UNIVERSITÀ DEGLI STUDI DI TRENTO
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica Specialistica

Elaborato Finale

Implicit Culture for Self-Organizing Systems: a concrete scenario and its performance analysis

Relatore

prof. Paolo Giorgini

Laureando

Massimiliano Bernabé

Anno accademico 2005/2006

Contents

Chapter 1

Introduction

The idea that the order of a system can be increased by its dynamics has a long history. One of the earliest statements of this idea was by the philosopher Descartes, in the fifth part of his Discourse on Method, where he presents it hypothetically. He further elaborated on the idea in a book called Le Monde that was never published.

The ancient anatomists believed that designing intelligence was unnecessary, because given enough time and space, organization come certainly out, although there is not a specific tendence for this to happen. What Descartes introduced was the idea that the ordinary laws of nature tend to produce organization.

Starting from 1700 the scientific community tried to understand the "universal laws of form" in order to explain the observed shapes of living organisms but nothig concrete happened. In the beginnin of the 20th century D'Arcy Wentworth Thompson and other researchers resumed those ideas concluding that there exists a universal law governing the growth and the form of biological systems.

The term "self-organizing" was introduced in 1947 by W. Ross Ashby. It was used by Heinz von Foerster, Gordon Pask, Stafford Beer and Norbert Wiener in the second edition of their "Cybernetics: Control and Communication in the Animal and the Machine". Self-organization as a word and concept was used by those scientists associated with general systems theory in the 1960s, but did not become commonplace in the scientific literature until its adoption by physicists and researchers in the field of complex systems in 1970s and 1980s.

1.1 Motivation

The software industry is facing a hard-to-solve problem: computing systems are getting more and more complex due to the integration and the increasing presence of the network in the enterprises. This makes ordinary administra-

tion more complex too and seems to be a limiting factor in the economics development.

This trouble motivates the scientific comunity to propose software systems which self-organize. Unfortunately, a proper classification of them is missing and in many cases the self-organizing property is not clearly proved.

In our point of view this is undesirable because it adds useless confusion to the field.

For those reasons we decided to present a new method for the evaluation of self-organizing systems. The method is based on the standard performance analysis and it has been tested on a real self-organizing system.

1.2 Contribution

The aim of this thesis is twofold: present a rational procedure to evaluate self-organizing systems and proove that the Implicit Culture framework can be used to build them. The thesis is divided in four parts:

In the first chapter we focus on self-organizing systems. We give some operative definitions to distinguish self-organizing software from others. Moreover we present some principles and suggestions to follow in order to build an effective SOS are shown in the final part of the chapter.

The second chapter is on the evauation of a general self-organizing system. We take a simple case study and we extend some definitions from standard performance evaluation to it.

In the third chapter we survey the Implict Culture framework and its components, then we motivate the choice of Implicit Culture for implementing SOSs.

The fourth chapter presents a real self-organizing software supported by Implicit Culture, the description of the agents acting in it, their behaviour, the protocols of interaction and the class diagrams. In the final section we prove that our system is self-organizing, we carry out the performance analisys and we show how our method for evaluation can be used for organization detection.

Introduzione

L'idea che il grado di organizzazione di un sistema puó essere incrementato dai suoi componenti ha una lunga storia. Una delle prime dichiarazioni di quest'idea fú del filosofo Descartes, nella quinