrast to the micro-level components and processes out of which they arise.

It is worth noting that this definition of emergence depends upon the definition of self-organisation, i.e. emergence is produced by self-organisation. Consider now the definition of self-organisation provided in [CDF+01]:

> Self-organisation is a process in which pattern at the global level of a system **emerges** solely from numerous interactions among the lower-level components of the system. Moreover, the rules specifying interactions among the system's components are executed using only local information, without reference to the global pattern.

Here the definition of self-organisation depends upon the definition of emergence and, as previously, self-organisation produces emergence.

Recursiveness in scientific definitions should be avoided because is a source of great confusion. Nonetheless, in the self-organisation and emergence literature, there are plenty of definitions similar to the quoted ones.

2.1.4 Example: Trail Formation in Ants

In order to clarify the concepts of self-organisation and emergence, we describe trail formation in ant colonies: the following discussion is mainly based on [CDF+01]. As soon as an ant find a food source, other ants arrive and join the food foraging task: soon, a trail is formed connecting the food source to the nest. As long as the food source is not exhausted, the trail is maintained and evolves towards the shortest path. When the food source is exhausted, the ants discard the former trail and try to discover new food sources.

This skill of ants to coordinate for achieving a global goal has always puzzled researchers. In 1959, the entomologist P.P. Grassé proposed the theory of stigmergy [Gra59], which explained coordination observed in social insects:

> The coordination of tasks and the regulation of constructions are not directly dependent from the workers, but from constructions themselves. The worker does not direct its own work, he is driven by it. We name this particular stimulation stigmergy.

In particular, Grassé pointed out that the environment plays the important role of shared coordination media. Later, it has been discovered that ants lay chemical substances, called *pheromone*, in the environment: the pheromone act as a marker for a specific activity. When an ant carries food, it deposits pheromone on its way back to the nest: other ants can perceive pheromone gradients, hence finding a trail. Ants join an existing trail to reach either the nest or the food source: when carrying food the ant tends to reinforce the existing pheromone trail. Conversely, the environment tend to evaporate the

pheromone, providing the negative feedback that closes the feedback loop: when the food source is exhausted, the pheromone trail is no longer reinforced and eventually disappears.

This system is a clear example of both self-organisation and emergence: it is self-organising since the trail is caused by indirect coordination between autonomous entities, the ants. It is emergent since, to explain the trail formation process it is necessary to consider the complex dynamics generated by the simple actions of perceiving and depositing pheromone: furthermore, the trail tend to approximate the shortest path, fact that is even more difficult to explain.

Since the discovery of this coordination mechanisms, ants have become a reference case study for self-organisation and have inspired several algorithms [DS04, DBS06] as well as industrial applications [PBS05, WSHL05, SA94, MZ07].

2.2 The Multiagent Paradigm and the A&A Metamodel

2.2.1 Introduction to the Multiagent Paradigm

In the last decade, software systems have changed dramatically, leading to new engineering challenges: currently, most of the systems are concurrent and the distribution of components is rapidly increasing. Furthermore, we are becoming more and more dependant on software functionalities for many key aspect of everyday life, and as a consequence software must be *always on*. Multiagent system (MAS) is a promising paradigm to face the complexity of modern ICT systems: quoting from [ZO04]

> Agent-based computing promotes designing and developing applications in terms of autonomous software entities (agents), situated in an environment, and that can flexibly achieve their goals by interacting with one another in terms of high-level protocols and languages.

In particular, we can identify three interesting aspects: (i) autonomy is necessary when designing or analysing distributed systems; (ii) flexibility of interaction is needed because of the very unpredictable environmental conditions; (iii) the notion of agent provide a unified view with artificial intelligence [RN02]. Then, an agent has the following characteristics [ZJW03, ZO04]

- autonomy: an agent is proactively oriented towards its goal, instead of being driven by external forces;
- situatedness: an agent can perceive and affect the environment where it is immersed, whether it is a computational environment or a physical one;
- sociality: an agent does not live in isolation, rather it is likely to coordinate with other agents to achieve a global or individual goal.

A multiagent system it is not just a collection of interacting agents: indeed, the environment plays an important role in multiagent systems providing agents with services ranging from basic life-cycle to advanced coordination aspects [WOO07, VHR+07].

Because of the new focus and challenges in multiagent systems, it is required to evolve software engineering practices to consider agent-specific issues,

that is Agent-Oriented Software Engineering (AOSE) [ZO04]. In the AOSE research context has been developed several methodologies such as Gaia [ZJW03], ADELFE [BCGP04] and SODA [MODR06]: each methodology was initially concerned with specific issues and has been later extended to encompass more aspects of the whole engineering process. We now consider the basic aspects when engineering self-organising systems, and then discuss a suitable MAS metamodel.

2.2.2 The Role of Environment in Self-Organising Systems

From the analysis of natural self-organising systems [BDT99, CDF+01, SB06], and existing experience in prototyping artificial ones [WSHL05, PBS05, CGV07, GVO07a, MZ05, MZ07], it is recognised that the environment plays a crucial role in the global SOS dynamics. A typical explanatory example is the case of stigmergy: as pointed out by [Gra59], among social insects the workers are driven by the environment in their activity. Indeed, in animal societies self-organisation is typically achieved by the interplay of individuals and the environment, such as the deposition of pheromone by ants or the movement of wooden chips by termites [CDF+01]. In particular, these interactions are responsible for the establishment and sustainment of a feedback loop: in the case of ant colonies, positive feedback is provided by ants depositing pheromones, while negative feedback is provided by the environment through evaporation [CDF+01].

When moving to artificial systems, and to MAS in particular, there are a few questions that need to be answered. The first one is where is the right loci for embedding self-organising mechanisms. The above discussion promotes the distribution of concerns between active components and the environment—in the MAS context, between agents and the environment. This partially frees agents from the burden of system complexity, and provides a more natural mapping for those behaviours that are not goal-oriented. The second question concerns which are the minimum requirements for an environment to support self-organisation. From the definition of self-organisation provided by [CDF+01] we can identify some basic requirements: *(i)* the environment must support indirect interactions between the components of a system, *(ii)* the environment must support some notion of locality, and *(iii)* such locality should affect interactions, e.g. promoting local ones. Moreover, specific self-organising mechanisms may require an *active environment*, i.e. the presence of active processes in the environment evolving the environment to a proper state: e.g. in pheromone-based systems, the environment may either provide a reactive *evaporation service*, or proactively act upon pheromone-like components to emulate the effect of evaporation.

2.2.3 Engineering MAS Environment Using the A&A Metamodel

Software conceived according to the MAS paradigm is modelled as a composition of agents, as autonomous entities situated in an environment, either computational or physical, interacting each other and with environmental resources to achieve either individual or social goals [WOO07, VHR+07]. Traditionally, the environment consists in the deployment context which provides communication services and access to physical resources: in this context, MAS engineers

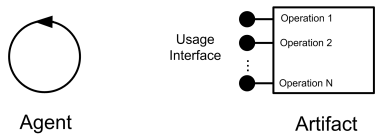

Figure 2.1: A&A metamodel featuring and agents as proactive goal-driven entities, and artefacts as encapsulating services to be exploited by agents through a usage interface.

design agents while the environment is just an output of the analysis stage. Recently, the environment has been recognised as an actual design dimension [WOO07, VHR+07]: then, MAS engineers can hide system complexity behind environmental services, freeing agents of specific responsibilities. In this article, we adopt the latter notion of environment, i.e. the part of MAS outside agents that engineers should design to reach the objectives of the application at hand.

In order to describe the environment, we have to provide suitable abstractions for environmental entities. As pointed out by [MOV07], despite most of the current AOSE methodologies and metamodels provide little – or sometimes completely absent – environment support, it is useful to adopt the A&A metamodel, where a MAS is modelled by two fundamental abstractions: *agents* and *artefacts* [RVO06]. Agents are autonomous pro-active entities encapsulating control, driven by their internal goal/task—left side of Figure 2.1. When developing a MAS, sometimes entities do not require neither autonomy nor proactivity to be correctly characterised. This is typical of entities that serve as tools to provide specific functionalities: these entities are the so-called artefacts. Artefacts are passive, reactive entities providing services and functionalities to be exploited by agents through a *usage interface*—right side of Figure 2.1. It is worth noting that artefacts typically realise those behaviours that cannot or do not require to be characterised as goal-oriented [RVO06]. Artefacts mediate agent interactions, support coordination in social activities, and embody the portion of the environment that can be designed and controlled to support MAS activities.

Chapter 3

Design Patterns for Self-Organising Systems

In this chapter, we discuss the role of design patterns for self-organising systems engineering. In particular, we present an architectural pattern encoding a recurrent solution for self-organising MAS: the pattern plays a key role in our methodology which is described in the next chapter.

The content of this chapter is mainly based on the following publications [GVO07b, GVO07a], although the underlying ideas can be found in several articles of ours. In particular, the architectural pattern has been first introduced in [GVCO07], while [GVCO08, GVO08] are the most recent works including aspects related to the pattern.

3.1 Motivations and Overview

When designing self-organising systems, it is becoming common practice to exploit existing models of natural systems, particularly social insects [BDT99]. The alternative approach consists in identifying the behaviours of individual agents eventually leading to the desired global dynamics: this direction has proven to be unsatisfactory, since it is mostly unfeasible for non-trivial systems. Hence, to our knowledge, the only reliable approach is to reverse engineer the strategies developed in Nature. In many articles, when it comes to discussing design choices it often appears a formula of the kind *"this design has been inspired by .."*: without any further detail, we assume that the authors have little control over their design process. Moreover, since the process of drawing inspiration is both time consuming and prone to error, we look for more systematic approaches.

To this purpose, we promote the use and development, since few works exist about this topic [BCD+06, DWH07, GVO07b, GVO07a], of design patterns for self-organising systems to establish a mapping between artificial systems problems and natural systems solutions. First introduced in 1977 by Alexander in architecture [AIS+77], later the concept of *design pattern* became popular in computer science with the object-oriented paradigm [GHJV95]. A design pattern provides a reusable solution to a recurrent problem in a specific domain: it is worth noting that a pattern does not describe an actual design, rather it

encodes an abstract model of the solution using specific entities of the paradigm in use. The use of design patterns offers several advantages, such as, reducing design-time by exploiting off-the-shelf solutions, and promoting collaboration by providing a shared language. Specifically, in the case of self-organising systems, patterns play a key role in driving the designer choices among the subtleties of complex systems dynamics.

To promote the acceptance of new patterns, as well as to reduce ambiguity, it is necessary to frame a pattern with respect to a shared scheme, ensuring a good degree of coherence of the whole pattern catalogue. For example, patterns for the object-oriented paradigm are described according to the scheme provided in [GHJV95]: as pointed out in [Lin03], since the agent paradigm cannot be effectively characterised using only object-oriented abstractions, patterns for MAS should be described using specific schemata. However, pattern schemata for MAS, like the one in [Lin03], do not adequately capture the peculiarities of self-organising systems, namely, which are the forces responsible for the feedback loop, and which notion of locality/topology is needed: to our knowledge, no specific pattern scheme has been proposed for self-organising MAS.

In this chapter, we start discussing our reference architectural pattern when designing self-organising MAS, appeared in [GVCO07] for the first time. Then, we briefly discuss a pattern scheme for self-organising MAS introduced in [GVO07a] and evaluate a few behavioural design patterns. It is worth noting that design patterns play a role in the modelling phase of our methodological approach, which is the subject matter of the next chapter.

3.2 Reference Architectural Pattern

The components of modern computational systems need often to interact with an environment populated by *legacy systems*. Hence, the environment can be either completely or partially given: this is subject to investigation during the analysis phase [MOV07]. In the MAS context, during the design phase, resources are assigned to artefacts, providing a uniform way for agents to exploit resources. Unfortunately, in a scenario involving legacy systems, we may only have partial control on the environment, thus making it difficult to embed self-organising mechanisms within artefacts. Then, to inject self-* properties in MAS, we need to add a layer on top of existing environmental resources.

To this purpose, we rely on the notion of *environmental agent*: such agents are responsible for managing artefacts to achieve the target self-* property. Hence, environmental agents are seen distinct from standard agents, also called *user agents*, which exploit artefact services to achieve individual and social goals. This recurrent solution has been encoded in the form of architectural pattern [GVO07a] with reference to the A&A metamodel. As shown in Figure 3.1, environmental agents act upon artefacts through a management interface: this interface may be public, i.e. accessible to all agents or, most likely, restricted and allowing access to operations typically granted to system administrators. A similar approach to achieve self-organisation, involving managers and managed entities, has been adopted also in the Autonomic Computing community [KC03].

Adopting the architecture encoded in this pattern provides several advantages. When working with legacy environmental resources – e.g. provided by an existing infrastructure – relying on additional environmental agents is the only

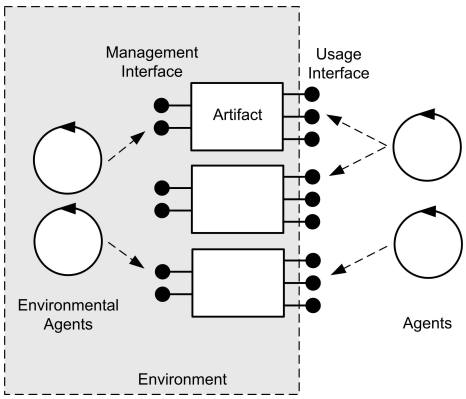

Figure 3.1: Architectural pattern featuring environmental agents as artefact administrators.

viable solution to add new properties and behaviour, due to a limited control on environmental resources. When developing systems from scratch, the use of this pattern allows different mechanisms to be isolated, thus achieving a finer control on the overall system. Furthermore, we are able to identify and suppress conflicting dynamics that may arise when exploiting different self-organising mechanisms at the same time [GVO07a]. It is worth noting that environmental agents differentiate from user agents because they play a special role in the system, bound within the environment as perceived by user agents. This is not in contradiction with previous works such as [ORV06, RVO06], but rather one step beyond: in fact, here user agents perceive the environment exactly in the same way, that is, populated only by artefacts, whereas environmental agents cannot interact with user agents, since they are somehow "encapsulated" within MAS environment.

This pattern can be successfully applied to embed self-organising mechanisms in MAS environments, especially to environmental services that do not natively support all the self-organisation features required. From a methodological viewpoint, when dealing with self-organising MASs relying on the A&A metamodel and the architectural pattern, the designer focuses its attention to the development of strategies for environmental agents. Indeed, environmental agents are the locus for encapsulating self-organising mechanisms, since we may not have control over environmental resources.

3.3 Reference Pattern Scheme

The literature provides several pattern schemata for different paradigms and metamodels, e.g. [GHJV95, Lin03]. In a first attempt to devise design patterns, we relied on the scheme for MAS patterns described in [Lin03]. We soon recognized the failure to capture essential self-organisation aspects, namely forces involved in the feedback loop and topology notion. The following pattern scheme extends the one described in [Lin03] and has been first introduced in [GVO07a]: the scheme is summarised in Table 3.1, where the novel items are emphasised. Particularly relevant to this work are the *feedback loop* and *locality* elements:

Name	The name of the pattern
Aliases	Alternative names
Problem	The problem solved by the pattern
Forces	Trade-offs in system dynamics
Entities	Entities participating to the pattern
Dynamics	Interactions between entities
Feedback Loop	Interactions responsible for the feedback loop
Locality	Describe the type of locality required
Dependencies	Environmental requirements
Example	An abstract example of usage
Implementation	Hints on implementation
Known Uses	Existing applications using the pattern
Consequences	Effects on the overall system design
See Also	References to other patterns

Table 3.1: This pattern scheme extends the one described in [Lin03] and has been first introduced in [GVO07a].

Feedback loop Describes the processes or actions involved in the establishment of a feedback loop, i.e. the actions providing positive and negative feedback. For example, in a digital pheromone infrastructure, the positive feedback consists in the agent depositing pheromones, while the environment provides the negative feedback in the form of pheromone evaporation.

Locality Requirements in terms of spatial topology or action-perception ranges: if the environment has a notion of continuous space, perception range is specified as a float value; if the environment has a graph topology, ranges are specified as the number of hops.

3.4 Basic Patterns for Self-Organising Systems

In this section, we describe a few design patterns already presented in [GVO07b, GVO07a]. Although these patterns describe very basic dynamics, they model fundamental aspects of many self-organising systems: furthermore, we prefer to consider simple patterns first and then develop more complex patterns relying on the basic ones.

3.4.1 Collective Sorting Pattern

Social insects tend to arrange items in their surroundings according to specific criteria, e.g. broods and larvae sorting in ant colonies [DGF+91, BDT99, CDF+01]. Collective sorting strategies may play an important role in artificial systems as well: for instance, grouping together related information helps to manage batch processing.

We consider our previous exploration of Collective Sorting dynamics in a MAS context [CGV07, GVCO07, GVCO08] in order to synthesise a pattern.

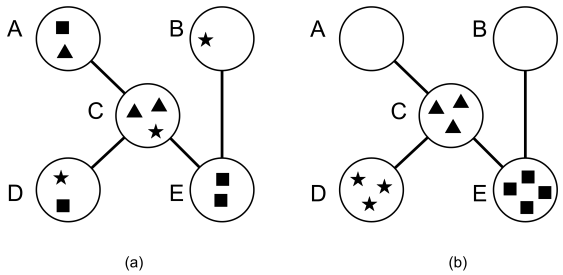

Figure 3.2: Collective sorting (a) an initial state (b) the final state.

From any arbitrary initial state, the goal of Collective Sorting is to group together similar information in the same node, while separating different kinds of information: Figure 3.2 displays a visual example of the system dynamics. Sometimes the deployment scenario does not allow to reach this goal: e.g. in a network having two nodes and three kinds of information, two kinds are going to coexist on the same node. Due to random initial situation and asynchronous interactions the whole system can be modelled as stochastic. Hence, it is not generally known a priori where a specific cluster will appear: clusters location is an emergent properties of the system [CGV07], which indeed supports robustness and unpredictable environmental conditions. Table 3.2 summarises the features of the collective sorting pattern.

3.4.2 Evaporation Pattern

In social insects colonies, coordination is often achieved by the use of chemical substances, usually in the form of pheromones: pheromones act as markers for specific activities, e.g. food foraging [BDT99, CDF$^+$01]. Specifically, these substances are regulated by environmental processes called *aggregation*, *diffusion* in space and *evaporation* over time: each process has its own relevance, hence it is analysed as a separate pattern. This class of mechanisms for indirect coordination mediated by the environment is called *stigmergy* [Gra59], and it has been widely applied in the engineering of artificial systems [MZ05, MZ07, PBS05, WSHL05]: for more details about stigmergy see Section .

Evaporation is a process observed in everyday life, although with different implications: e.g. from scent intensity it is possible to guess the parameters of its source, such as size and distance. In the case of insect colonies, marker concentration tracks activities: e.g. absence of pheromone implies no activity or no discovered food source. In ant food foraging [BDT99, CDF$^+$01] when a food source is exhausted, the pheromone trail is no longer reinforced and slowly fades away.

Evaporation has a counterpart in artificial systems that is related to information obsolescence [ROV$^+$07, Par06]. For instance, consider a news web portal: while newer information is inserted at the top of the page, older information *fades* as time passes, which can be visually translated into a movement towards the end of the page. In general, evaporation can be considered a mechanism to reduce information amount, based on a time relevance criterion. It is worth noting that evaporation fades information over time eventually erasing it, if no reinforcement is provided: Figure 3.3 displays the dynamics of evaporation.

Name	Collective Sorting
Aliases	Brood Sorting, Collective Clustering
Problem	MAS environments that do not explicitly impose constraints on information repositories may suffer from the overhead of available information.
Forces	Optimal techniques require more computation while reducing communication costs: on the other hand, heuristics allow background computation but increase communication costs.
Entities	The pattern involves artefacts, user agents and environmental agents.
Dynamics	User agents inject information in the artefacts. The artefacts have to provide specific content inspection primitives depending on the implementation. Environmental agents monitor artefacts for new information, and depending on artefacts content may decide to move an information to a neighboring artefacts.
Feedback Loop	Positive feedback is determined by environmental agents moving items to the appropriate cluster, while negative feedback happens when an item is misplaced.
Locality	Either continuous and discrete topology are suitable. Larger perception range improve strategy efficiency, but perception of immediate neighborhood is sufficient, but requires memory of items encountered.
Dependencies	It requires an environment compliant to the A&A metamodel.
Example	See Figure 3.2 for a visual example.
Implementation	Environmental agents may perform periodic inspection or been triggered by an insertion action: both approaches are suitable and choice depends on performance requirements. Moving information requires an aggregated view upon artefacts content, e.g. using counters or spatial entropy measures: in the case this is not feasible or too expensive, content sampling techniques can be used, see [CGV07] for a detailed discussion.
Known Uses	Explorations in robotics for sorting a physical environment [BDT99].
Consequences	Collective Sorting may not work when used in combination with other patterns that spread information across the MAS: in particular collective sorting opposes to Diffusion (Section 3.4.4).
See Also	-

Table 3.2: A summary of the features of the collective sorting pattern according to the reference scheme.

3.4.3 Aggregation Pattern

Pheromone deposited in the environment is spontaneously *aggregated*, i.e. separate quantities of pheromone are perceived as an individual quantity but with greater intensity [BDT99, CDF+01]: Figure 3.4 provides a visual example for the aggregation pattern. Aggregation is a mechanism of reinforcement and is also observable in human social tasks [ROV+07, Par06]. The ranking mechanism is a typical example [ROV+07]: when browsing the Internet someone finds

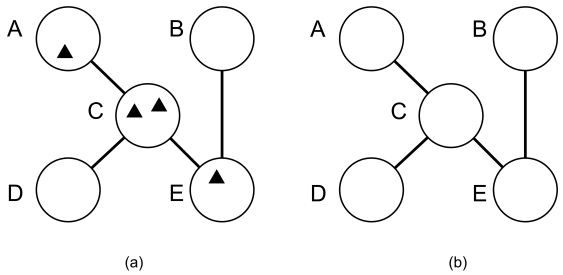

Figure 3.3: Evaporation (a) an initial state (b) the final state with no reinforcement.

Name	Evaporation
Aliases	None to our knowledge.
Problem	MAS environments can soon become overwhelmed by information deployed by agents.
Forces	Higher evaporation rates release memory, but require more computation: furthermore, evaporated information cannot be recovered!
Entities	The pattern involves artefacts, user agents and environmental agents.
Dynamics	User agents inject information in the artefacts. The artefacts assign a time-stamp/counter to the received information. Environmental agents erase obsolete information/information whose counter reached zero: eventually, all the information is removed.
Feedback Loop	User agents deposit items in the environment while environmental agent evaporate them.
Locality	Perceptions and actions happen only locally. Either continuous and discrete topologies are suitable.
Dependencies	It requires an environment compliant to the A&A metamodel.
Example	See Figure 3.3 for a visual example.
Implementation	Environmental agents may perform periodic inspection or been triggered by a specific event: both approaches are suitable and choice depends on performance requirements.
Known Uses	A fundamental element of stigmergy [CDF+01] and digital pheromone based application [MZ05, MZ07, PBS05, WSHL05].
Consequences	-
See Also	When used in combination with Aggregation (Section 3.4.3) or Diffusion (Section 3.4.4), it allows for building complex behaviours: in particular, Evaporation + Aggregation + Diffusion is the Stigmergy pattern.

Table 3.3: A summary of the features of the evaporation pattern according to the reference scheme.

an interesting fact, he/she can leave a (reinforcement) comment that is typically anonymously and automatically aggregated with comments of other users.

It is then evident that, while evaporation is driven by the environment,

aggregation is driven by the user agent. When used in combination with evaporation, aggregation lets the designer close a positive/negative feedback loop, allowing for auto-regulated system in self-organisation and Autonomic Computing [KC03] styles.

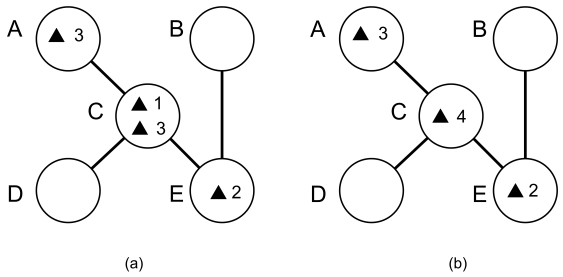

Figure 3.4: Aggregation (a) an initial state (b) the final state.

3.4.4 Diffusion Pattern

When pheromone is deposited into the environment it spontaneously tends to diffuse into neighboring locations according to local concentrations [BDT99, CDF+01]. This process, called *diffusion*, is omnipresent in nature and hence is studied in several fields under different names, e.g. osmosis in chemistry. Starting from any arbitrary state, diffusion eventually distribute the information equally across all nodes, providing an distributed averaging system [BCD+06]: Figure 3.5 displays a visual example of diffusion.

While aggregation and evaporation processes act locally, diffusion requires a notion of topology. Furthermore, in diffusion the initial quantity of information is *conserved* but spatially spread: other forms of diffusion may be conceived to produce stable gradients [MZ05].

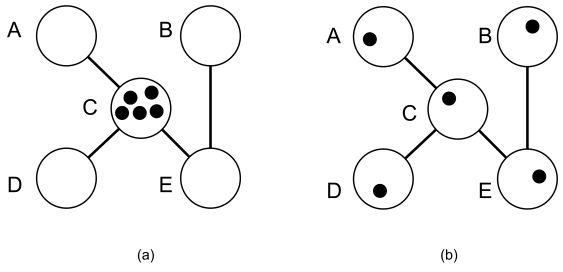

Figure 3.5: Diffusion dynamics (a) an initial state (b) the desired final state.

Name	Aggregation
Aliases	None to our knowledge.
Problem	Large scale MAS suffer from the amount of information deposited by agents, which has to be sifted in order to synthesise macro information.
Forces	Higher aggregation rates provide results closer to the actual environment status, but require more computation.
Entities	The pattern involves artefacts, user agents and environmental agents.
Dynamics	User agents inject information in the artefacts. Environmental agents look for new information and aggregate it with older information to produce a coherent result.
Feedback Loop	User agents deposit items in the environment while environmental agents synthesise an aggregated info.
Locality	Perceptions and actions happen only locally. Either continuous and discrete topologies are suitable.
Dependencies	It requires an environment compliant to the A&A metamodel.
Example	See Figure 3.4 for a visual example.
Implementation	Environmental agents may perform periodic inspection or been triggered by a specific event: both approaches are suitable and choice depends on performance requirements. It is worth noting that aggregation is a very simple task and could be automatically handled by artefacts, when properly programmed: on the other hand, it is easier to have separate agents for different functionalities, which can be individually paused or stopped.
Known Uses	A fundamental element of stigmergy [CDF+01] and digital pheromone based application [MZ05, MZ07, PBS05, WSHL05]. In e-commerce applications customers feedback is usually aggregated, e.g. average ranking, in order to guide other customers.
Consequences	-
See Also	When used in combination with Evaporation (Section 3.4.2) or Diffusion (Section 3.4.4), it allows the synthesis of more complex behaviours: in particular, Evaporation + Aggregation + Diffusion is the Stigmergy pattern.

Table 3.4: A summary of the features of the aggregation pattern according to the reference scheme.

Name	Diffusion
Aliases	Plain Diffusion, Osmosis.
Problem	In MAS where agents have limited perception radius, the reasoning process may suffer from the lack of knowledge about neighboring nodes.
Forces	Higher diffusion radius brings information further away from its source, providing a guidance also to distant agents: as a consequence, the infrastructure load increases, both in terms of computation and memory occupation. Furthermore, diffused information does not reflect the current status of the environment hence providing false hints.
Entities	The pattern involves artefacts, user agents and environmental agents.
Dynamics	User agents inject information in the artefacts. A weight is assigned to the information from artefacts or user agents. Environmental agents diffuse information decreasing the weights in local node and correspondingly increasing the weights in neighboring nodes.
Feedback Loop	User agents deposit items in the environment while environmental agent scatter them to neighboring locations.
Locality	User agents perceptions and actions happens only locally, while environmental agents need to perceive and act at least at one hop of distance. Either continuous and discrete topologies are suitable.
Dependencies	It requires an environment compliant to the A&A metamodel.
Example	See Figure 3.5 for a visual example.
Implementation	Environmental agents may perform periodic inspection or been triggered by a specific event: both approaches are suitable and choice depends on performance requirements.
Known Uses	A fundamental element of stigmergy [CDF+01] and digital pheromone based application [MZ05, MZ07, PBS05, WSHL05]. In e-commerce applications the *see-also* hint is a typical example of information diffusion were the topology is built upon a similarity criterion of products.
Consequences	Diffusion may not work when used in combination with other patterns that spread information across the MAS: in particular diffusion opposes to Collective Sorting (Section 3.4.1).
See Also	When used in combination with Evaporation (Section 3.4.2) or Aggregation (Section 3.4.3), it allows the synthesis of more complex behaviours: in particular, Evaporation + Aggregation + Diffusion is the Stigmergy pattern.

Table 3.5: A summary of the features of the diffusion pattern according to the reference scheme.

Chapter 4

A Systematic Approach for Engineering Self-Organising Systems

In this chapter, we describe our methodological approach for engineering self-organising MAS according to the A&A metamodel and our architectural pattern. Initially, it was mostly a simulation-driven approach exploiting formal tools and languages, e.g. see [GVC06b]: then, it has been evolved to an iterative approach articulated in four stages, namely (1) modelling, (2) simulation, (3) verification, and (4) tuning, exploiting formal languages and tools at each step, e.g. see [GVO08]. This chapter is mainly based on the publications [GVO05a, GVO05c, GVO05b, GVO06a, GVC06b, GVO06b, GVC06a, GVCO07, GVCO08, GVO08]: each publication provides a snapshot of the methodology at a different stage of development. The most recent publication containing the latest stage of development of the methodology is [GVO08].

4.1 Motivation and Context

Currently, the development of agent-oriented systems is supported by several software engineering methodologies. Most methodologies were initially conceived to cover specific issues, and then evolved to encompass the whole software process: for instance, the Gaia methodology [ZJW03] was mostly concerned with intra-agents problems, while the initial target of the SODA methodology [MODR06] was to tackle the social, inter-agent dimension. Conversely, other methodologies restricted the domain of applicability to a specific class of MAS, like ADELFE for the Adaptive MAS theory [BCGP04].

Concerns like embedding self-organising mechanisms within an existing MAS, or engineering SOS from scratch, raise peculiar issues that are not typical or so crucial in current AOSE methodologies. Indeed, as pointed out in [MOV07], even core elements such as the environment is currently explicitly supported by only a few of the existing AOSE methodologies. Furthermore, AOSE methodologies, as well as object-oriented ones, tend to focus on design-time aspects rather than run-time ones: in fact, it is common practice to assume that once

a system has been designed, its structure will not change and will behave according to the specifications. The Autonomic Computing proposal suggests to consider run-time issues at design-time: then, aspects such as maintenance become a functional requirement of the problem to be solved [KC03], thus increasing the degree of autonomy and adaptiveness of the target system. Along this line, we promote the use of techniques that allow us to preview and analyse global system dynamics at design-time: indeed, when dealing with SOS, more attention should be devoted to observe the emergence of desired properties early in the design stage rather than waiting for the final implementation. In particular, when developing SOS, we have to answer the following question: how can we design the individual environmental agent's behaviour in order to ensure the emergence of the desired properties? To tackle this issue, two approaches are typically exploited: (i) devising an *ad-hoc* strategy by decomposition that will solve the specific problem; (ii) observing a system that achieves similar results, and trying to reverse-engineer its strategy. It is generally acknowledged that the former approach is applicable only to a limited set of simple scenarios: due to the non-linearity in entity behaviours, global system dynamics becomes quite difficult to be predicted. Instead, in the self-organisation community, the latter approach is commonly regarded as more fruitful: in nature, it is possible to recognise patterns that are effectively applicable to artificial systems [BCD$^+$06, DW07, GVO07a, BDT99]. Since it is quite unlikely to find a pattern that completely fits a given problem, it is common practice to rely on some modified and adjusted version—in the next section we will elaborate on the implications of these modifications.

Then, once a suitable strategy has been identified and adapted, how can we guarantee that it will behave as expected? Given the specifications of a SOS, how to ensure the emergence of the desired global dynamics is still an open issue. While automatic verification of properties is typically a viable approach with deterministic models, verification becomes more difficult and soon intractable when moving to stochastic models: then, it is useful to resort to a different approach, possibly mixing formal tools and empirical evaluations, so as to support the analysis of the behaviour and qualities of a design.

Before describing our approach in the next section, we would like to point out that it is not our goal to develop a brand-new complete methodology for MAS engineering. Instead, we would rather aim at integrating our approach within existing AOSE methodologies, and addressing the peculiar issues raised by self-organising MAS. For instance, by considering Gaia [ZJW03], our approach could be seen as a way to direct early design phases: on the one hand, this could help the developer in defining responsibilities for agents and services of the environment by taking inspiration from patterns found in natural systems; on the other hand, it could make it possible to preview the global dynamics of the MAS and tune its behaviour before committing to a specific design solution.

4.2 Overview of the Approach

Our approach for the engineering of self-organising MAS relies on the previously described A&A metamodel and architectural pattern. Since we embed self-organising mechanisms into environmental agents, according to the architectural pattern, our method is mainly focused on the behaviour of such agents. Our

method should not be considered a full methodology, i.e. encompassing aspects from requirements to maintenance: conversely, we heavily rely on existing MAS methodologies for many development stages. Indeed, we mainly concentrate in the early design phase, bridging the gap between analysis and the actual design phase: this is probably the most delicate phase when dealing with self-organisation and emergence because of the complexity in dynamics.

Our methodology could be summarised by the statement *"bringing science back into computer science"* [Tic98]: indeed, it is heavily based on existing tools commonly used in scientific analysis, especially for complex systems, but that are typically not used in software development. Specifically, our approach is iterative, that is, cycles are performed before committing to a specific design: during each cycle four steps are performed

1. *modelling* proposing a model for the system to engineer based on existing design patterns;
2. *simulation* analysing global qualitative dynamics in different scenarios before continuing with quantitative analysis;
3. *verification* verifying that the properties of interest holds and identifying working conditions;
4. *tuning* adjusting system behaviour and devising a coarse set of parameters for the actual system.

Across the whole process we rely on the use of formal tools and techniques, in order to provide unambiguous specifications and enable automatic processing. At this development stage of the method, we exploit the PRISM tool [PRI07, KNP04], a Probabilistic Symbolic Model Checker developed at University of Birmingham that provides model-checking capabilities along with simulation and model editing integrated within the same software: more details about the tool can be found in Section 5.3. We now continue detailing issues related to each of the steps.

4.3 Modelling

In the modelling phase we develop an abstract model of the system, providing a characterisation for (i) environmental agents, (ii) artefacts, and eventually (iii) user agents. As far as artefacts are concerned, we can provide an accurate model of their behaviour with respect to the usage interface and set of services exposed—though it is often the case that a detailed description of the inner working is not available. Conversely, the repertoire of user agents' behaviour may be too vast to be accurately modelled: indeed, in open environments, it is basically impossible to entirely foresee the dynamics of agents to come—self-organisation is precisely used to adapt to unpredicted situations. Then, it is necessary to abstract from their peculiarity, resorting to probabilistic or stochastic models of user agents' behaviour. Models for these agents are developed in terms of usage of resources, i.e. with respect to the observable behaviour and by abstracting away from inner processes such as planning and reasoning. The accuracy of the user-agent internal model is not so crucial, since self-organisation

is built on top of indirect interactions mediated by the environment: hence, it is sufficient to know how user agents perceive and modify their environment.

Once a suitable model for user agents and artefacts is provided, we move to the core part of modelling, that is, the characterisation of environmental agents. A suitable model for environmental agents is typically built on top of the services provided by artefacts, and functionally coupled with user agents' behaviour in order to establish and sustain a feedback loop: indeed, a feedback loop is a necessary element in every self-organising system. Consider for example the case of ant colonies, where ants deposit pheromone which diffuses and evaporates in the environment: then, by perceiving pheromone gradient, ants can coordinate their movement without a priori knowledge of the path to follow.

To find a candidate model for environmental agents, we can take inspiration from known SOS and look for a model exhibiting or approximating the target dynamics. This step implies the existence of some sort of design-pattern catalogue, a required tool for an engineer of SOS. Although this sort of catalogue does not exist yet, several efforts by different research groups are moving along this direction. Indeed, several patterns with important applications in artificial systems have already been identified and characterised [BDT99, BCD$^+$06, DW07, GVO07a]: for more details on the topic refer to Chapter 3. Hence, even though patterns that perfectly match the target system dynamics can hardly be found, it is still feasible to identify some patterns approximating such a dynamics; however, this typically requires typically requiring changes of some sort. Given the complex behaviour that characterises SOS, modifications should be done with care since they require expertise in mechanisms underlying SOS.

Although the model may be provided in several notations, we favour the use of formal languages. In fact, formal languages allow both to devise unambiguous specifications and to perform further automatic analysis, such as simulation and verification: we will discuss more about simulation and verification in the next two sections.

4.4 Simulation

We use simulation tools to quickly and easily preview system dynamics before actually performing quantitative analysis. Before performing simulations, we have to define two key aspects: (i) providing a set of suitable parameters for the model, and (ii) choosing the test instances, i.e. the initial states we consider representative and challenging for the system.

When dealing with self-organising MAS we mostly rely on stochastic simulation in order to capture both timing and probability aspects. Parameters for this kind of simulation are typically expressed in terms of *rates of action* defined according to suitable statistical distributions: the exponential distribution is typically used because of the *memoryless property*, i.e. to generate new events it is not necessary to know the whole event history but only the current state. Furthermore, the use of exponential distribution allows the mapping to Continuous-Time Markov Chains, which are commonly used in simulation and performance analysis. Then, rates should reflect the conditions in the deployment scenario, otherwise the simulations results would be meaningless. While parameters for artefacts can be accurately measured, user agents provide a ma-

jor challenge since we cannot foresee all their possible behaviours. Hence, we have to make assumptions about artefact behaviours both from the qualitative (e.g. rational exploitation of resources) and quantitative (e.g. rate of actions and rate of arrivals/departures) standpoint: once the parameters of artefacts and agents are defined, we devise an initial set of parameters for environmental agents.

Testing the dynamics of the system in different scenarios and worst case scenarios is very important for a reliable evaluation of the quality and robustness of the model. Typically, we tend to make observations first in very extreme conditions, strictly dependent on the application at hand, since they quickly reveal the presence of faults in the model: then, we continue the analysis with more real scenarios. In the simulation stage we do not perform quantitative analysis, which is instead a major concern in the verification and tuning stage: this is especially true during the first iteration cycle, since we still do not have sufficiently characterised the system model.

4.5 Verification

Among the available verification approaches, we are interested in the one called model checking: *model checking* is a formal technique for automatically verifying the properties of a target system against its model [EMCGP99]. The model checker accepts a formal specification of the system and a list of properties expressed in a suitable variant of temporal logic: then, the properties are automatically evaluated from the model checker for every system execution. Hence, with respect to simulation that test only a subset of all possible executions, model checking provides more reliable results: the main drawback of model checking is known as the *state explosion problem* which does not allow to perform model checking for a problem instance having a large states space. As far as tools are concerned, we currently rely on the PRISM-Probabilistic Symbolic Model Checker [PRI07], a software developed at University of Birmingham. More details about model checking and the PRISM tool can be found in Section 5.3.

In our approach, model checking is exploited to provide strong guarantees about the emergence of global properties, and in general for precisely characterising the dynamics of the systems. From the viewpoint of the model checker, emergent properties are not different from ordinary properties. Conversely, from the designer viewpoint emergent properties are quite challenging: a model checker needs properties formulated according to modelled states, but because the very nature of emergent properties, modelled states cannot be automatically mapped to the emergent property. Hence, the designer should shift from the global to the individual dynamics and identify the micro properties that are a symptom of the emergent property. This practice will become more clear in Section 6.3 when applying the method to a case study.

4.6 Tuning

In the *tuning* phase, environmental agents' behaviour and working parameters are successively adjusted until the desired dynamics are observed. The tuning

process is performed exploiting both simulation and verification tools to devise a coarse set of working parameters for the actual system: this process still not automatised, hence it can be quite time-consuming. It is worth noting that setting parameters to arbitrary values may lead to unrealistic scenarios. Furthermore, the working rate of environmental agents may affect the actual working rate of artefacts: in a realistic scenario, computational resources are typically limited, hence increasing the working rate of environmental agents may require a decrease in the service rate of artefacts. Without considering this problem, the dynamics of the deployed system may significantly deviate from the expected ones.

At the end of the tuning process, we may realise that the devised set of parameters does not satisfy performance expectations, features unrealistic values with respect to the execution environment, or deviates the system from the desired behaviour. In any of these scenarios, we cannot proceed to the actual design phase since the system is not likely to behave properly when deployed. Hence, it is required to perform another iteration for reconsidering the modelling choices or evaluating other approaches. Conversely, when a model meets the target dynamics and the parameters lie within the admissible ranges, we can proceed by providing a more accurate statistical characterisation of the system behaviours: this can be performed either by simulation, when the problem instance is too big or does not require strong guarantees, or by model checking which produces more accurate results.

Chapter 5

Formal Languages and Tools

In this chapter, we describe the most relevant formal languages/techniques and the respective software tools evaluated during the activities that lead to this thesis. Specifically, we discuss them in the same order as we first evaluated them:

- stochastic simulation with stochastic π-calculus [MPW92a, Pri95] and the Stochastic Pi-Machine (SPiM) [Phi07, PC04]: these techniques and tools have been used in the following publications [GVO05a, GVO05c, GVO05b, Gar05, GVO06a, GVO06b, GVC06a] and have been evaluated with respect to the case study *Detection of Anomalous Behaviour*, see Section 6.1;
- stochastic simulation with the MAUDE tool: these techniques and tools have been used in the following publications [CGV06b, CGV06c, CGV06a, VCG07, CGV07, GVCO07, GVCO08] and have been evaluated with respect to the case study *Collective Sorting*, see Section 6.2;
- simulation and model checking [EMCGP99, KNP07] using the Probabilistic Symbolic Model Checker (PRISM) [KNP04, PRI07]: these techniques and tools have been used in the following publication [GVO08] and have been evaluated with respect to the case study Plain Diffusion 6.3.

In particular, the use of model checking techniques and the PRISM tool reflects the current state of our research and support the advancements in our methodological approach. In the next chapter, we will discuss a case study for each of the presented formal tools.

5.1 Simulation with Stochastic π-Calculus and SPiM

In this section, we describe the stochastic π-calculus and the Stochastic Pi-Machine (SPiM), a software tool to run stochastic simulations from system specifications written in Stochastic π-Calculus: we do not discuss here the formal semantics of π-Calculus and SPiM because it is out of the scope of this thesis.

5.1.1 From Process Algebra to Stochastic π-Calculus

Process Algebra is a class of formal languages for modelling concurrent systems, in particular providing support for interaction and composition of processes.

Among the various Process Algebra, we provide here an overview of the π-Calculus, a calculus of processes having changing structure [Mil99, MPW92a, MPW92b]: in particular, π-Calculus extends a previous Process Algebra, Calculus of Communicating Systems (CCS), by adding support for channel mobility. According to [Mil99, MPW92a], the π-Calculus syntax can be summarized by

$$P ::= 0 \mid P_1 + P_2 \mid \bar{y}x.P \mid y(x).P \mid \tau.P \mid P_1|P_2 \mid (\nu x)P \mid [x = y]P \mid !P \qquad (5.1)$$

- 0 is the empty process, and it's called *inaction*, often omitted;
- the silent prefix τ means that a silent action is performed;
- the replication $!P$ means that you can have as many copies – but a finite number – as you wish, i.e. $P|P|P|..$;
- summation $P_1 + P_2$ means that the process behaves like P_1 or P_2 in a non-deterministic fashion;
- the prefix $\bar{y}x$ is a sort of output port, so $\bar{y}x.P$ means send x across y channel and then behave like P;
- the prefix $y(x)$ is a sort of input port, so $y(x).P$ means receive a value across y channel, name it x and then behave like P;
- the composition $P_1|P_2$ means that the two processes are executed in parallel;
- the restriction $(\nu x)P$ means that the process behaves like P except for the fact that any action across x channel is prohibited;
- $[x = y]P$ means that the process behaves like P if y matches x, otherwise 0.

As an example to illustrate π-Calculus syntax, we consider a simple client-server system: we are interested in modelling the interactions between Client and Server mediated by a Queue.

$$\begin{aligned} C &\equiv push().C \\ S &\equiv pop().S \\ Q &\equiv (\overline{push} + \overline{pop}).Q \\ Sys &\equiv (C \mid S \mid Q) \end{aligned}$$

As it can be noticed, the specification is very simple and consists in three processes running in parallel modelling respectively a client, a server and a queue.

Although very useful to model the system behaviour, π-Calculus does not allows the evaluation of quantitative aspects, which are very important to profile performance and reliability. The solution proposed by Stochastic π-Calculus [Pri95] is to add probabilistic distributions to actions: the duration of an action is an aleatory variable drawn according to the probabilistic distribution. To

this purpose, the exponential distribution is a common choice because of the *memoryless property*, i.e. an action duration does not depends on system history but only current state: the exponential distribution $P = 1 - e^{-rt}$ is completely defined by the parameter r which is called *rate*. Hence, stochastic π-calculus labels actions with rate, modelling an aleatory action duration: furthermore, the use of rates rules out non-determinism. Indeed, in a choice the fastest action is chosen while the others are discarded, according to a *race condition* [Pri95]: it is worth noting that since the action duration is aleatory, each time the selected action may differ.

The use of exponential distributions in stochastic π-calculus allows the construction of Markov Chains [BH01]: Markov Chains are commonly used for numerical analysis of systems mainly for performance evaluation, simulation and formal verification.

5.1.2 SPiM: the Stochastic Pi-Machine

The Stochastic Pi-Machine (SPiM) is a software for simulating systems modelled in Stochastic π-Calculus: specifically, SPiM implements a variant of the calculus [Phi07, PC04]. Initially developed for investigating biological systems [PC04], SPiM can be used to simulate any systems that can be modelled in Stochastic π-Calculus.

Basically, SPiM accepts a stochastic π-calculus specification and produces a simulation trace based on user directives, e.g. duration and sampling. For programming convenience, SPiM language uses a different syntax with respect to the original π-Calculus: for further details please refer to the SPiM Language Manual [Phi07].

The stochastic selection strategy has been implemented following the Gillespie's algorithm [Gil77, PC04]: basically, an action is selected and the respective duration is computed depending on the rates and the number of processes willing to perform that action. Because of that simulation algorithm, we may experience some unexpected results when modelling systems other than chemical reactions or biological processes. In artificial systems, the duration of an action depends on the rate itself, and not on the number of components: indeed, a server does not work faster as the number of clients increases.

As an example to illustrate SPiM syntax, we consider a simple client-server system: we are interested in observing the dynamics of the requests queue, i.e. the pending job submission.

```
directive sample 0.01
directive plot ?count as "Job"

new push@1.0 : chan
new pop@1.0 : chan
new count@1000.0 : chan

let Client() = !push; Client()
let Server() = !pop; Server()
let QM() =
do ?push; (QM() | ?count)
or ?pop; (QM() | !count)
```

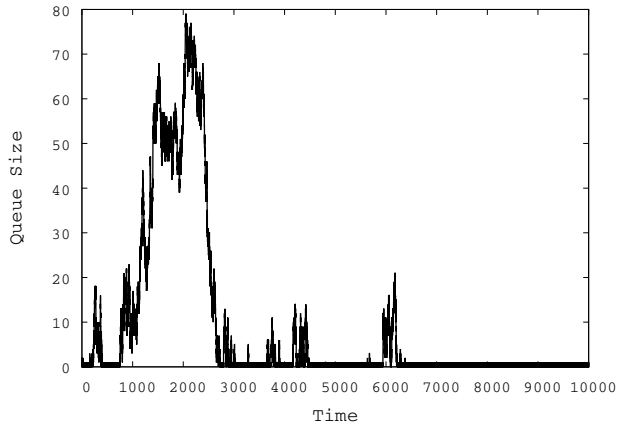

Figure 5.1: A simulation run of the client-server specification showing the queue size evolution.

```
run 1 of (Client(); Server(); QM())
```

The specification starts with two directives `sample` and `plot` respectively indicating the duration of the simulation and the quantities to trace. Then, it follows the declaration of channels: `push` models the client request, `pop` models the server response and `count` serves as a counter for pending requests, i.e. queue size. Then, it comes the actual processes specification: the `Client()` sends a signal on the `push` channel and continues; the `Server()` sends a signal on the `pop` channel and continues; the queue manager `QM()` waits for signals on the channels `push` or `pop`, then continues and in parallel either sends or waits for a signal on the channel `count`. Using channels in the shape of counters by matching the actions `!count` and `?count` is a common practice. SPiM allows to trace only the number processes or the number of channels waiting for communication: as previously stated, due to the simulation algorithm, counting processes may cause unexpected results. The last instruction simply starts one instance for each of the defined processes.

Given the previous specification, it is then possible to run simulations, eventually tuning channels rate to evaluate qualitative system dynamics. For instance, Figure 5.1 shows the evolution of the queue size: notice that, because of the stochastic nature of the system, bursts of activity may occur.

5.2 Stochastic Simulation with Maude

5.2.1 Overview of Maude

Maude is a high-performance reflective language supporting both equational and rewriting logic for specifying a wide range of applications [Mau07, CDE+07].

Rewriting logic is a logic of concurrent change that can naturally deal with state and with concurrent computations.

MAUDE should be intended as a metalanguage to provide domain-specific languages. MAUDE basic programming statements are *equations* and *rules*, both having a simple rewriting semantics in which instances of the left hand side pattern are replaced by corresponding instances of the right hand side [CDE+07].

In the course of finding a general simulation tool for stochastic systems, we considered MAUDE a particularly appealing framework, for it allows to directly model a system in terms of transition rules, or to prototype a new domain-dependent language to have more expressiveness and compact specifications. This is therefore a natural starting point for addressing the simulation of stochastic aspects in coordination: other languages require the designer to model systems in terms the available abstractions, e.g. channels and processes in π-Calculus [MPW92a], which might not be suitable in the general case. On the other hand, MAUDE cannot be used off the self, since it requires additional efforts for the definition of a custom language. Since MAUDE does not natively support stochastic aspects, we have developed a stochastic simulation framework on top of it. In the next section, we provide an overview of the simulation framework: for more details the reader can refer to [CGV07]. The full specification of the simulation framework is listed in Appendix A.

5.2.2 A Stochastic Simulation Framework

The idea of our library is to model a stochastic system by a labelled transition system where transitions are of the kind $S \xrightarrow{r:a} S'$, meaning that the system in state S can move to state S' by action a, where r is the (global) *rate* of action a in state S. The rate of an action in a given state can be understood as the number of times an action could occur in a time-unit (if the system would rest in state S), namely, its occurrence frequency. This idea is inspired by the activity mechanism of stochastic π-Calculus [Pri95, PC04], 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 similar to that approach, for the way the global rate is computed is custom, and ultimately depends on the application at hand, e.g. the global rate can be fixed, or can depend on the number of system sub-processes willing to execute an action. Given a transition system of this kind and an initial state, a simulation is simply executed by: (i) checking each time the available actions and their rate; (ii) picking one of them probabilistically (the higher the rate, the more likely the action occurs); (iii) accordingly changing the system state; and finally (iv) advancing the time counter following an exponential distribution, so that the average frequency is the sum of the action rates. This technique is again similar to the one adopted in SPiM [PC04].

The framework implementation is organised in three Maude modules:

- `STOCHASTIC-SELECTION` contains the definition of the functions handling probabilities and randomness;
- `STANDARD-CARRIER` provides all the definitions a specific system has to implement in order to be simulated by this tool;

- `STOCHASTIC-TRACES` contains the definition of the simulation engine.

We are not describing here the whole implementation of the simulation framework, hence, the interested reader should refer to [CGV07].

We consider now the simple example of the $Na - Cl$ chemical reaction to briefly explain the process of creating a system specification to simulate.

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

 sort NaClState .
 subsort NaClState < State .

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

 ops ionization deionization : -> Action .

 vars Na Na+ Cl Cl- : Nat .

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

endm
```

This system is characterised by a state of the kind `<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: `ionize` stands for ionization and `deionize` for deionization. Finally, the transition system is expressed by a single equation, associating to any state two possible effects: one in which ionization decrements `Na` and `Cl` (by prefix predecessor function `p`) and increments `Na+` and `Cl-` (by prefix successor function `s`), and the other that behaves in the opposite way. Note that, according to the Gillespie's selection algorithm in [Gil77], the rate of ionization and deionization is here proportional to the product of the two reactants, multiplied by a constant value: that is, we here enforce deionization factor as being twice that of ionization. Below, it is displayed a sample simulation trace, showing that the system reaches a dynamical equilibrium around `<60,40,60,40>`.

```
[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],
..
[7 : < 61,39,61,39 > @ 3.9845251139158447e-2],
[6 : < 60,40,60,40 > @ 3.9980318990300842e-2],
[5 : < 59,41,59,41 > @ 4.029131950475788e-2],
[4 : < 58,42,58,42 > @ 4.0294167525983679e-2],
[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]
```

5.3 Probabilistic Model Checking with PRISM

In this section, we provide a brief overview of model checking techniques and describe the facilities provided by the PRISM software tool.

5.3.1 Model Checking

Model checking is a formal technique for automatically verifying the properties of a target system against its model [EMCGP99]. The model to be verified is

expressed in a formal language, typically in a transition system fashion: the model checker accepts the finite state system specification and translate it into an internal representation, e.g. Binary Decision Diagrams (BDD) [EMCGP99]. Properties to be verified are expressed using a suitable variant of temporal logic, e.g. Linear Temporal Logic (LTL) [EMCGP99]: then, the properties are automatically evaluated from the model checker for every system execution. The main drawback of model checking is the *state explosion problem*: since states space typically grows in a combinatorial way, the number of states quickly become untractable as the system grows. Modern techniques allows the reduction of this problem, and verification of models with large states space has been carried out successfully [CGL94]: nonetheless, state explosion is still the main limitation of model checking and abstraction is often required.

Although model checking was initially targeted to deterministic systems, advancements in the last decade consider also probabilistic as well as stochastic aspects [KNP07, RKNP04]. A probabilistic model checker uses a temporal logic extended with the suitable operators for expressing probabilities, e.g. Probabilistic Computational Tree Logic (PCTL) and Continuous Stochastic Logic (CSL) [KNP07, RKNP04]. Beyond boolean answers, a probabilistic model checker allows the computation of the actual probability value for the tested property.

In our approach, model checking techniques are exploited to provide strong guarantees about the emergence of global properties, and in general for precisely characterise the dynamics of the systems. From the viewpoint of the model checker, emergent properties are not qualitatively different from ordinary properties. Conversely, from the designer viewpoint the encoding of emergent properties is quite challenging: a model checker needs properties formulated according to modelled states, but because the very nature of emergent properties, modelled states cannot be automatically mapped to the emergent property. Hence, the designer should shift from the global to the individual dynamics and identify the micro properties that are a symptom of the emergent property.

5.3.2 The PRISM Software

PRISM-Probabilistic Symbolic Model Checker is a software tool developed at University of Birmingham [PRI07, KNP04]: PRISM provides integrated editing and probabilistic model checking capabilities, and basic simulation tools mainly for debugging purpose. The PRISM modelling language is based on Reactive Modules and models are specified in a transition systems fashion. The language is able to represent either probabilistic, non-deterministic and stochastic systems using, respectively, Discrete-Time Markov Chains (DTMC), Markov Decision Processes (MDP) and Continuous-Time Markov Chains (CTMC) [KNP04, KNP07]. Components of a system are specified using modules, while the system state is encoded as a set of finite-values variables. Furthermore, modules are allowed to interact using synchronisation in a process algebra style, i.e. labelling commands with actions: module composition is achieved using the standard parallel composition of Communicating Sequential Processes (CSP) process algebra.

As an example consider the specification of a stochastic cyclic counter having base 100

```
ctmc
module counter
    value : [0..99] init 0;
    [] value < 99 -> 1.0 : (value'=value+1);
    [] value = 99 -> 1.0 : (value'=0);
endmodule
```

where `ctmc` declares that the specification models a Continuous Time Markov Chain, the module definition is wrapped into the block `module .. endmodule`, `value` is a variable ranging between 0 and 99 initialised to 0, and a transition is expressed according to the syntax `[] guard -> rate : (variable-update)`. For more features and details about the PRISM language syntax please refer to the PRISM documentation [PRI07].

Given the previous specification it is possible to use the PRISM built-in simulator to observe the system dynamics. The simulator engine makes it possible either to perform step-by-step simulation, or to specify the number of steps to be executed: in particular, the step-by-step mode which is very useful for debugging purposes. The simulator traces the values for each variable in the model, in this case only `value`: it is worth noting that the variable range does not affect simulation performance, conversely to model checking, but can produce unexpected results since ranges affect transition guards. PRISM does not provide any plotting capability, however it allows simulation traces to be exported in different formats, so this is not a major concern and can be easily overcome using third-party plotting software. Unfortunately, PRISM does not allow experiments to be run, i.e. multiple simulations spanning values for several parameters.

PRISM model checking facilities are very robust: it provides three model checking engines, namely, MTBDD, Sparse and Hybrid, having different memory and computational costs. Properties are expressed according to the temporal logics Probabilistic Computational Tree Logic (PCTL) for DTMC and MDP models, and Continuous Stochastic Logic (CSL) for CTMC models [KNP04, KNP07]. Beyond boolean properties, PRISM allows the computation of actual probability values as well as values for reward-based properties. For instance, referring back to the cyclic counter, the simple property "Which is the steady state probability for the counter to contain the value 10?" is encoded in the CSL formula `S=? [value=10]`, and obviously the result is 0.01. Another example, "Which is the probability for the counter to reach the value 80 within 75 time units?" is encoded in the PCTL formula `P=? [true U<=75 A=80]`, and the result approximated to four decimals is 0.2968.

A very compelling feature of PRISM, because of the CPU intensive nature of model checking algorithms, is the ability to run model checking experiments. Once the values for parameters range have been defined, the tool automatically performs verification in all the combinations of parameters values.

Another interesting feature is the possibility to evaluate properties with simulation instead of model checking: this allows to extend model checking results when the instance becomes to large to be formally verified. The user is allowed to set several parameters, e.g. confidence, sample, in order to obtain the desired approximation level.

Chapter 6

Case studies

In this chapter, we discuss three case studies, namely, Detection of Anomalous Behaviour, Collective Sorting and Plain Diffusion: each case study is representative of several aspects

Detection of Anomalous Behaviour It is the reference case study for the publications related to the PhD activities of the first year [GVO05a, GVO05c, GVO05b, Gar05, GVO06a, GVO06b]. It has been analysed via stochastic simulation, relying on Stochastic π-Calculus and SPiM (see Section 5.1): furthermore, it represents the early development stage of our methodology.

Collective Sorting It is the reference case study for the publications related to the PhD activities of the second and part of the third year [CGV06b, CGV06c, CGV06a, VCG07, GVCO07, CGV07, GVCO08]. It has been analysed using the stochastic simulation framework developed on top of the MAUDE tool (see Section 5.2): furthermore, it represents an intermediate evolution of our methodology.

Plain Diffusion It is the reference case study for the publications related to the PhD activities of part of the third year [GVO08]. It has been analysed via simulation and model checking using the PRISM tool (see Section 5.3): furthermore, it represents the current stage of the methodology.

6.1 Detection of Anomalous Behaviour

In this section, we analyse the case of a basic Intrusion Detection System for multiagent systems that detects anomalies in agents' behaviour. A self-organisation approach is adopted to dynamically balance the resources required to detection depending on the entity of intruders' attack. For the formal specification and the simulation, we rely on stochastic π-calculus [MPW92a, Pri95] and the SPiM tool [Phi07, PC04], see Section 5.1.

6.1.1 Emergent Harmful Sequences of Actions and Intrusion Detection

Every artificial resource implies a usage protocol, i.e. a collection of usage patterns: for example, a pattern for an agent using a DVD player might consist in (i) turning-on the device, (ii) inserting the DVD, (iii) pushing the play button, (iv) pushing the stop button, (v) removing the DVD, and (vi) turning-off the device. If the user behaves differently than what the pattern prescribes, it may get no result, e.g. when pushing the stop button before the play button. But in some cases non-functional requirements may enter the picture: it is possible that combinations of actions complying with usage protocol may result in a damage for the system, e.g. when repeatedly turning on and off a light. Moreover, as the usage protocol gets complicated and the number of combinations grows, it becomes quite difficult, from an engineering viewpoint, to specify or even foretell all the harmful sequences. While we believe that the most critical sequences of actions should be handled at design time, i.e. statically or via a pattern database, it is in general too expensive to account for all the possibilities, and some alternative approach might be worth pursuing.

The field of information security is in fact witnessing this scenario. With respect to the early days of computer, security techniques are now more sophisticated and are supported by publicly available automated tool: accordingly, the efficiency and subtlety of attacks is increasing. Traditional techniques, such as authentication, authorisation, and role-based access control are mostly static mechanisms and work as barriers, and are recognised as no longer sufficient to protect our information systems [FHS97]. A relatively recent approach to avoid these drawbacks is Intrusion Detection (ID), which tries to discover unsafe accesses to networked electronic devices and, in general, anomalies in the use of computational resources [MCA00]. ID techniques have been widely inspired from the mechanisms regulating the human immune system [FHS97]: several properties of such a system have been regarded as desirable also for information system security, like distribution, adaptiveness, protection at multiple levels, uniqueness of the immune response. ID techniques are partitioned onto two categories:

Signature-based (Also called *pattern-based*) They require a database of signatures, i.e. sequences of actions symptomatic of a malicious activity. Signatures are synthesised by observing previous attacks, hence, this approach cannot discover new threats. The main advantage of this approach is the low false alarm rate.

Anomaly-based They try to discover anomalies in the use of resources, by observing and modelling normal system usage, and thus detecting abnormal ones by comparison. This approach is able to detect novel attacks and relies on automated techniques developed in decision theory and artificial intelligence. The main drawback of this approach is that the false alarm rate could be significant.

As far as emergence and self-organisation are concerned, we shall in the following focus on anomaly-based approaches.

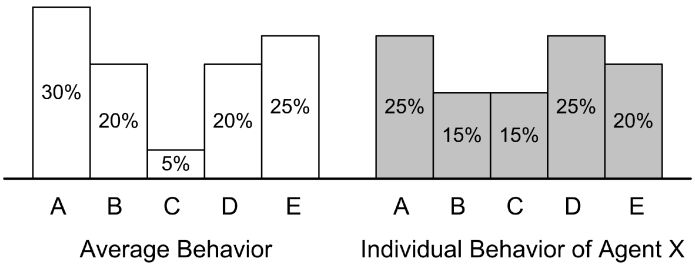

Figure 6.1: Comparison between the normal behaviour of agents and the behaviour of a specific agent.

6.1.2 A Basic Architecture for Intrusion Detection in an Agents & Artefacts Environments

In this section, we describe an architecture for implementing an intrusion detection layer on top an environment compliant to the A&A metamodel.

In our hypothetical MAS, agents dynamically join the society and then interact with resources modelled as artefacts: we assume the environment already provides authentication and authorization policies. We want to add a further security level accounting for dynamic and emergent aspects, such as e.g. detecting novel and unpredicted attacks though an anomaly-based IDS.

Consider an artefact exposing a usage interface composed of 5 possible actions, which we refer to as A, B, C, D, and E. An anomaly-based IDS needs first to trace agents' behaviour "for a while": gathered data are then used to build a profile of what is for agents a "normal way" to interact with that particular artefact. For instance, a basic profiling approach consists in gathering the relative percentage of actions, thus providing a statistical characterisation—see Figure 6.1 (left). At the same time, it is possible to profile the individual agent behaviour when interacting with that artefact—see Figure 6.1 (right).

Note that this approach makes sense under two hypotheses: *(i)* the number of traces is such that the data is statistically significant, and *(ii)* the percentage of agents exhibiting abnormal behaviour is not relevant during the initial observation stage. Then, it is possible to program ID so as to analyse the deviation of the specific agent behaviour from the average one, which is considered to be a symptom of intrusion or abnormal activity: some criteria is then applied as soon as sufficient data has been gathered for each agent. When an agent is detected, the intrusion response system properly reacts, e.g. banning the agent from the society. Several criteria for detection can be defined and used in combination:

- total deviation should be less than a given threshold (e.g. 20%);
- deviation on individual critical actions should be less than a given threshold (e.g. 5%);
- an agent can behave abnormally no longer than a given interval (e.g. 2 minutes).

Obviously, these criteria—and any more sophisticated approach evaluated in literature—influence false alarm and missed detection rates. However, here we are not concerned with performance issues at that level, and suppose both

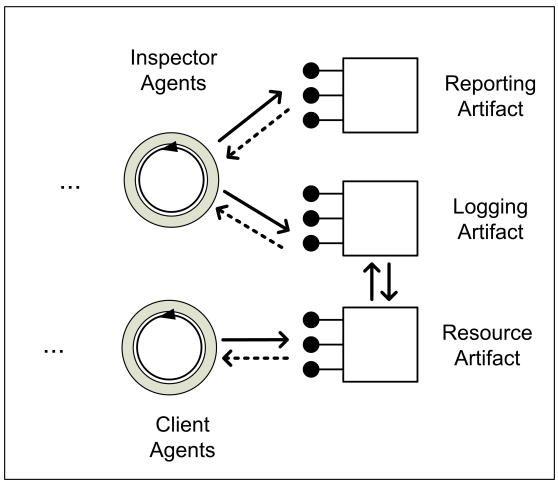

Figure 6.2: The basic architecture of the anomaly detection systems: the basic entities are agents and artefacts.

these factors are zero. Instead, we focus on designing the proper strategy for inspecting an agent abnormal behaviour, so as to reduce the overhead required for the IDS to effectively react to certain attacks.

In a possible implementation of the IDS for an A&A based environment, we would use three different artefacts:

Resource artefact Providing the actual service exploited by the agent, and exposing the usage interface;

Logging artefact Maintaining a coherent and updated characterisation of agents' behaviour, gathered from the interactions occurring with the resource artefact;

Reporting artefact Reporting the suspicious behaviours as discovered by inspector agents.

Two kinds of agents are then hosted: *client* agents requesting services and whose behaviour is monitored, and *inspector* agents responsible for applying the policies of the IDS. In our specific case, an inspector compares the average signature with the individual ones and report any anomaly to the reporting artefact.

For obvious reasons of performance, it is not possible to inspect all agents at each step, thus this is to be done through periodical sampling. Inspection is characterised by two parameters: the rate of inspection of each inspector agent, and the number of inspectors that are active simultaneously. Note that it is functionally equivalent to have one inspector working at rate kr_{insp} or k inspectors working at rate r_{insp}: while from an implementation viewpoint, as pointed out in [FHS97], having several entities makes the system more robust. The basic architecture of the system is depicted in Figure 6.2.

As this system should not be overloaded with the ID task for reasons of reliability, the system should *self-regulate*, raising its defences, otherwise releasing computational resources delegated to monitoring. Given this architecture,

we seek for a strategy to adapt inspection rate to the actual intruders in the system, satisfying the following basic requirements:

- if there are no intruders the rate of inspection should be at a minimum value;
- if intruders enter the system, the inspection should work at higher rate, resulting in the detection of more anomalies;
- detected agents are banned, causing a decrease in the number of intruders: inspection rate should also correspondingly decrease, since the attack is being contained;
- if new intruders arrive then the system should raise again the inspection rate etc.

In the next section we illustrate how to apply our methodological approach[1] to carefully design one such strategy with given performance characteristics.

6.1.3 Modelling the Solution

The first step consists in finding an adequate abstract model of the system to implement, exhibiting the emergent properties required for the problem at hand. One possibility is to use the "*catalogue*" of self-organising systems studied so far, looking for a model whose global dynamics matches the desired one. Among the many natural systems explored in the literature, it is easy to recognise that interesting similarities are exhibited by the *prey-predator system* [Ber92, SB06].

This model attempts to describe the coupled evolution of populations of preys and predators in biological systems. Intuitively, the model states that the evolution of a population depends on the product of the number of preys and predators, the population of predators grows as they kill preys, while it tend to vanish as preys diminish. The system evolution can be describe by the couple of equations

$$\begin{cases} dN/dt = aN - bNP \\ dP/dt = cNP - dP \end{cases}, \tag{6.1}$$

where N and P are respectively the amount of preys and predators, a and d are the rate of changes in isolation, b and c the rates of changes due to prey-predator interaction. The above set of equations leads to three possible cases, depending on these parameters (see Figure 6.3):

1. there is a dynamic equilibrium between preys and predators, and both populations evolve in a periodical fashion;
2. preys population grows faster than the one of predators, leading to an overpopulated environment;
3. preys population grows too slow to survive predators, leading to extinction of both populations.

[1] Notice that we are here referring to the early version of the approach as described by first papers, e.g. [GVO06a, GVO06b]

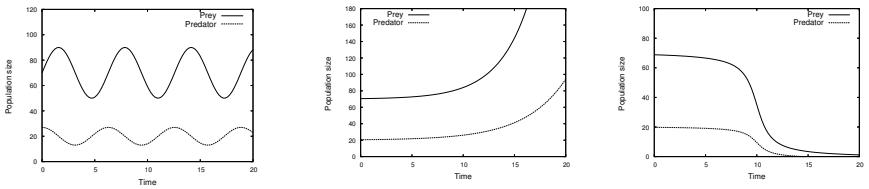

Figure 6.3: Qualitative dynamics of a prey-predator system in the case of dynamic equilibrium (left), overpopulation (centre) and extinction (right).

This model has similarities with the IDS we want to realise, for agents with abnormal behaviour very much resemble preys caught by inspector agents acting as predators—agents with correct behaviour form instead a population immune from predators. Moreover, we recognise the former of the three cases above as the one we intend to reproduce, since it is the situation where the population of intruders can be kept under control with the minimum allocation of resources for the IDS, and is the situation where detection really self-regulate with respect to the amount of anomaly.

Therefore, we proceed by modelling a system where the population of inspector agents grows as intruders are found, and diminishes in the opposite case. In particular, abnormally behaving agents enter the system according to various dynamics—we simulate and analyse some in the following—and are killed as soon as they are found by some inspecting agent. On the other hand, the system starts with one inspector agent, which clones itself as it perceives an abnormal activity; each such a cloned agent would possibly clone itself, but has a specific lifetime (expressed in terms of number of inspections) after which it dies. Hence, our solution provides both the positive feedback due to anomaly detection and the negative feedback due to limited lifetime, thus resembling the mechanisms of human immune system cells [FHS97].

In order to simulate the model of our system we need to develop an executable specification: to this purpose, we rely on stochastic π-calculus, and accordingly write the specification with the syntax of the SPiM tool, see Section 5.1 for more details.

Normally behaving agents They properly interact with resources and, when requested, reply to inspection;

Abnormally behaving agents They improperly interact with resources and, when requested, reply to inspection;

Inspector agents They periodically inspect agents and clone themselves adaptively driven by the perceived abnormal activity.

We first consider the specification of a normal agent. This is expressed as a π-calculus process `AgentN`. The process recursively spends some time in an idle state—modelling interaction with the resource artefact—or possibly receives a signal on the `inspect()` channel, which is replied with with a signal on the

channel `isNormal()`—modelling inspection. In SPiM syntax this is expressed by the following process definition:

```
let AgentN() =
    do delay@serviceRate;  AgentN()
    or ?inspect; !isNormal; AgentN()
```

Symbols "`?`" and "`!`" stand for receiving and sending a stimulus, `inspect` and `isNormal` are channels with instantaneous communication, `delay` is a channel with rate `serviceRate`, "`;`" is sequential composition, and the two final `AgentN()` statements realise recursive calls.

Similarly, the behaviour of an abnormal agent is realised through a process `AgentA` interacting with resources and receiving a signal on the `inspect` channel, replying with a signal on the channel `isAbnormal`. In the latter case, the process halts (no behaviour is specified as continuation), modelling the agent being banned from the system. This is expressed by the following definition:

```
let AgentA() =
    do delay@serviceRate;  AgentA()
    or ?inspect; !isAbnormal
```

The basic definition of an inspector is a bit more articulated, and is as follows:

```
let Inspector() =
    delay@inspectionRate;
    !inspect;
    do ?isAbnormal; (InspectorCloned(lifeTime) |
                     !cInspector |
                     ?cAbnormal |
                     Inspector())
    or ?isNormal; Inspector()
```

We first have exactly one occurrence of an inspector agent that never dies, which we use to guarantee that the service is always available. Such an inspector interacts according to a rate specified by the rate `inspectionRate` of channel `delay`. Each time it first signals on the channel `inspect`, which is perceived by one agent—either normal or abnormal as seen above—modelling its inspection. If the agent is recognised as abnormal—channel `isAbnormal` is stimulated—a new inspector is created and counters useful for simulation are updated. In any case, the behaviour `Inspector` is recursively called. The above counters are realised with processes which increment a value when they receive a message, and decrease it if they are able to send a message: counter `cInspector` and `cAbnormal` respectively track the number of inspectors and abnormal agents.

Process `InspectorCloned` realises an inspector similarly to `Inspector`, but the difference here is that such an inspector dies after `lifeTime` iterations, hence it is of this kind:

```
let InspectorCloned(life:int) =
    if life >0 then
       delay@inspectionRate;
       !inspect;
       do ?isAbnormal; (InspectorCloned(life-1) |
                        InspectorCloned(lifeTime) |
```

```
                    !cInspector |
                    ?cAbnormal )
    or ?isNormal; InspectorCloned(life-1)
else ?cInspector;
```

Process `InspectorCloned` has one integer parameter representing its lifetime. If this is greater than zero the behaviour is similar to a standard inspector, the only difference is that recursion makes lifetime decrease by one. If this is zero, the counter of inspectors is decreased, and the process ends.

By combining all these definitions and completing them with the declaration of channels, names and directives, we obtain the complete SPiM specification, as reported in Figure 6.4.

The first part of the specification sets simulation steps and values to be displayed (counter of abnormal agents and inspectors). Then, some constants are set, followed by the definition of channels along with their rate–rate 1000000.0 is used to simulate an instantaneous interaction. After the definition of processes seen above, the specification ends with the starting process, which in this case is made up of 1000 normal agents, and 1 inspector. For a detailed explanation of syntax and semantics the interested readers can refer to [Phi07].

6.1.4 Simulation and Tuning

Given the system specification above, the next step is to evaluate the global dynamics and tune the parameters: in particular we are interested in the dynamics of the population of inspectors (predators) and abnormal agents (preys). Different values for that parameters really lead to different performance values, hence it is generally necessary to run different simulations until hopefully reaching the desired quality of system behaviour.

For instance, supposing the inspection rate is 6 (that is, 6 inspections per time unit), and the lifetime of inspectors is 30, we obtain an instance of the IDS which reacts to a continuous attack in Figure 6.5. Here, normal agents are 1000, while abnormal agents are initially zero and are introduced periodically with a fixed rate, namely, 1 agent per time unit. As we see from the chart, the population of abnormal agents grows until around 40 elements, that is 4% of the entire agent population, but after a while inspectors are cloned and cause abnormal agents to die. The system reaches an equilibrium, where an average of 0.5% of inspector agents are able to bound abnormal agents to an average of 3%.

Changing the above parameters leads to different results. Supposing it is the designer's goal to limiting further abnormal agents, possibly requiring more inspectors, hence more overhead of the ID process: namely, keeping the population of abnormal agents to 2% with a number of inspectors around 1%. This is achieved by increasing inspectors lifetime, from 30 to 40, as shown in Figure 6.6.

Other interesting simulations that can be used to detect some features of this IDS instance vary in the kind of attack. For instance, we can simulate how the IDS would react to an impulse-like, massive attack, where an high number of abnormal agents suddenly appear. The chart in Figure 6.7 shows the result with an initial population with 10% of abnormal agents.

Again, the population of inspectors immediately starts growing, causing abnormal agents to reduce until disappearing after 20 time units; promptly, in-

```
(* Directives for Simulator *)
directive sample 500.0
directive plot !cAbnormal as "AgentA"; !cInspector as "Inspector"

(* Global Constants and Parameters *)
val arrivalRate = 1.0
val inspectionRate = 6.0
val serviceRate = 1.0
val lifeTime = 40

(* Channels for Signaling Purpose *)
new inspect@1000000.0:chan
new isNormal@1000000.0:chan
new isAbnormal@1000000.0:chan

(* Channels for Counting Events *)
new cAbnormal@1000000.0:chan
new cNormal@1000000.0:chan
new cInspector@1000000.0:chan

(* Agents Specifications *)
let AgentN() =  do delay@serviceRate;  AgentN()
                or ?inspect; !isNormal; AgentN()
let AgentA() =  do delay@serviceRate;  AgentA()
                or ?inspect; !isAbnormal
let Inspector() =  delay@inspectionRate;
                   !inspect;
                   do ?isAbnormal; (InspectorCloned(lifeTime) |
                                    !cInspector |
                                    ?cAbnormal |
                                    Inspector())
                   or ?isNormal; Inspector()
and InspectorCloned(life:int) = if life >0 then
                                    delay@inspectionRate;
                                    !inspect;
                                    do ?isAbnormal;
                                            (InspectorCloned(life-1) |
                                             InspectorCloned(lifeTime) |
                                             !cInspector |
                                             ?cAbnormal )
                                    or ?isNormal; InspectorCloned(life-1)
                                else ?cInspector;

(* Periodically Introduce Abnormal Agents in the System *)
let Welcome() = delay@arrivalRate; (AgentA() | !cAbnormal | Welcome())

(* Set up Initial Conditions *)
run 1000 of (AgentN() | !cNormal)
run 1 of (Inspector() | !cInspector)
run 1 of Welcome()
```

Figure 6.4: Complete specification of the system for the SPiM tool.

spectors reduce as well to the initial conditions. To tune the system in order to achieve different reaction performances, new simulations have to be run.

As we find a set of suitable parameters we can proceed to the actual implementation: otherwise we would have to test different parameters, and if the global dynamics would not reach the requirements it is necessary to get back to the abstract system model changing/adapting its general strategy.

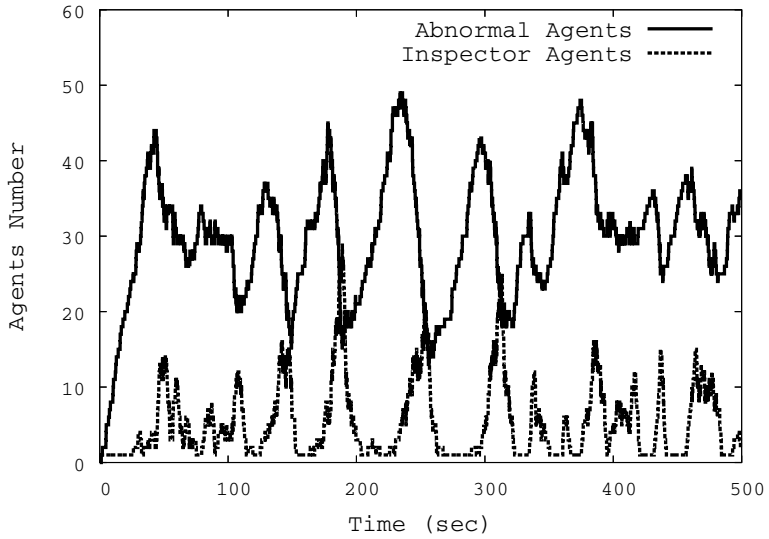

Figure 6.5: Dynamics of abnormal agents and inspectors with lifetime set to 30.

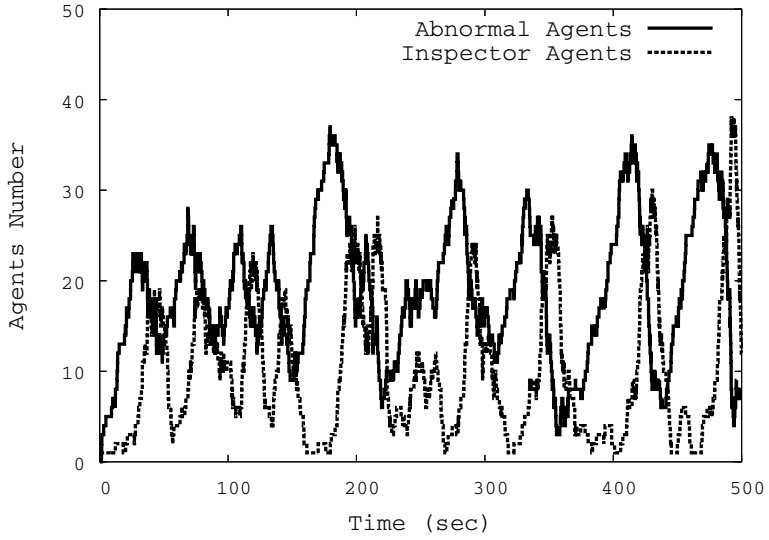

Figure 6.6: Dynamics of abnormal agents and inspectors with lifetime set to 40.

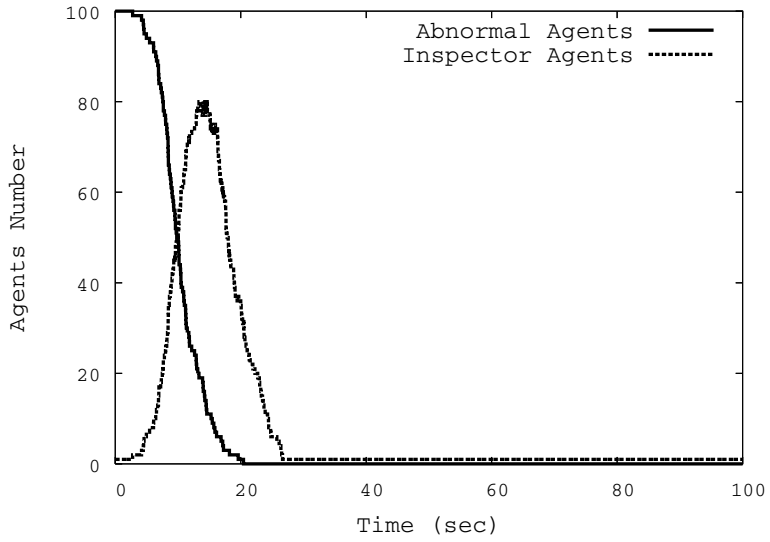

Figure 6.7: System behaviour in response to an impulse of abnormal behaviour.

6.2 Collective Sorting

In this section we analyse the case study of *collective sorting* [VCG07, CGV07, GVCO08], a distributed sorting algorithm inspired by the sorting behaviour observed in social insects colonies [DGF+91, BDT99]. We refer to an environment where artefacts have the shape of tuple spaces, namely, where agents are allowed to insert and retrieve tuples based on their content: this is a typical scenario of agent mediated interaction, and is general enough to support a wide range of applications, including stigmergy-based MAS [WOO07, PBS05]. Simulation results have been obtained using our stochastic framework developed on top of the MAUDE tool [Mau07]: for more details about MAUDE see Section 5.2. For convenience, the complete collective sorting specifications, including the stochastic simulation framework, can be found in Appendix A.

6.2.1 Problem Statement

A typical problem of tuple-space-based environments is that it is generally difficult to retrieve the tuples of interest when they could be inserted in any tuple space of the net. A possible solution is hence to allow agents to find tuples based on similarity, by grouping together similar tuples in the same tuple space while separating different tuples: in this way, if a tuple is found on a tuples space, similar ones are likely found in the same space later. Furthermore, an ordered environment allows for better batch processing of "items", e.g. for applying aggregation techniques, checking consistency, and so on.

Collective sorting amounts at providing an environment that offers a "back-

ground" sorting service. Given a set of N tuple spaces and a statically-defined clustering of tuples into N *kinds*, collective sorting amounts at moving tuples toward the fully-sorted situation where each space hosts only tuples of the same kind. Though, sorting should proceed in dynamic and unpredictable situations where user agents of the MAS keep interacting with tuple spaces, that is, moving, inserting, and dropping tuples. Therefore, the tuple space that will eventually aggregate a certain kind is not to be decided a priori: it should be rather implicitly and probabilistically selected as tuples start aggregating in one space rather than another due to the effect of multiple tuple movements—namely, in a self-organisation style. This approach is meant to tackle robustness (which is seen as more important than performance): we need sorting to be a property eventually emerging in spite of external interactions—of course, the more user agents keep altering the tuple configuration, the more resources should be devoted to sorting in order to converge.

In conformity to the architecture described in Section 3.2, the sorting task is hence assigned to a set of environmental agents, whose goal is to keep the environment ordered as much as possible: in this section we apply the proposed approach to provide a sound model of the behaviour of such agents. Taking inspiration from the similar problem of *brood sorting* [DGF$^+$91, BDT99] manifested in ant colonies, in step 1 we identify a possible model for the behaviour of environmental agents. In step 2, we discuss the outcomes of several simulations of system behaviour, showing that the solution is not completely adequate for it not always leads to full-sorting. In step 3, we hence tune the system model by introducing a mechanism in the style of *simulated annealing*: further simulations show the adequacy of the new model, and emphasise the behaviour of sorting during user agents interactions.

6.2.2 Identifying a Suitable Approach in Nature

Collective sorting 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 items, group and separate them.

In that context, solutions are typically searched in Nature, which is a rich source of simple but robust strategies. The segregation behaviour is already manifested by social insects in *brood sorting* [DGF$^+$91, BDT99]. When organising brood and larvae, ants are subject to the problem of grouping and keeping them separately 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 and their behaviour is modelled by two probabilities, the probability to pick up P_p and drop P_d an item: 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 picked.

The ant-based solution to the brood sorting problem is an intrinsically self-organised one: namely, ants are guided by spatially-local observations and are motivated by the only need of picking items up where concentration is low, and dropping them where it is higher: numerous such interactions make full sorting (i.e. the segregation pattern) emerge at the global level. Though the sorting performance is sub-optimal – it cannot compare with solutions based on global

observations – it is intrinsically robust: it promptly reacts to changes in the environment (e.g. new brood, larvae, or ants are dynamically added or removed), to faults like an environment split (e.g. a barrier splitting the ground in two parts), and to local malfunctioning (e.g. some ant behaving in a completely different way). Hence, it is interesting to seek for a solution for the collective sorting problem which is inspired by the ant-based solution to brood sorting.

However, as already discussed, the above solution should be significantly adapted, since the application scenarios of brood sorting and collective sorting have key differences. First of all, instead of being a continuous environment, our scenario features a set of N tuple spaces, each being a concentrated, conceptually unlimited bag of tuples. Secondly, our environmental agents do not likely move and carry tuples with them, but rather, for obvious performance reasons, they should reside in one of the N sites and send tuples away when needed. Finally, instead of perceiving items based on a range of locality, agents should be able to look for tuples in either the local tuple space, or a remote tuple space—the latter operation obviously being more expensive.

6.2.3 Step 1: Modelling Collective Sorting

We consider N environmental agents, also called *sorting agents*, each situated in a different site hosting one of the N tuple spaces: each agent is hence assigned to one tuple space, which is seen as the tuple space local to the agent—interacting with it is hence less expensive than with other tuple spaces. Similarly to ants, an agent performs a partial system observation, namely, an observation of the private tuple space (where the item is possibly picked up) and an observation of some remote tuple space chosen randomly (where the item is possibly dropped). If according to such observations it can be inferred that some tuple is better sent to the remote tuple space, then the agent removes the tuple locally and inserts it remotely. This observation-action cycle is executed with a fixed sorting agent rate r, whereas the global sorting agent rate would be $N * r$—which is the number of moving attempts per time unit. This scenario is depicted in Figure 6.8.

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

1. a remote tuple space R is drawn randomly;
2. a "uniform `rd`" operation is performed on L, yielding a tuple of kind K_L;
3. a "uniform `rd`" operation is performed on R, yielding a tuple of kind K_R;
4. if $K_L \neq K_R$ a tuple of kind K_R is moved from L (if any exists there) to R.

Uniform `rd` operation, also called `urd`, is the operation by which the agent reads any tuple from the tuple space—i.e., 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 occurring most, and hence, K is the best candidate so far for finally aggregating on that space. Accordingly, at the time third task is executed, the agent knows that space L is aggregating

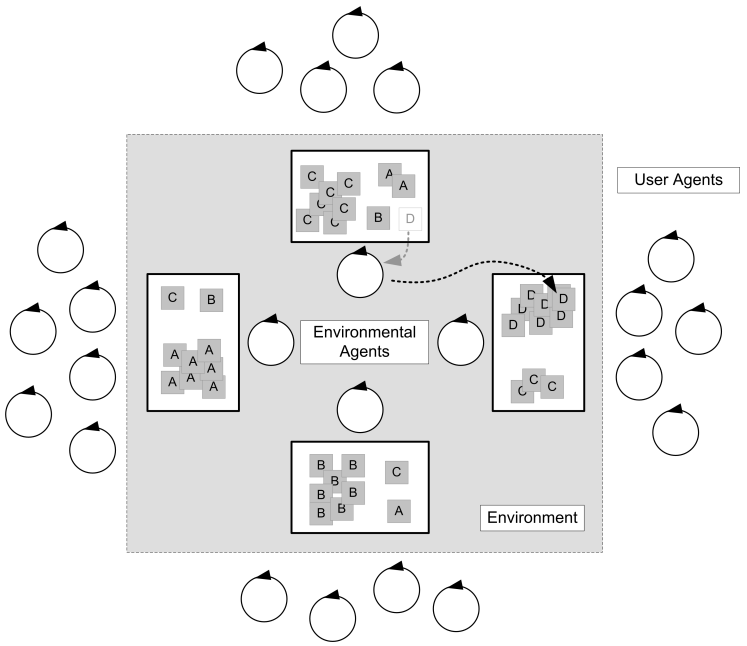

Figure 6.8: Basic architecture for collective sorting.

K_L while space R is aggregating K_R. The rationale of fourth task is hence that if K_R and K_L are different, then the agent can 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 correspondingly aggregate more.

This completes the description of a first candidate solution, which can be turned into any stochastic specification language [PRI07, Phi07]—we relied on the stochastic simulation library for the MAUDE term rewriting language discussed in [CGV07], though any other could be used.

6.2.4 Step 2: Simulating Collective Sorting

The observation and then the decision taken by the agent are affected by probability, hence the correctness of this distributed algorithm is to be checked by simulation, in order to verify, first of all, whether complete ordering is reached starting from any initial situation—even the most chaotic one. As an example, consider as initial configuration a very chaotic one, where each tuple space has the same number of tuples of any kind, and e.g. $N = 4$. We represent this system state by the syntax:

```
T1[25,25,25,25],T2[25,25,25,25],T3[25,25,25,25],T4[25,25,25,25]
```

expressing the fact that each tuple space `Ti` has 25 tuples of each kind (named `K1`, `K2`, `K3`, and `K4` in the following). An example of simulation trace is pictorially represented in Figure 6.9 (a), reporting the dynamics of the "winning" tuple in each tuple space—namely, the tuple that eventually aggregates there. Note that tuples reach their full aggregation level at different points in time, in mostly an

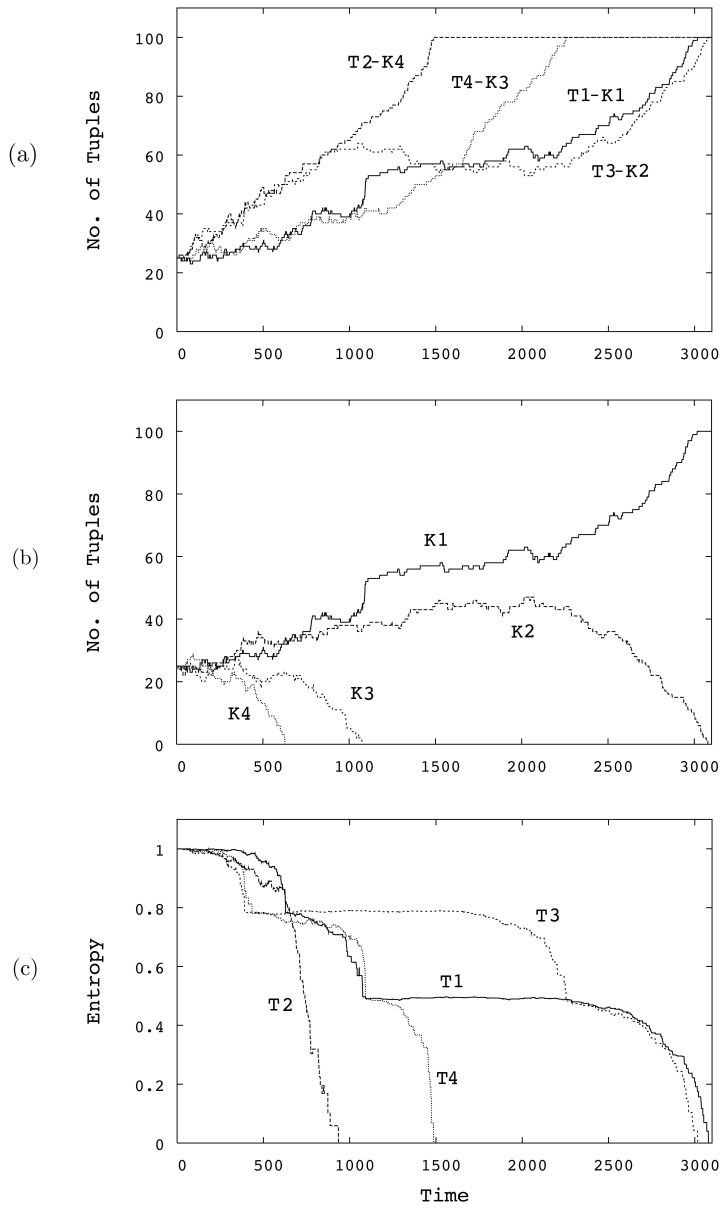

Figure 6.9: Charts of a simulation trace: (a) Winning Tuple; (b) Tuple space T1; (c) Entropy in each tuple space.

unpredictable way. The chart in Figure 6.9 (b) displays instead the evolution of the tuple space `T1` taken as an example: notice that only tuples of kind `K1` aggregate there despite its initial concentration was the same as other tuples. In particular, e.g., at some point around step 1000 there is a bifurcation which promotes aggregation of tuple kind `K1` instead of `K2`.

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 strategy we simulate is trying to increase the inner order of the system we expect the entropy to decrease to zero, as actually shown in Figure 6.9 (c). The entropy associated to a tuple space is computed in the standard manner [CGV07], considering the concentration of each tuple kind, and the total entropy is normalised so it ranges between 0 and 1. Each chart reports the number of protocol instances (moving attempts) executed by agents: supposing the single agent rate is 0.25, then the global agent rate is 1.0—there is an average of 1 simulation step per time unit—meaning that full sorting is reached after around 3000 time units. Other simulations performed with a different number of tuples and tuple spaces show similar qualitative results.

In general, the outcome of a simulation should highlight the system performance, but it can sometime show flaws in the design. In our case, though it first appeared that the proposed model always leads to complete sorting from any initial configuration of tuples, more thorough simulations show that there are certain stable states attracting the system trajectory and having positive entropy, that is, characterised by a non-complete degree of sorting. A state of this kind is called *local minimum* (for the entropy). An example of such a minimum is the following state, obtained by traces shown in Figure 6.10 (a) and (b):

```
T1[100,0,0,0],T2[0,69,0,0],T3[0,31,0,0]),T4[0,0,100,100]
```

Tuple kind `K2` is the only one aggregating in both spaces `T2` and `T3`, and at the same time, kinds `K3` and `K4` aggregate both 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. Moreover, one such state is an attractor, for simulations starting from states sufficiently near to it appear to converge back to this local minimum. This makes the strategy of environmental agents inadequate, and thus requires a tuning of the model in order to find a suitable solution.

6.2.5 Step 3: Tuning Collective Sorting

There is a main reason why the local minimum analysed above cannot be escaped: the strategy we developed does not explicitly avoid the case where the same tuple aggregates in two different tuple spaces. In fact, due to step 4 of the agent protocol, nothing is done when $K_L = K_R$. Hence, it can happen that a same tuple fully aggregates on two different tuple spaces, and dually, two remaining tuples aggregate in the same one as shown in the local minimum above.

These two issues can actually find a common solution by more carefully analysing the brood sorting problem for social insects. There, an ant takes an item and releases it where a new place is found where such an item has a

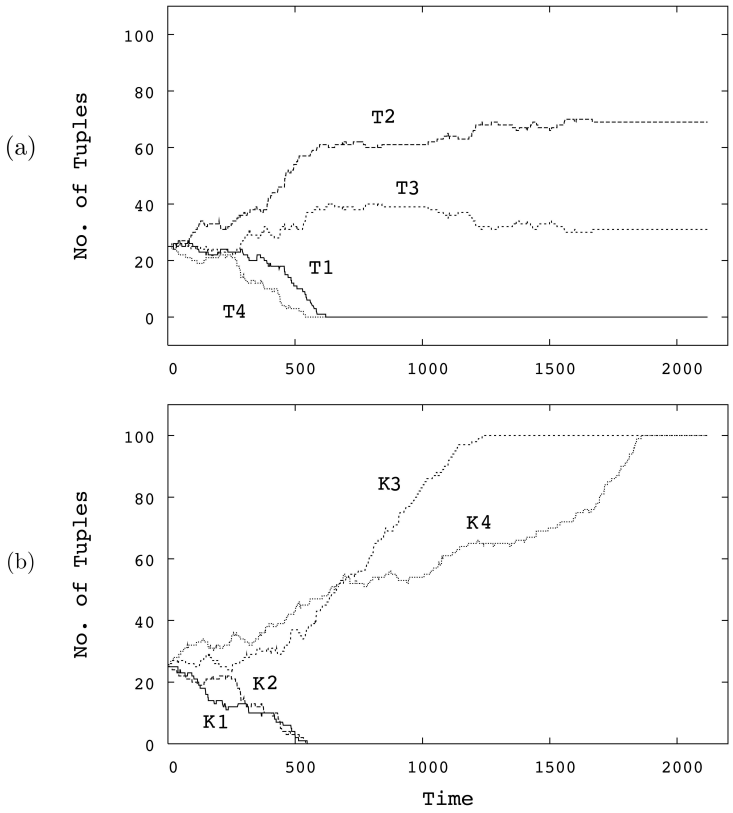

Figure 6.10: Charts of a simulation trace to a local minimum: (a) Tuple kind k2 aggregating in spaces T2 and T3; (b) Both kinds k3 and k4 aggregating in space T4.

greater concentration, expressed as quantity of brood over a unit of space. That is, implicitly the ant is able to compare the amount of brood with respect to a standard quantity, which in that specific case is represented by the amount of vacuum.

To implement a mechanism supporting this idea, we add to tuple spaces another kind of tuple called `noise`, which – for simplicity – we initially suppose to be constant throughout sorting. Now an observation by a uniform rd can also be "perturbed", yielding a tuple `noise`. As in previous model, if the local and remote observations are different, a tuple is moved anyway from the local space to the remote space. Though, if an observation is perturbed by reading `noise`, the correctness of moving is now probabilistically altered. However, the probability of picking a tuple in `T3` is expected to be higher than in `T2`, and this should promote tuples of kind `K2` leaving `T3` more quickly. As a result, this mechanism might be expected to globally result in complete sorting.

This mechanism would actually resemble the concept of *simulated annealing* [KGV83]. There, a perturbation is added to an optimisation algorithm in order to avoid the risk of finding non-optimal solutions: such a perturbation is initially high and is made fading 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? One possibility 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, whose optimal value would depend on the average occupation of tuple spaces during system execution [VCG07]: this situation is not appealing since we want our approach to work independently of the number of tuples in the system. What we actually look for is a fully-adaptive noise mechanism, where noise is initially very low, it increases as the system is approaching to a local minimum, and it decreases if such a minimum is escaped. In this way, we could expect the system performance to be only slightly affected if the system stays sufficiently far from local minima, and on the other hand, the noise production may become significant only in unfortunate cases where local minima are approached.

To achieve this result, we will manage noise as follows: *(i)* initially only one noise tuple occurs in each tuple space; *(ii)* each time two tuple spaces seem to aggregate the same tuple, noise is increased; and *(iii)* when some tuple is correctly transferred – without involving a perturbed observation due to noise – noise is decreased. Accordingly, we change the environmental agent design by relying on the following protocol:

1. a remote tuple space R is drawn randomly;
2. a uniform `rd` operation is performed on L, yielding a tuple of kind K_L;
3. a uniform `rd` operation is performed on R, yielding a tuple of kind K_R;
4. if $K_L \neq K_R \neq$ `noise` a tuple of kind K_R is moved from L to R;
5. if $K_L \neq K_R =$ `noise` a tuple of kind K_L is moved from L to R;
6. if `noise` $\neq K_R = K_L$ noise is increased by one in L;
7. if $\neq K_L \neq K_R \neq$ `noise` noise is decreased by one in L.

Now both K_L and K_R could be noise. Fourth and fifth task say that differences in observations 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. Sixth task increases noise if L and R are aggregating the same (non-noise) tuple $K_R = K_L$, and finally seventh task decreases noise if a non-perturbed transfer is actually executed.

Considering now the worst case of a symmetric local minimum:

```
T1[100,100,0,0],T2[0,0,50,0],T3[0,0,50,0],T4[0,0,0,100]
```

we expect that noise starts increasing in both tuple spaces `T2` and `T3`. At some point, movement of tuples `K3` will occur between `T2` and `T3` for some noise is observed. Because of a bifurcation effect, if either space `T2` or `T3` will have a greater concentration of tuples `K3` with respect to noise, that would cause more tuples to be transferred there, and that space will eventually fully aggregate tuples `K3`. Accordingly, the other tuple space will be emptied, it will loose noise tuples, and it will finally become target of tuples of kind `K1` and/or `K2`. This

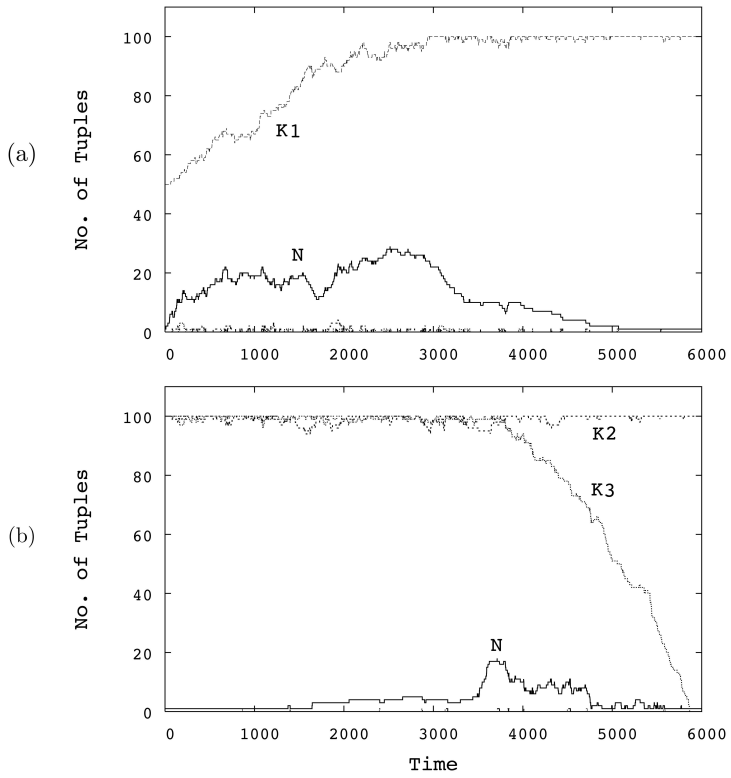

Figure 6.11: Charts of a simulation trace escaping from a local minimum: (a) Situation in space T2: winning tuple and noise in evidence; (b) Situation in space T1: kind K3 leaves the space.

is actually what can be observed from the traces in Figure 6.11 (a) and (b), showing how the local minimum is escaped in spaces `T2` and `T1`: in both cases we see that as noise tuples increase, the system escapes the local minimum configuration, and after that, noise tuples fade.

More simulations performed on 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; *(iii)* full sorting is always eventually reached; *(iv)* these results are independent from the number of tuple spaces (and kinds) N.

6.2.6 Evaluation of Reactiveness

Now that a promising solution is found it is interesting to get back to simulation: in this section we report the final results we obtained, and evaluate interesting system parameters to be used in subsequent steps of the MAS design.

A main reason why the collective sorting problem for tuple spaces has been solved using a self-organising approach is to tackle unpredictable interactions with the environment. The 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 known

a priori, hence, sorting should be able to react to changes of the surrounding conditions, in a fully-adaptive way. What we show in this section is how the ratio between user agent rate and sorting agent rate, called *perturbation/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 *change rate* for user agents, that is, at that rate a user agent randomly moves a tuple from one space to another. Starting from an initially sorted configuration of tuples (400 tuples, $N = 4$), depending on that rate we easily expect that either *(i)* full-sorting is almost always maintained, *(ii)* a certain level of (partial) sorting can be maintained, and *(iii)* the system becomes more and more unsorted as time passes. The evolution through such situations is reported in Figure 6.12, where each chart provides the evolution of entropy in time at a different rate.

As shown in the summary Figure 6.13, the key factor is the perturbation/sorting ratio, which gives a clear indication of the adequacy of sorting resources, in terms of the maximum level of entropy they can guarantee. Along with the environmental agent behaviour identified, the bound to perturbation/sorting ration set to 0.5 is a critical system parameter that only simulation could reveal, and that should be exploited in subsequent design steps—which we do not discuss here. E.g., a form of load-balancing is required to be sure the resources of sorting are adequate with respect to the current degree of disorder, and can self-adapt to it—increasing on a by-need basis and then decrease. Techniques related to the prey-predator approach as studied e.g. in [GVO07a, GVCO07] could be evaluated in the subsequent steps of design.

6.3 Plain Diffusion

In this section we describe a self-organising strategy for achieving a plain diffusion behaviour: the solution is analysed according to the methodological approach described in Chapter 4 and the PRISM tool described in Section 5.3. We decided to consider the case study of plain diffusion mainly for two reasons: on the one hand, our solution to plain diffusion is very simple but exhibits all the key features of self-organisation, hence allowing us for an effective explanation of our methodological approach; on the other hand, plain diffusion is a key element of many chemical and biological phenomena, e.g. in chemotaxis [Mur02] or in pheromone diffusion in ant colonies [CDF+01]. Furthermore, plain diffusion has been recognised as an important design pattern for self-organising artificial systems allowing to produce gradients and averaging quantities [BCD+06, GVO07a]: indeed, despite its simplicity, the diffusion mechanism plays a key role in every digital pheromone-based application, e.g. in the case of Autonomous Guided Vehicles [PBS05, WSHL05], and in many distributed systems strategies such as in load-balancing [CDU06].

6.3.1 Problem Statement

Consider a networked set of nodes having an arbitrary topology and where each node is labelled with a non-negative quantity. We want to devise a strategy that from an arbitrary initial state eventually evolves into a dynamical state where each node is labelled with the same quantity. In particular, we want the strategy to be self-organising, i.e. where each node is autonomous and transfer quantities

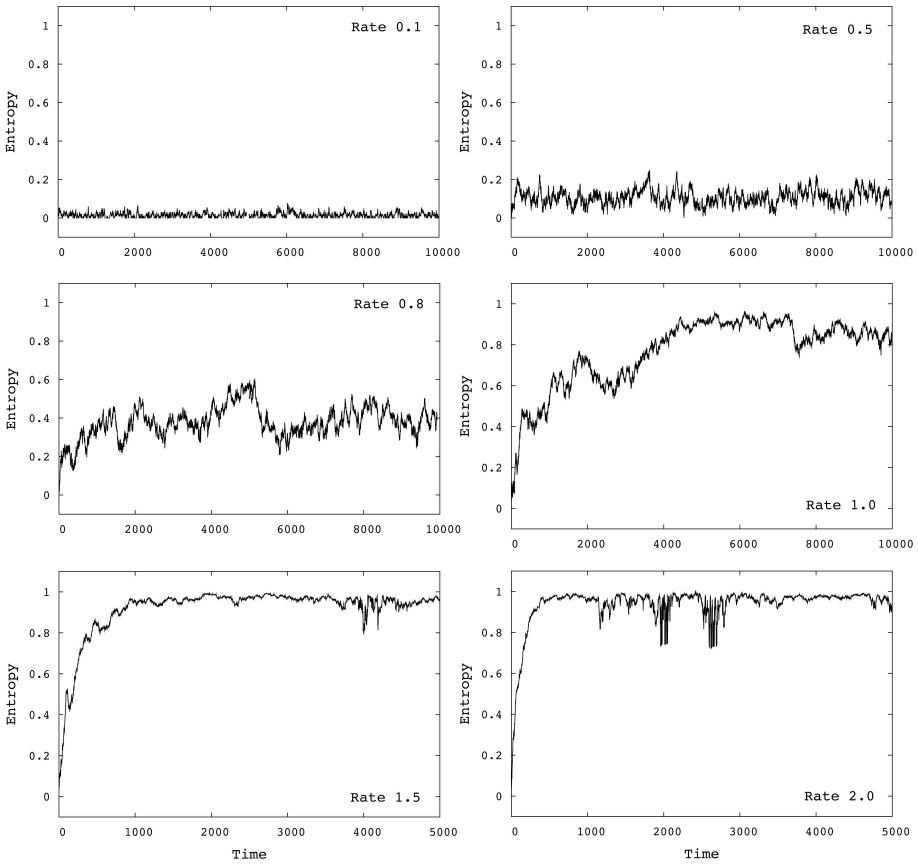

Figure 6.12: Evolution of entropy with different perturbation/sorting ratio

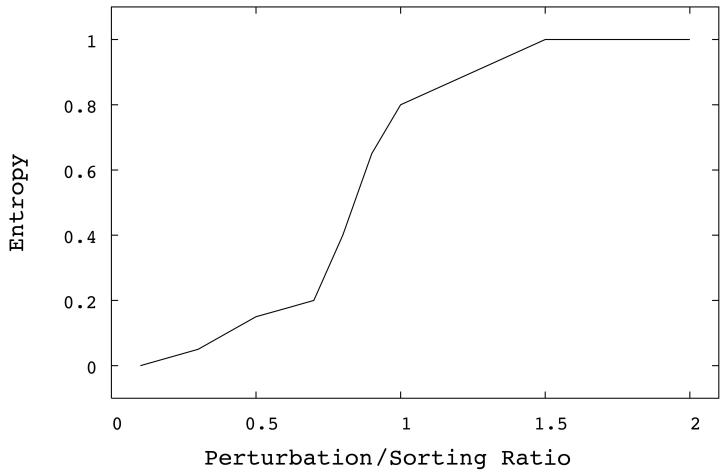

Figure 6.13: Maximum entropy depending on perturbation/sorting ratio

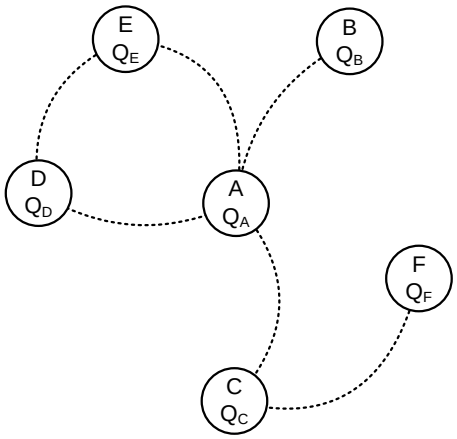

Figure 6.14: The reference network topology: this network is interesting because it exhibits features found in real topologies such as cycles, hubs and nodes with limited connectivity.

according to local knowledge: in this way our strategy will be independent from the network topology, the distribution of quantities and the overall amount of quantities. A node knows the identities of neighbouring nodes and the local quantity, while it has no information about network size and quantities in other nodes.

In order to evaluate our proposal, we have to test it against an actual instance of network. Specifically, we choose the 6-nodes topology displayed in Figure 6.14 since it exhibits features commonly found in actual networks: these features include cycles, hubs and nodes with limited connectivity. We believe that the 6 node topology is enough large to clarify the approach, which is our main objective here: scalability issues are discussed later in Section 6.3.6. For the sake of clarity, from now on when considering system states we use the compact notation $((A, Q_A), .., (F, Q_F))$. The required dynamics for strategy are the following: each node i having a local quantity Q_i have to eventually reach a state where $Q_{avg} = \frac{\sum_{i=1}^{N} Q_i}{N}$. It is worth noting that because of the limited knowledge available to each node the strategy will never converge since it has no criteria regarding the halting condition: the best result we can achieve is to establish a dynamic equilibrium close to the average value.

6.3.2 Modelling Plain Diffusion

In this section we provide a solution to the previously described problem with respect to the agents and artefacts meta-model and architectural pattern. The mapping between the network and the A&A equivalent architecture is straightforward, and basically amounts to replace a node with an artefact and its environmental agent, see Figure 6.15. More precisely,

- each node is represented by an artefact acting as a data repository;
- each artefact is managed by a dedicated environmental agent;

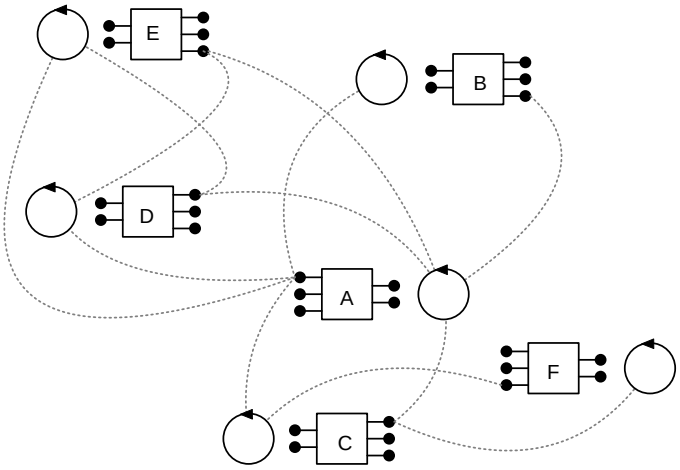

Figure 6.15: The picture shows the agents and artefacts diagram equivalent to the previous network topology: it is worth noting that since agents/artefacts relation is not symmetric, two association are required making the diagram appears more cluttered.

- the connections represent neighbourhood information that can be encoded either within environmental agents or artefacts: both approaches are viable thus modelling different constraints.

The constraints and possible actions then becomes

- an environmental agent can put/remove an item only in the local artefact and remote artefacts in its neighbourhood;
- an environmental agent knows the number of items contained within the local artefact;
- an environmental agent knows a limited set of artefacts, we call neighbouring artefacts;
- the agent does know neither the overall number of artefacts nor the the overall number of items in the system.

In our approach the first step consists in finding an existing pattern: we recognise that the problem can actually be assimilated to Plain Diffusion, which has been recognised as an important design pattern in [BCD$^+$06, GVO07a]. However, the solution proposed in [BCD$^+$06] requires the exchange of information between nodes, which is in contrast with our requirements. Hence, we propose a different approach for achieving plain diffusion: to this purpose, we start considering the two nodes network $((A, 20), (B, 10))$ where B has twice as many items as A. Since they actually do not know the number of items of the other node the only way to reach an equilibrium is via dynamic exchange criteria: unfortunately if nodes exchange items at the same speed the balance remains unchanged. Hence, as a first proposal we suggest that nodes send items at a speed proportional to the number of items possessed. We notice that this basic strategy do not work

for the network $((A, 20), (B, 10), (C, 20))$ where A and C are connected only to B: indeed, we expect a gradient from this situation since B receives items both from A an C. In order to compensate for this gradient, the working rate of each agent should be proportional not only to the number of items but also to the number of neighbouring nodes. Hence, the formula for the agent send rate r_i becomes

$$r_i = \frac{Q_i * S_i}{P} \tag{6.2}$$

where Q_i is the local number of items, S_i is the local star, i.e. number of neighbouring nodes, and P is a global parameter that scales the overall workload.

From the requirements and the basic strategy we now provide a formal model using the PRISM modelling language: the whole specification is listed in Figure 6.16. Since PRISM language allows the definition of stochastic transition systems [PRI07], we have to reinterpret the system dynamics in terms of transitions. To the purpose of our model, in this case, we abstract from artefacts details: since in the plain diffusion model we are only interested into artefacts content, this information can be encoded in a simple variable. Conversely, agents are encoded in modules, that is, a collection of transitions: hence, agents manipulate local and neighbouring artefacts by simply modifying the corresponding variable. With respect to the topology defined in Figure 6.14, the definition of the environmental agent A is

```
module agentA
[] tA > 0 & tB < MAX & tC < MAX & tD < MAX ->
rA : (tA'=tA-1) & (tB'=tB+1) +
rA : (tA'=tA-1) & (tC'=tC+1) +
rA : (tA'=tA-1) & (tD'=tD+1) +
rA : (tA'=tA-1) & (tE'=tE+1);
endmodule
```

where `tA` is the local artefact, `tB`, `tC`, `tD` are neighbouring artefacts, `rA` is the rate of the transition defined by `rA = tA / base_rate`. Each transition models the motion of an item from the local artefact to a neighbouring one: the choice between neighbours is probabilistic and in this case all the transitions are equiprobable. It is worth noting that the rate formula does not explicitly take into account the number of neighbouring nodes: indeed, this factor is implicitly encoded in the transition rules. Since the model is interpreted as a Markov Chain the overall rate is the sum of all the transitions rates: in the previous code sample we have four top-level possible transitions with rate `rA`, hence the overall working rate of `agentA` is `4rA`. The definition of the other agents is very similar to the one of `agentA` but for the number of neighbouring artefacts.

6.3.3 Simulating Plain Diffusion

In order to qualitatively evaluate the dynamics of the system, in this section we run some simulations: PRISM allows the execution of simulations directly from the formal specification as long as we provide values for all the parameters. In our model the only parameter is the `base_rate`: this parameter allows the tuning of the system speed according to deployment requirements. Since at the moment we are not interested in performance issues we set it to the arbitrary

```
ctmc

const int MAX = 54;
const double base_rate = 100;

formula rA = tA / base_rate;
formula rB = tB / base_rate;
formula rC = tC / base_rate;
formula rD = tD / base_rate;
formula rE = tE / base_rate;
formula rF = tF / base_rate;

global tA : [0..MAX] init 14;
global tB : [0..MAX] init 0;
global tC : [0..MAX] init 8;
global tD : [0..MAX] init 16;
global tE : [0..MAX] init 12;
global tF : [0..MAX] init 4;

module agentA
[] tA > 0 & tB < MAX & tC < MAX & tD < MAX ->
rA : (tA'=tA-1) & (tB'=tB+1) + rA : (tA'=tA-1) & (tC'=tC+1) +
rA : (tA'=tA-1) & (tD'=tD+1) + rA : (tA'=tA-1) & (tE'=tE+1);
endmodule

module agentB
[] tB > 0 & tA < MAX ->rB : (tB'=tB-1) & (tA'=tA+1);
endmodule

module agentC
[] tC > 0 & tA < MAX & tF < MAX->
    rC : (tC'=tC-1) & (tA'=tA+1) + rC : (tC'=tC-1) & (tF'=tF+1);
endmodule

module agentD
[] tD > 0 & tA < MAX & tE < MAX->
    rD : (tD'=tD-1) & (tA'=tA+1) + rD : (tD'=tD-1) & (tE'=tE+1);
endmodule

module agentE
[] tE > 0 & tA < MAX & tD < MAX ->
    rE : (tE'=tE-1) & (tA'=tA+1) + rE : (tE'=tE-1) & (tD'=tD+1);
endmodule

module agentF
[] tF > 0 & tC < MAX ->rF : (tF'=tF-1) & (tC'=tC+1);
endmodule
```

Figure 6.16: The PRISM specification of the plain diffusion strategy for the reference 6-node network topology: it is worth noting that each module represents a node in the network and the respective environmental agent.

value of 100. We consider now a few scenarios modelling extreme deployment scenarios to evaluate the robustness and adaptiveness of the solution.

The first instance we consider has all the items clustered into a single node, specifically node A, the hub: using the compact notation the system initial sate is ((A, 600), (B, 0), (C, 0), (D, 0), (E, 0), (F, 0)). As it can be observed from Figure 6.17, all the nodes eventually reach the average value of 100 and then stay close to it: in particular, the node F requires more time to reach the value because it is two hops far from node A, while all the other nodes are just one hop far.

The next instance we consider is the one having all the items clustered into the peripherical node F: specifically, the system initial state is ((A, 0), (B, 0), (C, 0), (D, 0),(E, 0), (F, 600)). As it can be observed from Figure 6.18, all the nodes eventually converge to the average value of 100: in particular, node C converges quickly because it is one hop far from the node F, while all the other nodes are two hops far. It is also worth noting that before node C reach dynamic equilibrium it goes over the average value: this phenomenon is due to the fact that node A works many times faster than node C which slowly diffuses items to neighbouring nodes. With respect to the previous instance, this configuration requires almost twice as much time to establish a dynamic equilibrium: such a big variance depends on the fact that while on the previous instance the items were clustered in a node with 4 neighbours, in this instance the items were clustered in a peripherical node with only one neighbour, causing a bottleneck.

The next instance we consider is the one having items spread across the nodes, specifically ((A, 50), (B, 150), (C, 200), (D, 0),(E, 50), (F, 150)). As it can be observed from Figure 6.19, all the nodes eventually reach the average value of 100 and stay close to it: since this configuration was more ordered than the previous ones it reaches dynamic equilibrium faster.

The next instance we consider models the situation where a node is dynamically added to the network: specifically the initial state is ((A, 120), (B, 120), (C, 120), (D, 120), (E, 120), (F, 0)) where the nodes from A to E have the same number of items and F is the newly-added node. As it can be observed from Figure 6.20, the nodes move from the average value of 120 to the new average value of 100 due to the connection of a new node: it is worth noting that the speed of the adaptation is strongly dependent on the number of connections of the new node, but also weakly dependent on the network topology.

Since in all the simulated scenarios the strategy seems to behave properly we now move to the verification of the desired properties.

6.3.4 Verifying Plain Diffusion

The verification process consists in testing whether the properties of interest hold or not: this process is performed relying on stochastic model checking techniques [KNP07, RKNP04]. As anticipated in Section 5.3, model checking techniques suffer from the state explosion problem: considering an instance of the same size as done for the simulations it is just unfeasible. Hence, we consider a far smaller instance, specifically, the system instance having 36 items and expecting an average value of 6 items per node: in this section, we always refer to the initial configuration ((A, 4), (B, 0), (C, 10), (D, 12), (E, 6), (F, 4)). Although being a small instance, it is already computationally intensive:

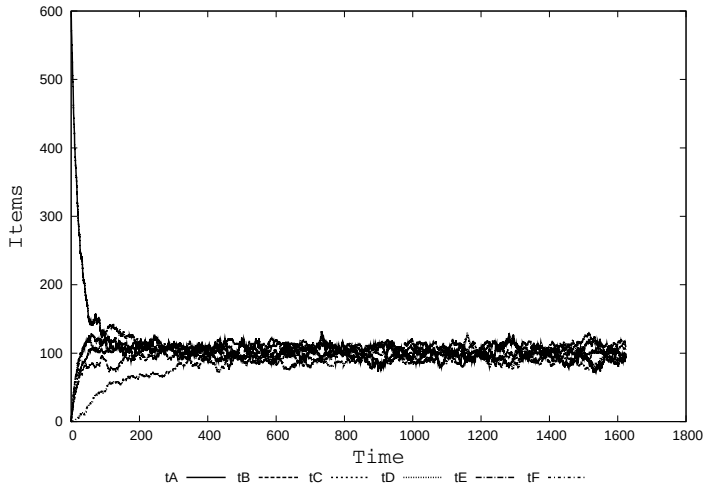

Figure 6.17: The evolution of the instance ((A, 600), (B, 0), (C, 0), (D, 0), (E, 0), (F, 0)): as it can be noticed the node F converges more slowly because it is two hops far from node A, while all the other nodes are just one hop far.

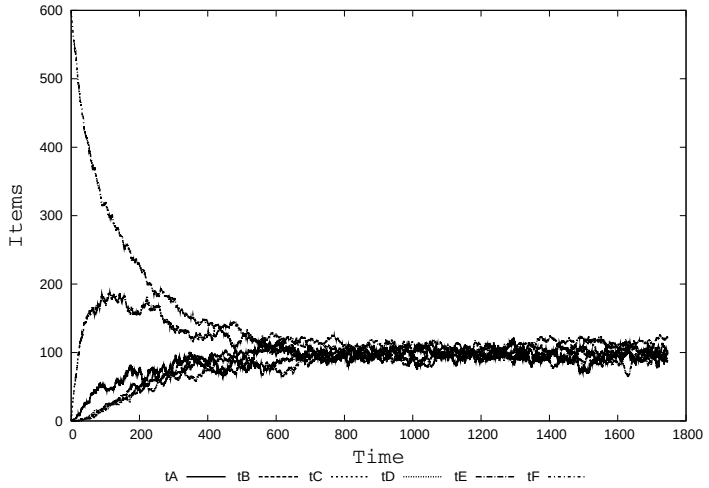

Figure 6.18: The evolution of the instance ((A, 0), (B, 0), (C, 0), (D, 0), (E, 0), (F, 600)): as it can be noticed, node C converges more quickly because it is one hop far from the node F, while all the other nodes are two hops far.

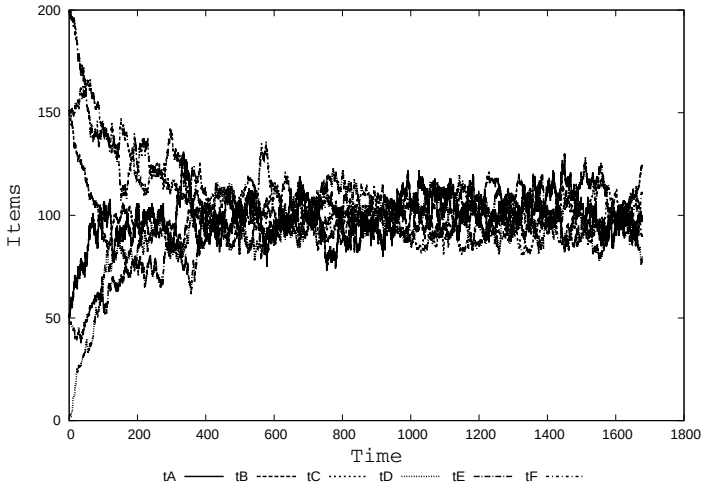

Figure 6.19: The evolution of the instance ((A, 50), (B, 150), (C, 200), (D, 0), (E, 50), (F, 150)): as it can be noticed all the nodes eventually converge to the average value.

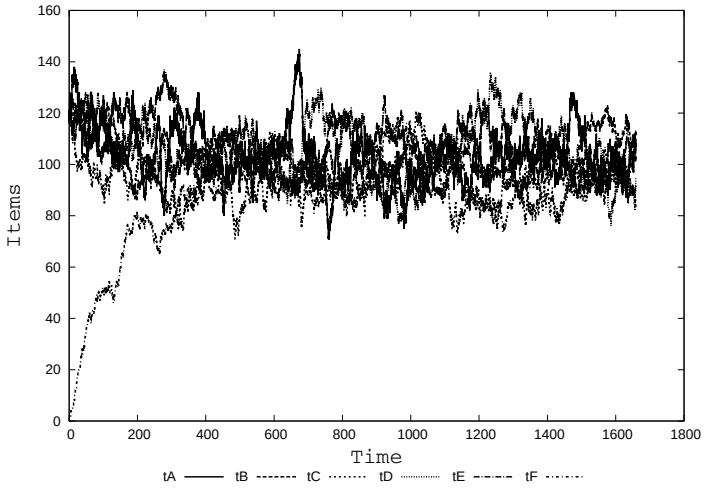

Figure 6.20: The evolution of the instance ((A, 120), (B, 120), (C, 120), (D, 120), (E, 120), (F, 0)) modelling the dynamic connection of a new node: as it can be noticed, the average value is moved from 120 to 100 and all the nodes eventually reach the new average value.

specifically, the instance is defined by 749398 states and by 7896096 transitions, and it takes about 15 seconds just to compile the model using the PRISM Hybrid Engine[2].

We are here interested in verifying a few system properties: the first property is about the quality of the strategy with respect to its goal, i.e. produce an average value. Since the strategy modelled is stochastic, we can provide a statical characterisation of this property: in particular, we can devise the probability[3] distribution for a node to be in a specific state, that is, being assigned a particular value. In Continuous Stochastic Logic, this property is equivalent to the statement *"Which is the steady-state probability for the variable X to assume the value Y?"*: since we want a probability distribution and not a single value, we run an experiment where Y span the range 0..36. Using the PRISM syntax, this property translates to `S=? [tA=Y]` where `S` is the steady state operator, `tA` is the variable containing the actual value and `Y` is the unbounded constant ranging in the interval 0..36. The chart in Figure 6.21 displays the results of the model checking experiment over the specific node `tA`: the experiment took about 3 hours using the Hybrid Engine and Jacobi iterative method. Experiments over the other nodes showed an identical probability distribution, providing evidence of the correctness of the strategy and the independence from initial node value and placement in the network. As we expected, the maximum probability peak is in correspondence of the average value, although being only the 17.59%: the system has a probability of 49.68% of being in the range 6 ± 1, while has a probability of 73.91% of being in the range 6 ± 2. It is worth noting that the chart is not symmetrical with respect to the average value because of the asymmetry of the admissible range.

The next test we consider is about the time for reaching the average value: specifically, considering the previous test instance ((A, 4), (B, 0), (C, 10), (D, 12), (E, 6), (F, 4)) we evaluate the time for tB, having an initial value of 0, to reach the average value of 6. Since CSL does not provide a time operator, we still have to reason in terms of probability: hence, the query is *"Which is the probability for the node tB to be equal to 6 within Y time steps?"*. Using the PRISM syntax, this formula becomes `P=? [true U<=Y tB=6]` where `P` is the probability operator, `true U<=Y` means to be verified in the next `Y` time steps, and `Y` is an unbounded constant. The chart in Figure 6.22 displays the results of model checking for `Y` spanning the range 0..600 at step size equal to 10. Although valid only for node tB, this type of chart is very useful since it provides a solid basis for tuning and in-depth performance evaluation.

6.3.5 Tuning Plain Diffusion

Since the modelled strategy already exhibits global dynamics compliant to our requirements there is no need for tuning but for performance issues. In the simulation and verification sections we assumed the arbitrary value of 100 for the `base_rate` parameter: if we want to give guarantees with respect to deployment conditions we have to consider a meaningful parameter value. For example, we

[2]The computer used for all the simulations and verifications has the followings processing capabilities: CPU Intel P4 Hyper Threading 3.0 GHz, RAM 2 GB DDR, System Bus 800 MHz.

[3]It is worth noting that model checking techniques provide exact probability values rather than estimation as for simulation.

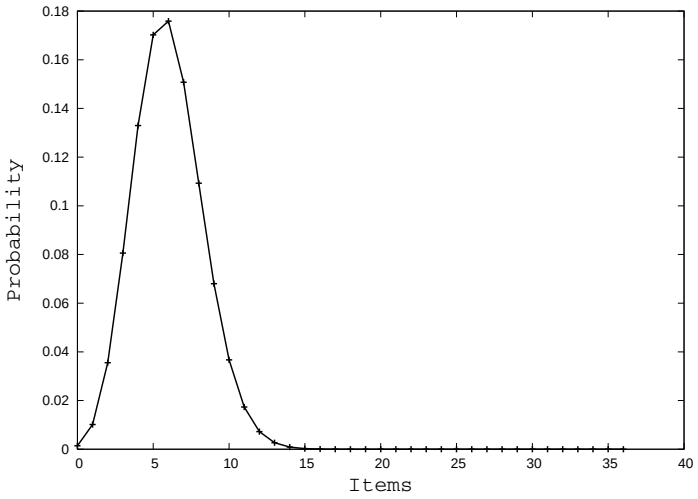

Figure 6.21: The chart displays the distribution of the probability for a node to contain a specific number of items: further experiments show that the chart is the same for each node. Notice that the probability peak is located in correspondence with the average value, that is 6. The chart is not symmetrical due to the asymmetry in the range of values.

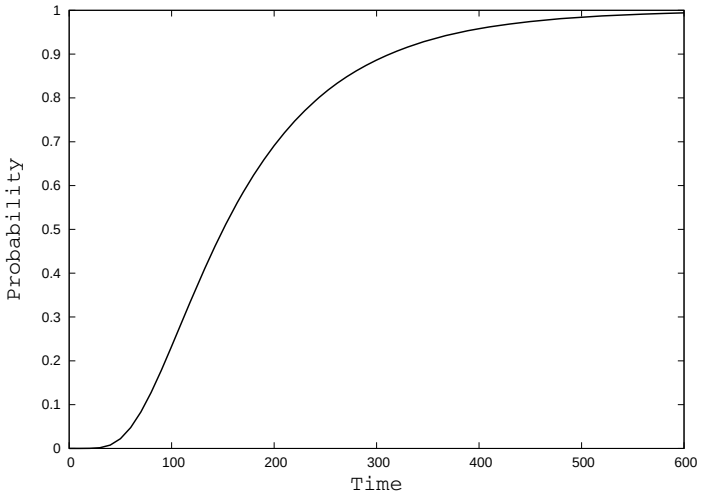

Figure 6.22: The chart displays the probability for the node tB to reach the average value from zero: this chart is very useful for the tuning process since it provide a solid basis for performance evaluation.

can establish a requirement for all the nodes to have a probability greater or equals to 90% to reach the average value within T time units under specific workloads conditions: then, by performing several tests we can devise the actual value for the `base_rate` parameter that meets the target requirement.

In order to evaluate the performances, we consider the worst scenario which consists in all the items clustered in the peripherical node F: specifically the initial state is ((A, 0), (B, 0), (C, 0), (D, 0), (E, 0), (F, 36)). As time constraint, we want the system to reach dynamic equilibrium before 200 time units, while for the probability constraint we set the lower bound to 90%. Hence, by adjusting the value of `base_rate`, we have to test the following property for all nodes: *"Is the probability of reaching dynamic equilibrium condition within 200 time units greater or equals to 90% ?"*. Using the PRISM syntax this property becomes `P>=0.9 [true U<=200 tA=6]` for the node `tA`.

We start by considering the farthest node tB, since the property must hold for all nodes, testing first the farthest node saves us a lot of computation: as a first exploration, with respect to the property `P=? [true U<=200 tB=6]`, we plot the probability values within the range 10..100. As it can be observed from chart in Figure 6.23, the trend of the probability is non-linear: nonetheless, we can guess that the desired value for `base_rate` lies in the range 30..40. Hence, we repeat the experiment zooming in the range 30..40 with unitary step: the results are plotted in Figure 6.24. As we can observe the value for `base_rate` that produce the probability value closest to 90% is 37. Hence, we fixed this value for the base rate and tested if the property is also satisfied for the other nodes: since the node tB is the farthest one, as we expected the property is satisfied also for node tA, tC, tD, tE, but not for tF. Hence, we investigated the property for node tF and results are plotted in Figure 6.25: as can be observed the property holds for node tF when `base_rate=31` and not 37. This phenomena can be explained by the fact that we requested a 90% probability for the nodes to reach the target value: hence, there is a 10% probability for each node not to have reach the target value, which summed up in the source node cause this delay.

Hence, from all the experiment we obtain that the value for the `base_rate` parameter to satisfy the property of $\geq 90\%$ probability of convergence within 200 time units is 31: our first guess was that the bottleneck should have been the farthest node, conversely it proved out to be the source node. The parameter `base_rate=31` means that initially the system send about 1 item per unit of time per connection, and decreases while converging to about 1 item per 5 units of time per connection.

Concerning the number of transfers, because of the very nature of the strategy, it is not trivial to evaluate the number of transfers: indeed, they depends upon neighbourhood, local number of items which changes over time. If there was no dependence from the number of neighbours, assuming a static topology, the global transfer rate could have been evaluated by the simple formula

$$r_g = \frac{Tot.Items}{baserate} \tag{6.3}$$

Conversely, because of the dependence from number of neighbours the instan-

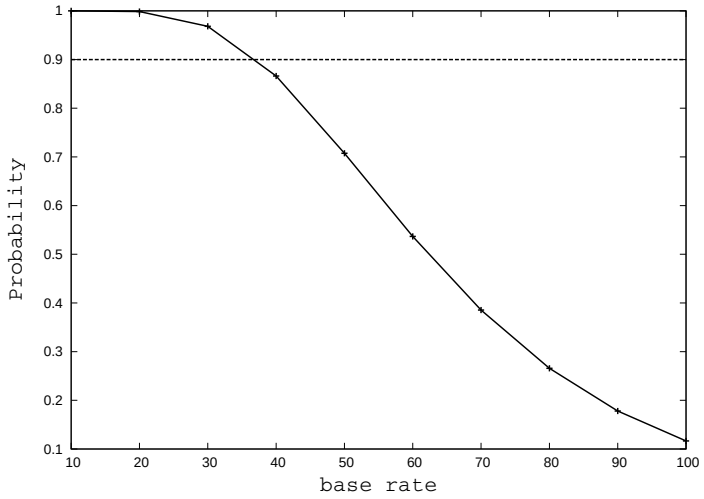

Figure 6.23: The chart displays the probability values for the node tB according to the CSL formula $P =?\ [true\ \ U <= 200\ \ tB = 6]$: we can guess that the desired value is within the range 30..40.

taneous global rate is given by the formula

$$r_g(t) = \sum_{i=1}^{N} \frac{Items_i(t) \times Neighbours_i}{baserate} \tag{6.4}$$

where $i = 1..N$ is the node selector: this formula explicitly depends from time, and requires either a simulation or real data to be computed.

6.3.6 About Scalability of the Strategy

In this section we briefly consider scalability of the proposed strategy: specifically, we are interested here in the scalability with respect to the total number of items within the network. The initial state for each trial consists in all the items placed within node A, i.e. A=X, B=C=D=E=F=0. Since we are using model checking tools to perform such evaluation, we will limit the number of items to 48: furthermore, to avoid fractional average values we consider only quantities multiple of the number of nodes, specifically the set of values $\{6, 12, 18, 24, 30, 36, 42, 48\}$.

The property we are interested in evaluating is the time required to converge with a probability $\geq 90\%$: unfortunately PRISM does not allow time queries, hence, we have to run several experiments in order to profile probability of convergence with respect to time. The property we want to verify can be translated into `P=? [true U<=T tA=X/6]`, where T represents the time value and X is the total number of items. Notice that we do not have a formula for describing global convergence, hence, we have to provide a formula for convergence of a

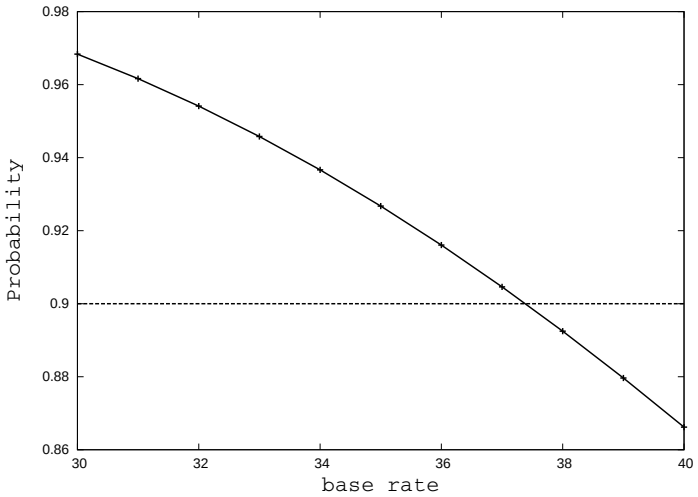

Figure 6.24: The chart displays the probability values for the node tB according to the CSL formula $P =?\ [true\ \ U <= 200\ \ tB = 6]$ within the range 30..40 of base rate.

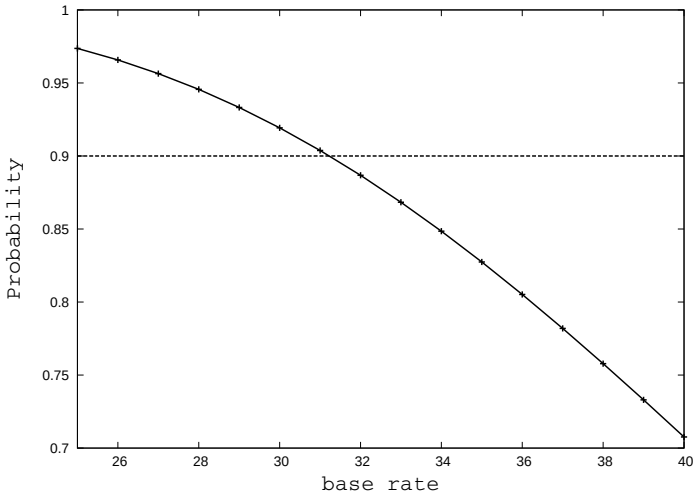

Figure 6.25: The chart displays the probability values for the node tF according to the CSL formula $P =?\ [true\ \ U <= 200\ \ tF = 6]$ within the range 25..40 of base rate.

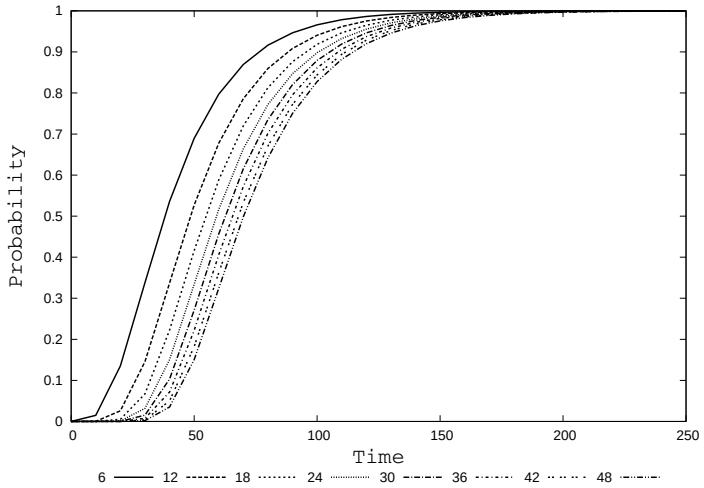

Figure 6.26: The chart displays the probability values for the node tA to reach the average value with respect to time and from different initial states.

single node: we consider the source node since, from previous experiments, we know this is the most restrictive constraint. The charts in Figure 6.26 and 6.27 display the probability for the node tA to reach the average value with respect to time: from these charts it is possible to extrapolate, although with some approximations, the time values for each initial state having a probability $\geq 90\%$ reach the average value. Data extracted from previous charts is depicted in Figure 6.28: as it can be easily noticed the trend is sub-linear. It is worth to point out that these results have been obtained with respect to the specific network topology and initial distribution of items: hence, the characterization provided here is far from being complete, although we are confident to obtain similar results with different distributions of items. Conversely, different topologies may heavily affect the scalability of the approach.

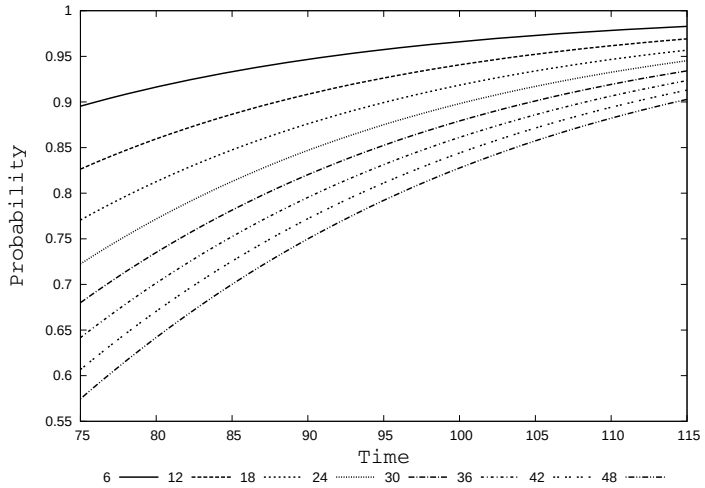

Figure 6.27: The chart displays the probability values for the node tA to reach the average value with respect to time and from different initial states: this chart zoom into the range [75..115] with a finer time step.

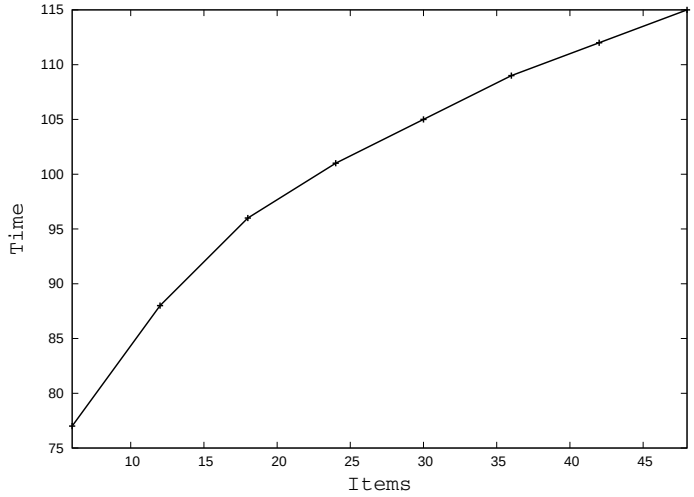

Figure 6.28: The chart represents the scalability with respect to the initial number of items contained in the node tA: it is worth noting that the trend is sub-linear. Since, the results depend on the network topology and distribution of items, further investigation is needed to provide a complete characterization of the strategy.

Chapter 7

Related Works

While there is plenty of literature about the analysis of self-organising and emergent mechanisms, the interest in engineering aspects grew only recently: indeed, the first edition of the International Workshop Engineering Self-Organising Applications (ESOA) dates back to 2003, the first edition of the International Conference Self-Adaptive and Self-Organising Systems (SASO) dates back to 2007, and the first issue of ACM Transactions on Autonomous and Adaptive Systems dates back to 2006. In this chapter we only summarise those works about self-organising systems that are strictly related to methodological aspects, including design patterns and formal techniques. Unfortunately, providing a broad survey about these topics is out of the scope of this thesis and would require a dedicated effort because of the amount of material: hence, we consider here only those design patterns, AOSE methodologies and formal tools that have been applied to the engineering of self-organising systems. Similarly, concerning the case studies there is a whole literature that would be worth discussing: however, since solving the case studies is not the focus of this thesis, we will not discuss those aspects here, though a several related works are cited and discussed along each case study.

7.1 Design Patterns for Self-organising Systems

Design pattern literature flourished in the last decade with the object-oriented paradigm [GHJV95]: concerning multiagent systems, there are several contributions, but little attention has been payed to self-organising systems. To our knowledge, beyond our contributions, there exist only two works in that specific context [BCD+06, DWH07]. It is worth noting that there is plenty of literature about applications of self-organising systems inspired by biological systems: unfortunately, despite the great insights presented in works such as [BDT99, CDF+01], none of them encode successful solutions in a reusable form, and takes a lot of experience for a computer scientist to extract the actual strategy. Conversely, in [BCD+06, DWH07] the authors successfully establish a mapping between existing natural system and its counterpart in computer science, easing the designer task: in particular, the authors' goal is to facilitate the adoption of biology-inspired ideas in distributed systems engineering.

In [BCD+06] the authors proposed several design patterns (i) Plain Diffusion,

(ii) Replication, (iii) Stigmergy, (iv) Chemotaxis and (v) Reaction-Diffusion: in particular, the authors present first the simpler ones, i.e. Plain Diffusion, Replication and Stigmergy, and then discuss the composite ones, i.e. Chemotaxis and Reaction-Diffusion. We have adopted the same approach because it allows a better organisation of patterns and identification of interplays between simple patterns: indeed, the advantages of using simple patterns individually is often overlooked. Particularly relevant to our work is the plain diffusion pattern, though we provided an alternative solution with respect to [BCD+06]. Concerning the pattern schema, the authors rely on the one provided in [GHJV95], missing the chance to better highlight key features for self-organisation. Concerning the applications, the authors provided a thorough discussion: in particular, they focussed on networks problems, namely, Power Optimization in MANETs, Unstructured Overlay Topology Management, Structured Overlay Topology Management.

In [DWH07], the authors provide a thorough description of two complex patterns about decentralised coordination mechanisms, namely, gradient fields and market-based control: in particular, patterns are described at the conceptual level, hence there is not identification of actual software components. Concerning the pattern granularity, the patterns described in [DWH07] are quite coarse and it is possible to identify finer patterns, organising the patterns in a hierarchy. Concerning the pattern schema, the authors rely on an existing pattern schema non-specific for self-organising systems. The work is contextualised with respect to the case study of Autonomous Guided Vehicles (AGV) [DW07, DWH07, WSHL05].

7.2 AOSE Methodologies for Self-organising Systems

There are few attempts to devise methodological approaches for the engineering of self-organising systems with emergent properties, and only one of them is a comprehensive methodology.

In [DW07, DWSHR05], the authors present a specific approach called equation-free macroscopic analysis. The approach allows the investigation of macroscopic properties of systems specified in terms of micro components without the need to devise evolution equations: required data is extracted from short simulations at micro level and combined with standard numerical techniques to perform system analysis. Quoting from [DWSHR05]

> Simulation measurements are analysed, but, in contrast to mere observation, the numerical analysis algorithms acquire the results themselves by steering the simulation process towards the algorithms goal. The advantage is that the results are calculated on the fly and only those simulations are executed that are actually needed to obtain a specific result. The results are therefore of equal scientific value as the equation-based analysis, while reducing the computational effort drastically compared to mere simulations.

This approach could be considered an hybrid between purely formal and purely empirical, combining advantages of both worlds: the outcome of this approach

is similar to the outcome of our simulation and verification stages. It is worth noting that the approach has been applied in an industrial case study about Autonomous Guided Vehicles (AGV) [DW07, DWSHR05, WSHL05].

From what it concerns more comprehensive approaches, to our knowledge, ADELFE is the only AOSE methodology providing support for self-organisation [BCGP04], more precisely for the Adaptive MAS theory (AMAS). In the AMAS theory agents pursue their local goal while trying to keep cooperative relations with other agents embedded in the system [BCGP04]: ADELFE extends the Rational Unified Process (RUP) and uses UML notation and a software tool to provide graphic design capabilities. ADELFE describe an engineering process driving the designers from the analysis to the fast-prototyping stage. Although ADELFE eventually identify the need for verification of the adopted model, there is no evidence on how to perform it pragmatically: recently, the authors evaluated the integration of simulation techniques in their methodology [BGP07] to better support the design stage and the analysis of agents' behaviours. Prior to that investigation, ADELFE focussed more on aspects related to design-time, such as entities and responsibilities, rather than the actual system dynamics.

7.3 Formal Tools for Self-Organising Systems

To our knowledge, up to now there have been few attempts in using formal languages and tools for engineering self-organising systems with emergent properties. This is probably due the common misconception that emergent systems are not formalisable: while it is true that is quite difficult to capture emergent systems in formal compact models, we have provided some evidence of formalisability and a few works exist on the literature. Indeed, most of the literature involving formal methods and self-organisation is about providing formal models, but we found no use of model checking or other advanced analysis techniques. In particular, to our knowledge, none of the languages and tools presented in Chapter 5 have previously been used to formalise or verify self-organising systems.

Beyond establishing a methodological approach, the equation-free macroscopic analysis [DW07, DWSHR05] described in the previous section is heavily based on formal methods.

An example of evaluation of formal tools for SOS has been performed in [RHTR06] for the Autonomous Nano-Technology Swarm (ANTS) mission: the mission will explore the asteroid belt using swarms of cooperative autonomous spacecraft, hence, with limited or no human control involved. Hence, the authors felt the need for more guarantees about the quality of the emergent properties: in this work the authors identifies several requirements for a formal language to support their project and survey a few formal tools, including process algebras.

A less recent example include the attempt to model behaviours observed in ant colonies using the Weighted Synchronous Calculus of Communicating Systems (WSCCS) [Tof91], a process algebra extending CCS, the same that is extended by π-Calculus. In [Tof91] the author also proven some properties of the modelled systems, but without resorting to automatic techniques such as model checking. A similar approach is followed in [SBB01], where the authors first build WSCCS specifications and perform Markov Chain analysis for devising probabilities values.

Chapter 8

Conclusion and Future Works

In this chapter we conclude the thesis by summarising the contributions, the limitations and by listing future works.

8.1 Summary and Contributions

The contributions of this thesis are twofold: (i) we described a systematic approach for engineering self-organising multiagent systems, and (ii) we applied the method to several case studies and devised a solution for each of them.

For what it concerns the method, we developed a practical approach for the early stage of design, since we felt this aspect was currently overlooked in existing AOSE methodologies. The approach is a practical one since it deeply relies on experimentation techniques used in scientific analysis rather on traditional structural decomposition: indeed, quoting from [Tic98], we feel that *computer scientist should experiment more*, especially when dealing with self-organisation and emergence. Specifically, our approach is iterative, that is, cycles are performed before actually committing to a specific design: during each cycle four steps are performed

modelling proposing a model for the system to engineer based on existing design patterns extracted by natural systems: in particular, the approach relies on an architectural pattern we proposed for the A&A metamodel, but a similar one has been proposed in the Autonomic Computing context [KC03];

simulation analysing global qualitative dynamics in different scenarios before continuing with quantitative analysis;

verification verifying that the properties of interest holds and identifying working conditions;

tuning adjusting system behaviour and devising a coarse set of parameters for the actual system.

The approach can be successfully supported by formal languages and tools: indeed, formal languages allow the unambiguous specification of systems precisely selecting features to be modelled. Once modelled, it is then possible to further analyse the system via simulation or formal verification without the need of recoding the system model. To this purpose, tools like PRISM [PRI07, KNP04] have proven to be very valuable.

For what it concerns the applicability of the approach, we presented three case studies reflecting the successive developments of our approach. In particular, collective sorting is a distributed algorithm for clustering together similar items while separating different ones: we provided a novel solution to this problem in the context of tuples spaces [GVCO08]. Plain diffusion is a distributed strategy for distributed homogeneously items across a network: we provided an alternative solution to this problem that has been completely characterised in [GVO08] and reflects the current advancements of the method.

8.2 Limitations of the Approach

Although the approach described helped us in gaining new insights in self-organisation and emergence, it has some limitations too:

- Since it is difficult to devise self-organising emergent strategies from scratch, we assume the existence of a pattern encoding the desired strategy, which in many situations may not be the case: hence, modifications to the patterns may be needed, but this requires a lot of expertise [GVO08].
- We always considered one self-organising strategy at a time: things get complicated when more self-organising strategies coexist: indeed, the combination of self-organising mechanisms acting on the same environment may produce unexpected results [GVO07a].
- The approach exploits formal verification techniques, which become unfeasible for large problem instances, although abstraction techniques broaden its applicability.
- Because of the very nature of emergent properties, it may be difficult to characterise an emergent property for verification in terms of the micro dynamics: expertise is required to identify those micro-conditions triggering the emergence of the desired property [GVO08].

8.3 Future Works

We identify several potential future developments for the methodology: first, it should be better integrated with existing AOSE methodologies, and eventually specialise those activities that differs when dealing with SOS. More investigation is needed in order to evaluate self-organising strategies not observed in Nature, since relying on existing models is the main limitation of the approach.

Concerning the software, we feel that a better support should be provided at the tool level: although being valuable, the PRISM tool has some limitations. In particular, it does not provide any facility for the tuning phase and the

modelling language it is quite low-level: e.g. expressiveness comparable to a process algebra such as π-Calculus [MPW92a] would be desirable.

For what it concerns design patterns, the patterns described are far from being considered a catalogue: further effort should be devoted to identify more patterns and devise more complex patterns on top of the simpler ones. Furthermore, by their very nature, design patterns tend to emphasise design-time issues and software structure. Conversely, self-organising systems and emergence it is all about system dynamics: hence, it may be worthy considering to devise an alternative form for encoding patterns and strategies.

Appendix A

Maude Specifications for the Collective Sorting Strategy

```
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 FAST ACTIONS

op @ : [Nat] [Float] -> Event [ctor] .     *** AN EVENT
op next : List{Float} -> Event .           *** RANDOM EVENT GENERATION
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$State Float -> Step [ctor format (ni d d d d d d d)] .
    op evt(_,_,_) : Nat X$Observation Float -> Evt [ctor format (ni d 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(O,S,F) .

    op last : Trace -> Evt .

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

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

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

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

    op series : Nat Step -> Evts .

    eq series ( 0 , P ) = nil .
    eq series ( s N , P ) = last( P < empty > ) series( N , P ) .

endm

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 TYPE QQ

*** 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 .
    ***ceq move (Q1,Q2) = true if Q2 == 'noKind /\ Q1 =/= 'noKind .

    ***ceq move (Q1,?) = true if Q1 =/= ? .
    eq move (Q1,Q2) = false [owise] .

    ***op noise : TupleType TupleType Nat -> Nat .
    ***eq noise(Q1,Q2,N) = N .


    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] .
***eq temp( DS ) = false .
    *** OBS

    subsort State < Observation .
    ***subsort Nat < 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 o ( S:State ) = S:State .


     ***eq o ( S:State ) = 0 [owise] .
     ***eq o ( S:State ) = 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-2 : -> State .
eq SS-2 = ( init | {2} |
< 0 @ ('a[250])|('b[250])|(?[0]) | tot(2)  > |
< 1 @ ('a[250])|('b[250])|(?[0]) | tot(2)  > |  # 0 # | # 2.0 # ) .

op SS-3 : -> State .
eq SS-3 = ( init | {3} |
< 0 @ ('a[25])|('b[25])|('c[25])|(?[0]) | tot(3) > |
< 1 @ ('a[25])|('b[25])|('c[25])|(?[0]) | tot(3)  > |
< 2 @ ('a[25])|('b[25])|('c[25])|(?[0]) | tot(3)  > |
  # 0 # | # 3.0 # ) .

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 # ) .


op SS-5 : -> State .
eq SS-5 = ( init | {5} |
< 0 @ ('a[25])|('b[25])|('c[25])|('d[25])|('e[25])|(?[0]) | tot(5) > |
< 1 @ ('a[25])|('b[25])|('c[25])|('d[25])|('e[25])|(?[0]) | tot(5)  > |
< 2 @ ('a[25])|('b[25])|('c[25])|('d[25])|('e[25])|(?[0]) | tot(5)  > |
< 3 @ ('a[25])|('b[25])|('c[25])|('d[25])|('e[25])|(?[0]) | tot(5)  > |
< 4 @ ('a[25])|('b[25])|('c[25])|('d[25])|('e[25])|(?[0]) | tot(5)  > |  # 0 # | # 5.0 # ) .
```

```
op SS-5-bis : -> State .
eq SS-5-bis = ( init | {5} | # 1333 # | # 5.0 # |
< 0 @ tot(1) | ('a[0]) | ('b[0]) | ('c[125]) | ('d[0]) | ('e[0]) | (?[0]) > |
< 1 @ tot(2) | ('a[0]) | ('b[74]) | ('c[0]) | ('d[0]) | ('e[43]) | (?[0]) > |
< 2 @ tot(1) | ('a[0]) | ('b[51]) | ('c[0]) | ('d[0]) | ('e[0]) | (?[0]) > |
< 3 @ tot(1) | ('a[0]) | ('b[0]) | ('c[0]) | ('d[125]) | ('e[0]) | (?[0]) > |
< 4 @ tot(2) | ('a[125]) | ('b[0]) | ('c[0]) | ('d[0]) | ('e[82]) | (?[0]) > ) .

op SS-6 : -> State .
eq SS-6 = ( init | {4} |
< 0 @ ('a[10])|('b[10])|('c[10])|('d[10])|('e[10])|('f[10])|(?[0])| tot(6) > |
< 1 @ ('a[10])|('b[10])|('c[10])|('d[10])|('e[10])|('f[10])|(?[0])| tot(6) > |
< 2 @ ('a[10])|('b[10])|('c[10])|('d[10])|('e[10])|('f[10])|(?[0])| tot(6) > |
< 3 @ ('a[10])|('b[10])|('c[10])|('d[10])|('e[10])|('f[10])|(?[0])| tot(6) > |  # 0 # | # 40.0 # ) .

op SS-7 : -> State .
eq SS-7 = ( init | {4} |
< 0 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|(?[0]) > |
< 1 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|(?[0]) > |
< 2 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|(?[0]) > |
< 3 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|(?[0]) > |  # 0 # ) .

op SS-10 : -> State .
eq SS-10 = ( init | {4} |
< 0 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|('h[13])|('i[13])|('l[13])|(?[0]) > |
< 1 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|('h[13])|('i[13])|('l[13])|(?[0]) > |
< 2 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|('h[13])|('i[13])|('l[13])|(?[0]) > |
< 3 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|('h[13])|('i[13])|('l[13])|(?[0]) > |  # 0 # ) .

op SS-13 : -> State .
eq SS-13 = ( init | {4} |
< 0 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|('h[13])|
('i[13])|('l[13])|('m[13])|('n[13])|('o[13])|(?[0]) > |
< 1 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|('h[13])|
('i[13])|('l[13])|('m[13])|('n[13])|('o[13])|(?[0]) > |
< 2 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|('h[13])|
('i[13])|('l[13])|('m[13])|('n[13])|('o[13])|(?[0]) > |
< 3 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|('h[13])|
('i[13])|('l[13])|('m[13])|('n[13])|('o[13])|(?[0]) > |
# 0 # ) .

op SS-16 : -> State .
eq SS-16 = ( init | {4} |
< 0 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|('h[13])|
('i[13])|('l[13])|('m[13])|('n[13])|('o[13])|('p[13])|('q[13])|('r[13])|(?[0]) > |
< 1 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|('h[13])|('i[13])|
('l[13])|('m[13])|('n[13])|('o[13])|('p[13])|('q[13])|('r[13])|(?[0]) > |
< 2 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|('h[13])|('i[13])|
('l[13])|('m[13])|('n[13])|('o[13])|('p[13])|('q[13])|('r[13])|(?[0]) > |
< 3 @ ('a[13])|('b[13])|('c[13])|('d[13])|('e[13])|('f[13])|('g[13])|('h[13])|('i[13])|
('l[13])|('m[13])|('n[13])|('o[13])|('p[13])|('q[13])|('r[13])|(?[0]) > |  # 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 .
```

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. *Commununications of the ACM*, 43(5):74–82, May 2000.

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York, 1977.

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

[BCD+06] Ozalp Babaoglu, Geoffrey Canright, Andreas Deutsch, Gianni A. Di Caro, Frederick Ducatelle, Luca M. Gambardella, Niloy Ganguly, Márk Jelasity, Roberto Montemanni, Alberto Montresor, and Tore Urnes. Design patterns from biology for distributed computing. *Transactions on Autonomous and Adaptive Systems (TAAS)*, 1(1):26–66, September 2006.

[BCGP04] Carole Bernon, Valérie Camps, Marie-Pierre Gleizes, and Gauthier Picard. Designing agents' behaviors and interactions within the framework of ADELFE methodology. In *Engineering Societies in the Agents World*, volume 3071 of *LNCS (LNAI)*, pages 311–327. Springer, 2004. 4th International Workshops, ESAW 2003, London, UK, October 29-31, 2003, Revised Selected and Invited Papers.

[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, New York 10016, United States of America, 1999.

[Ber92] Alan A. Berryman. The origins and evolution of predator-prey theory. *Ecology*, 73(5):1530–1535, October 1992.

[BGP07] 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 Oğuz Dikenelli, editors, *Engineering Societies in the*

Agents World VII, volume 4457 of *LNCS (LNAI)*, pages 284–299. Springer, September 2007. 7th International Workshop, ESAW 2006 Dublin, Ireland, September 6-8, 2006 Revised Selected and Invited Papers.

[BH01] Ed Brinksma and Holger Hermanns. Process algebra and markov chains. In E. Brinksma, H. Hermanns, and J.-P. Katoen, editors, *Lectures on Formal Methods and Performance Analysis : First EEF/Euro Summer School on Trends in Computer Science Berg en Dal, The Netherlands, July 3-7, 2000, Revised Lectures*, volume 2090 of *LNCS*, pages 183–231. Springer, 2001.

[BMM02] Ozalp Babaoglu, Hein Meling, and Alberto Montresor. Anthill: a framework for the development of agent-based peer-to-peer systems. In *22nd International Conference on Distributed Computing Systems (ICDCS02)*, pages 15–22, Vienna, Austria, July 2002. IEEE Computer Society.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual*. University of Illinois ad Urbana-Champaign, 2.3 edition, July 2007. 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, New Jersey 08540, United States of America, 2001.

[CDU06] Geoffrey Canright, Andreas Deutsch, and Tore Urnes. Chemotaxis-inspired load balancing. *Complexus*, 3(1-3):8–23, August 2006.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, September 1994.

[CGV06a] Matteo Casadei, Luca Gardelli, and Mirko Viroli. A case of self-organising environment for MAS: the collective sort problem. In Barbara Dunin-Kęplicz, Andrea Omicini, and Julian Padget, editors, *4th European Workshop on Multi-Agent Systems (EUMAS 2006)*, volume 223, Lisbon, Portugal, December 2006. CEUR.

[CGV06b] Matteo Casadei, Luca Gardelli, and Mirko Viroli. Collective sorting tuple spaces. In Andrea Omicini Flavio De Paoli, Antonella Di Stefano and Corrado Santoro, editors, *Dagli oggetti agli agenti: Sistemi Grid, P2P e Self-*, AI*IA/TABOO Joint Workshop (WOA 2006)*, pages 173–180, Catania - Italy, September 2006. Technical University of Aachen.

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

[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 (ENTCS)*, 175(2):59–80, June 21 2007. Proceedings of the Fifth International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2006).

[CS04] Vincent A. Cicirello and Stephen F. Smith. Wasp-like agents for distributed factory coordination. *Autonomous Agents and Multi-Agent Sytems*, 8(3):237–266, May 2004.

[DBS06] Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant Colony Optimization: Artificial ants as a computational intelligence technique. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

[Des37] René Descartes. *Discourse on the Method of Rightly Conducting the Reason, and Searching for Truth in the Sciences.* Leyde, 1637. Available online at Project Gutenberg 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*, Classics, pages 356–363. MIT Press, Cambridge, Massachusetts 02142, USA, February 1991.

[DS04] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization.* MIT Press, Cambridge, MA, July 2004.

[DW07] Tom De Wolf. *Analysing and engineering self-organising emergent applications.* PhD in computer science, Faculteit Ingenieurswetenschappen — Katholieke Universiteit Leuven, Celestijnenlaan 200 A - 3001 Leuven - Belgium, May 2007.

[DWH05] Tom De Wolf and Tom Holvoet. Emergence versus self-organisation: Different concepts but promising when combined. In S. Brueckner, G. Di Marzo Serugendo, A. Karageorgos, and R. Nagpal, editors, *Engineering Self Organising Systems: Methodologies and Applications*, volume 3464 of *LNCS (LNAI)*, pages 1–15. Springer, May 2005.

[DWH07] Tom De Wolf and Tom Holvoet. Design patterns for decentralised coordination in self-organising emergent systems. In Sven Brueckner, Salima Hassas, Mrk Jelasity, and Daniel Yamins, editors, *Engineering Self-Organising Systems*, volume 4335 of *LNCS (LNAI)*,

pages 28–49. Springer, 2007. Fourth International Workshop, ESOA 2006, Future University-Hakodate, Japan, 2006, Revised Selected Papers.

[DWHS06] Tom De Wolf, Tom Holvoet, and Giovanni Samaey. Development of self-organising emergent applications with simulation-based numerical analysis. In Sven A. Brueckner, Giovanna Di Marzo Serugendo, David Hales, and Franco Zambonelli, editors, *Engineering Self-Organising Systems*, volume 3910 of *LNCS (LNAI)*, pages 138–152. Springer, May 2006. Third International Workshop, ESOA 2005, Utrecht, The Netherlands, July 2005, Revised Selected Papers.

[DWSHR05] Tom De Wolf, Giovanni Samaey, Tom Holvoet, and Dirk Roose. Decentralised autonomic computing: Analysing self-organising emergent behaviour using advanced numerical methods. In *Second International Conference on Autonomic Computing (ICAC'05)*, pages 52–63, Seattle,Washington, USA, June 2005. IEEE.

[EMCGP99] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[FHS97] Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, October 1997.

[Gar05] Luca Gardelli. Self-organization and coordination for multi-agent systems. Technical report, European Science Foundation (ESF) MiNEMA Scientific Programme, November 2005. MiNEMA Exchange Grants Publications (no. 805)- Visit to Katholieke Universiteit Leuven.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Professional Computing. Addison-Wesley, One Lake Street, Upper Saddle River, NJ, 07458, USA, December 1995.

[Gil77] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[Gol99] Jeffrey Goldstein. Emergence as a construct: History and issues. *Emergence*, 1(1):49–72, 1999.

[Gra59] Pierre-Paul Grassé. La reconstruction du nid et les coordinations interindividuelles chez bellicositermes natalensis et cubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs. *Insectes Sociaux*, 6(1):41–80, March 1959.

[GVC06a] Luca Gardelli, Mirko Viroli, and Matteo Casadei. Engineering the environment of self-organising multi-agent systems exploiting formal analysis tools. In *Congresso AICA 2006*, Cesena - Italy,

September 2006. AICA - Associazione Italiana per l'Informatica e il Calcolo Automatico.

[GVC06b] 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 on Environments for Multi-Agent Systems (E4MAS 2006)*, pages 96–101, AAMAS 2006, Hakodate, Japan, May 2006.

[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 Multi-Agent Systems III*, volume 4389 of *LNCS (LNAI)*, pages 254–271. Springer, February 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 (IJAOSE)*, 2(2):171–195, 2008. Special Issue on Multi-Agent Systems and Simulation.

[GVO05a] Luca Gardelli, Mirko Viroli, and Andrea Omicini. Engineering self-organizing MAS with coordination artifacts and ACCs. In Jean-Marie Jacquet and Gian Pietro Picco, editors, *7th International Conference on Coordination Languages and Models (COORDINATION 2005)*, Namur, Belgium, April 2005. Poster.

[GVO05b] Luca Gardelli, Mirko Viroli, and Andrea Omicini. On the role of simulation in the engineering of self-organising systems: Detecting abnormal behaviour in MAS. In Flavio Corradini, Flavio De Paoli, Emanuela Merelli, and Andrea Omicini, editors, *AI*IA/TABOO Joint Workshop "Dagli oggetti agli agenti: simulazione e analisi formale di sistemi complessi" (WOA 2005)*, pages 85–90, Camerino, MC, Italy, November 2005. Pitagora Editrice Bologna.

[GVO05c] Luca Gardelli, Mirko Viroli, and Andrea Omicini. On the role of simulations in engineering self-organizing MAS: the case of an intrusion detection system in TuCSoN. In Sven Brueckner, Giovanna Di Marzo Serugendo, David Hales, and Franco Zambonelli, editors, *3rd International Workshop on Engineering Self-Organising Applications (ESOA 2005)*, pages 161–175, AAMAS 2005, Utrecht, The Netherlands, July 2005.

[GVO06a] Luca Gardelli, Mirko Viroli, and Andrea Omicini. Exploring the dynamics of self-organising systems with stochastic π-calculus: Detecting abnormal behaviour in MAS. In Robert Trappl, editor, *Cybernetics and Systems 2006*, volume 2, pages 539–544, Vienna, Austria, April 2006. Austrian Society for Cybernetic Studies. 18th European Meeting on Cybernetics and Systems Research (EMCSR

2006), 5th International Symposium From Agent Theory to Theory Implementation (AT2AI-5).

[GVO06b] 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 *LNCS (LNAI)*, pages 153–166. Springer Berlin / Heidelberg, April 2006. Third International Workshop, ESOA 2005, Utrecht, The Netherlands, July 25, 2005, Revised Selected Papers.

[GVO07a] Luca Gardelli, Mirko Viroli, and Andrea Omicini. Design patterns for self-organising systems. In Hans-Dieter Burkhard, Gabriela Lindemann, Rineke Verbrugge, and László Z. Varga, editors, *Multi-Agent Systems and Applications V*, volume 4696 of *LNCS (LNAI)*, pages 123–132. Springer, Heidelberg, 2007. 5th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2007, Leipzig, Germany, September 25–27, 2007, Proceedings. In Press.

[GVO07b] Luca Gardelli, Mirko Viroli, and Andrea Omicini. Design patterns for self-organizing multiagent systems. In Tom De Wolf, Fabrice Saffre, and Richard Anthony, editors, *2nd International Workshop on Engineering Emergence in Decentralised Autonomic System (EEDAS) 2007*, pages 62–71, ICAC 2007, Jacksonville, Florida, USA, June 2007. CMS Press, University of Greenwich, London, UK.

[GVO08] Luca Gardelli, Mirko Viroli, and Andrea Omicini. *Agents, Simulation and Applications.*, chapter Simulation for the Development of Self-Organising Multi-Agent Systems. Taylor & Francis, 2008. To Appear.

[Hor01] Paul Horn. Autonomic computing manifesto. Available online at http://www.research.ibm.com/autonomic/manifesto/, October 2001.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[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, September 2004. Special section on tools and algorithms for the construction and analysis of systems.

[KNP07] Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. In Marco Bernardo and Jane 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, June 2007. 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007.

[Lin03] Jürgen Lind. Patterns in agent-oriented software engineering. In Fausto Giunchiglia, James Odell, and Gerhard Weiß, editors, *Agent-Oriented Software Engineering III*, volume 2585 of *LNCS*, pages 47–58. Springer, February 2003. 3rd International Workshop on Agent Oriented Software Engineering (AOSE 2002), Bologna, Italy, July 15, 2002, Revised Papers and Invited Contributions.

[Lui06] Pier Luigi Luisi. *The Emergence of Life: From Chemical Origins to Synthetic Biology*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 2006.

[MÖ4] Jean-Pierre Müller. Emergence of collective behaviour and problem solving. In Andrea Omicini, Paolo Petta, and Jeremy Pitt, editors, *Engineering Societies in the Agents World IV*, volume 3071 of *LNCS (LNAI)*, pages 1–21. Springer, June 2004. 4th International Workshops, ESAW 2003, London, UK, October 29-31, 2003, Revised Selected and Invited Papers.

[Mau07] Maude. The MAUDE system, November 2007. Developed at University of Illinois at Urbana-Champaign. Version 2.3 available online at http://maude.cs.uiuc.edu.

[MCA00] John McHugh, Alan Christie, and Julia Allen. Defending yourself: The role of intrusion detection systems. *IEEE Software*, 17(5):42–51, September/October 2000.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, June 1999.

[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, August-September 2006 2006.

[MODR06] Ambra Molesini, Andrea Omicini, Enrico Denti, and Alessandro Ricci. SODA: A roadmap to artefacts. In Oğuz Dikenelli, Marie-Pierre Gleizes, and Alessandro Ricci, editors, *Engineering Societies in the Agents World VI*, volume 3963 of *LNCS (LNAI)*, pages 49–62. Springer, June 2006. 6th International Workshop (ESAW 2005), Kuşadası, Aydın, Turkey, 26–28 October 2005. Revised, Selected & Invited Papers.

[MOV07] Ambra Molesini, Andrea Omicini, and Mirko Viroli. Environment in agent-oriented software engineering methodologies. *International Journal on Multiagent and Grid Systems*, 2007. In Press. Special Issue on Engineering Environments for Multiagent Systems.

[MPW92a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes I. *Information and Computation*, 100(1):1–40, September 1992.

[MPW92b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes II. *Information and Computation*, 100(1):41–77, September 1992.

[Mur02] James D. Murray. *Mathematical Biology: An Introduction*, volume 17 of *Interdisciplinary Applied Mathematics*. Springer, 3rd edition, 2002.

[MZ05] Marco Mamei and Franco Zambonelli. Programming stigmergic coordination with the TOTA middleware. In *4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*, pages 415–422, New York, NY, USA, July 2005. ACM Press.

[MZ07] Marco Mamei and Franco Zambonelli. Pervasive pheromone-based interaction with rfid tags. *Transactions on Autonomous and Adaptive Systems (TAAS)*, 2(2):Number 4, June 2007.

[ORV06] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. *Agens Faber*: Toward a theory of artefacts for MAS. *Electronic Notes in Theoretical Computer Sciences*, 150(3):21–36, May 2006. 1st International Workshop "Coordination and Organization" (CoOrg 2005), COORDINATION 2005, Namur, Belgium, 22 April 2005. Proceedings.

[Par06] H. Van Dyke Parunak. A survey of environments and mechanisms for human-human stigmergy. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for Multi-Agent Systems II*, volume 3830 of *LNCS (LNAI)*, pages 163–186. Springer, February 2006. 2nd International Workshop, E4MAS 2005, Utrecht, The Netherlands, July 25, 2005, Selected Revised and Invited Papers.

[PBS05] H. Van Dyke Parunak, Sven A. Brueckner, and John Sauter. Digital pheromones for coordination of unmanned vehicles. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for Multi-Agent Systems*, volume 3374 of *LNCS (LNAI)*, pages 246–263. Springer, February 2005. 1st International Workshop on Environments for Multiagent Systems, E4MAS 2004. New York NY, USA. July 19, 2004. Revised Selected Papers.

[PC04] Andrew Phillips and Luca Cardelli. A correct abstract machine for the stochastic pi-calculus. In *Concurrent Models in Molecular Biology (Bioconcur'04)*, London, August 2004.

[Phi07] Andrew Phillips. The stochastic pi-machine (SPiM). Version 0.044 available online at http://research.microsoft.com/~aphillip/spim/, November 2007.

[Pri95] Corrado Priami. Stochastic π-calculus. *The Computer Journal*, 38(7):578–589, 1995.

[PRI07] PRISM. PRISM: Probabilistic symbolic model checker, November 2007. Developed at University of Birmingham, UK. Version 3.1.1 available online at http://www.prismmodelchecker.org.

[RHTR06] Christopher A. Rouff, Michael G. Hinchey, Walter F. Truszkowski, and James L. Rash. Experiences applying formal approaches in the development of swarm-based space exploration systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):587–603, November 2006. Special Section On Leveraging Applications of Formal Methods.

[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, USA, 2004.

[RN02] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Pearson Education, Inc., Upper Saddle River, New Jersey 07458, USA, 2nd edition, December 2002.

[ROV+07] Alessandro Ricci, Andrea Omicini, Mirko Viroli, Luca Gardelli, and Enrico Oliva. Cognitive stigmergy: Towards a framework based on agents and artifacts. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for Multi-Agent Systems III*, volume 4389 of *LNCS (LNAI)*, pages 124–140. Springer, February 2007. 3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 may 2006. Selected Revised and Invited Papers.

[RVO06] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Programming MAS with artifacts. In Rafael P. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 3862 of *LNCS (LNAI)*, pages 206–221. Springer, March 2006. 3rd International Workshop (PROMAS 2005), AAMAS 2005, Utrecht, The Netherlands, July 26, 2005. Revised and Invited Papers.

[SA94] S. Steward and S. Appleby. Mobile software agents for control of distributed systems based on principles of social insect behaviour. In *International Conference on Communications Systems (ICCS'94)*, volume 2, pages 549–553, Singapore, November 1994. IEEE.

[SB06] Ricard V. Solé and Jordi Bascompte. *Self-Organization in Complex Ecosystems*. Number 42 in Monographs in population Biology. Princeton University Press, 41 William Street, Princeton, New Jersey 08540, United States of America, 2006.

[SBB01] D. J. T. Sumpter, G. B. Blanchard, and D. S. Broomhead. Ants and agents: A process algebra approach to modelling ant colony behaviour. *Bulletin of Mathematical Biology*, 63(5):951–980, September 2001.

[SFH+04] Giovanna Di Marzo Serugendo, Noria Foukia, Salima Hassas, Anthony Karageorgos, Soraya Kouadri Mostéfaoui, Omer F. Rana, Mihaela Ulieru, Paul Valckenaers, and Chris Van Aart. Self-organisation: Paradigms and applications. In Giovanna Di Marzo Serugendo, Anthony Karageorgos, Omer F. Rana, and Franco Zambonelli, editors, *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*, volume 2977 of *LNCS (LNAI)*, pages 1–19. Springer, May 2004. International Workshop on Engineering Self-Organizing Applications (ESOA 2003). Melbourne, Australia, July 2003.

[Tic98] Walter F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, May 1998.

[Tof91] Chris Tofts. Describing social insect behaviour using process algebra. *Transactions on Social Computing Simulation*, pages 227–283, 1991.

[Uhr02] Adelinde M. Uhrmacher. Simulation for agent-oriented software engineering. In W.H. Lunceford and E. Page, editors, *First International Conference on Grand Challenges*, San Antonio, TX, USA, January 2002. SCS, San Diego.

[VCG07] Mirko Viroli, Matteo Casadei, and Luca Gardelli. A self-organising solution to the collective sort problem for distributed tuple spaces. In *22th ACM Symposium on Applied Computing (SAC'07)*, pages 354–359, New York, NY, USA, March 2007. ACM Press. Special Track on Coordination Models, Languages and Applications.

[VHR+07] Mirko Viroli, Tom Holvoet, Alessandro Ricci, Kurt Schelfthout, and Franco Zambonelli. Infrastructures for the environment of multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):49–60, February 2007. Special Issue on Environments for Multi-agent Systems.

[vM93] Anneliese von Mayrhauser. The role of simulation in software engineering experimentation. In *Experimental Software Engineering Issues: Critical Assessment and Future Directions*, volume 706 of *LNCS*, pages 177–179. Springer, 1993.

[WOO07] Danny Weyns, Andrea Omicini, and James Odell. Environment as a first-class abstraction in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, February 2007. Special Issue on Environments for Multi-agent Systems.

[WSHL05] Danny Weyns, Kurt Schelfthout, Tom Holvoet, and Tom Lefever. Decentralized control of E'GV transportation systems. In *4th International Joint Conference on Autonomous Agents and Multi-*

agent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands, pages 67–74, New York, NY, USA, July 2005. ACM.

[ZJW03] Franco Zambonelli, Nicholas R. Jennings, and Michael J. Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(3):317–370, July 2003.

[ZO04] Franco Zambonelli and Andrea Omicini. Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3):253–283, November 2004. Special Issue: Challenges for Agent-Based Computing.

List of Publications

Book Chapters

Luca Gardelli, Mirko Viroli, and Andrea Omicini. In Danny Weyns and Adeline M. Uhrmacher, editors, Agents, Simulation and Applications. *Simulation for the Development of Self-Organising Multi-Agent Systems*. Taylor & Francis, 2008. To Appear.

Articles in Journals

Luca Gardelli, Mirko Viroli, Matteo Casadei, and Andrea Omicini. *Designing self-organising environments with agents and artifacts: A simulation-driven approach*. International Journal of Agent-Oriented Software Engineering (IJAOSE), 2(2):171-195, 2008.

Matteo Casadei, Luca Gardelli, and Mirko Viroli. *Simulating emergent properties of coordination in Maude: the collective sort case*. Electronic Notes in Theoretical Computer Science (ENTCS), 175(2):59-80, June 21 2007. Proceedings of the Fifth International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2006).

Articles in Springer LNCS

Luca Gardelli, Mirko Viroli, and Andrea Omicini. *Design patterns for self-organising systems*. In Hans-Dieter Burkhard, Gabriela Lindemann, Rineke Verbrugge, and László Z. Varga, editors, Multi-Agent Systems and Applications V, volume 4696 of LNCS (LNAI), pages 123-132. Springer, Heidelberg, 2007. 5th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2007, Leipzig, Germany, September 25-27,2007, Proceedings.

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 Multi-Agent Systems III, volume 4389 of LNCS (LNAI), pages 254-271. Springer,

February 2007. 3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 May 2006. Selected Revised and Invited Papers.

Alessandro Ricci, Andrea Omicini, Mirko Viroli, Luca Gardelli, and Enrico Oliva. *Cognitive stigmergy: Towards a framework based on agents and artifacts.* In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, Environments for Multi-Agent Systems III, volume 4389 of LNCS (LNAI), pages 124-140. Springer, February 2007. 3rd International Workshop (E4MAS 2006), Hakodate, Japan, 8 may 2006. Selected Revised and Invited Papers.

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 LNCS (LNAI), pages 153-166. Springer Berlin / Heidelberg, April 2006. Third International Workshop, ESOA 2005, Utrecht, The Netherlands, July 25, 2005, Revised Selected Papers.

Contributions at Conferences or Workshops

Luca Gardelli, Mirko Viroli, and Andrea Omicini. *Design patterns for self-organizing multiagent systems.* In Tom De Wolf, Fabrice Saffre, and Richard Anthony, editors, 2nd International Workshop on Engineering Emergence in Decentralised Autonomic System (EEDAS) 2007, pages 62-71, ICAC 2007, Jacksonville, Florida, USA, June 2007. CMS Press, University of Greenwich, London, UK.

Mirko Viroli, Matteo Casadei, and Luca Gardelli. *A self-organising solution to the collective sort problem for distributed tuple spaces.* In 22th ACM Symposium on Applied Computing (SAC'07), pages 354-359, New York, NY, USA, March 2007. ACM Press. Special Track on Coordination Models, Languages and Applications.

Matteo Casadei, Luca Gardelli, and Mirko Viroli. *A case of self-organising environment for MAS: the collective sort problem.* In Barbara Dunin-Kęplicz, Andrea Omicini, and Julian Padget, editors, 4th European Workshop on Multi-Agent Systems (EUMAS 2006), volume 223, Lisbon, Portugal, December 2006. CEUR.

Matteo Casadei, Luca Gardelli, and Mirko Viroli. *Collective sorting tuple spaces.* In Andrea Omicini Flavio De Paoli, Antonella Di Stefano and Corrado Santoro, editors, Dagli oggetti agli agenti: Sistemi Grid, P2P e Self-*, AI*IA/TABOO Joint Workshop (WOA 2006), pages 173-180, Catania - Italy, September 2006. Technical University of Aachen.

Matteo Casadei, Luca Gardelli, and Mirko Viroli. *Simulating emergent properties of coordination in Maude: the collective sorting case.* In Carlos Canal and Mirko Viroli, editors, 5th International Workshop on the Foundations of

Coordination Languages and Software Architectures (FOCLASA), pages 57-75, CONCUR 2006, Bonn, Germany, August 2006. University of Malaga, Spain.

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 on Environments for Multi-Agent Systems (E4MAS 2006), pages 96-101, AAMAS 2006, Hakodate, Japan, May 2006.

Alessandro Ricci, Andrea Omicini, Mirko Viroli, Luca Gardelli, and Enrico Oliva. *Cognitive Stigmergy: A Framework Based on Agents and Artifacts.* In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, 3rd International Workshop on Environments for Multi-Agent Systems (E4MAS 2006), pages 44-60, AAMAS 2006, Hakodate, Japan, May 2006.

Luca Gardelli, Mirko Viroli, and Matteo Casadei. *Engineering the environment of self-organising multi-agent systems exploiting formal analysis tools.* In Congresso AICA 2006, Cesena - Italy, September 2006. AICA - Associazione Italiana per l'Informatica e il Calcolo Automatico.

Luca Gardelli, Mirko Viroli, and Andrea Omicini. *Exploring the dynamics of self-organising systems with stochastic π-calculus: Detecting abnormal behaviour in MAS.* In Robert Trappl, editor, Cybernetics and Systems 2006, volume 2, pages 539-544, Vienna, Austria, April 2006. Austrian Society for Cybernetic Studies. 18th European Meeting on Cybernetics and Systems Research (EMCSR 2006), 5th International Symposium From Agent Theory to Theory Implementation (AT2AI-5).

Alessandro Ricci, Andrea Omicini, Mirko Viroli, Luca Gardelli, and Enrico Oliva. *Cognitive Stigmergy: A Framework Based on Agents and Artifacts.* In Marie-Pierre Gleizes, Gal A. Kaminka, Ann Nowé, Sascha Ossowski, Karl Tuyls, and Katja Verbeeck, editors, 3rd European Workshop on Multi-Agent Systems (EUMAS 2005), pages 332-343, Brussels, Belgium, December 2005. Koninklijke Vlaamse Academie van Belie voor Wetenschappen en Kunsten.

Luca Gardelli, Mirko Viroli, and Andrea Omicini. *On the role of simulation in the engineering of self-organising systems: Detecting abnormal behaviour in MAS.* In Flavio Corradini, Flavio De Paoli, Emanuela Merelli, and Andrea Omicini, editors, AI*IA/TABOO Joint Workshop Dagli oggetti agli agenti: simulazione e analisi formale di sistemi complessi (WOA 2005), pages 85-90, Camerino, MC, Italy, November 2005. Pitagora Editrice Bologna.

Luca Gardelli, Mirko Viroli, and Andrea Omicini. *On the role of simulations in engineering self-organizing MAS: the case of an intrusion detection system in* TuCSoN. In Sven Brueckner, Giovanna Di Marzo Serugendo, David Hales, and Franco Zambonelli, editors, 3rd International Workshop on Engineering Self-Organising Applications (ESOA 2005), pages 161-175, AAMAS 2005, Utrecht, The Netherlands, July 2005.

Luca Gardelli, Mirko Viroli, and Andrea Omicini. *Engineering self-organizing*

MAS with coordination artifacts and ACCs. In Jean-Marie Jacquet and Gian Pietro Picco, editors, 7th International Conference on Coordination Languages and Models (COORDINATION 2005), Namur, Belgium, April 2005. Poster.

Technical Reports

Luca Gardelli. *Self-organization and coordination for multi-agent systems.* Technical report, European Science Foundation (ESF) MiNEMA Scientific Programme, November 2005. MiNEMA Exchange Grants Publications (no. 805)-Visit to Katholieke Universiteit Leuven.

Biography

Luca Gardelli was born on June 22th, 1980 in Cesena (FC), Italy. In 2002, he received his *Laurea Triennale* degree cum laudae on Computer Science Engineering from the Alma Mater Studiorum—Università di Bologna, Italy: the thesis title was *Designing a Web Site for Mobile Phone Services* and was supervised by Prof. Andrea Omicini. In 2004, he received his *Laurea Specialistica* degree cum laudae on Computer Science Engineering from the Alma Mater Studiorum—Università di Bologna: the thesis title was *Target Tracking in Sensor Networks: a Multiagent Approach* and was supervised by Prof. Andrea Omicini. In 2005, he started working as a PhD student at the aliCE research group of the DEIS, Department of Electronics, Computer Science and Systems at the Alma Mater Studiorum—Università di Bologna. During his PhD course, he wrote several articles: specifically, two articles for international journals, a book chapter, a technical report, four articles published in Springer LNCS, and thirteen contributions among conferences and workshops. Furthermore, he participated to the MiNEMA project funded by European Science Foundation (ESF): between November and December 2005, under the supervision of Prof. Tom Holvoet, he collaborated with the DistriNet (Distributed Systems and computer Networks) research group of the Department of Computer Science at the Katholieke Universiteit Leuven in Belgium.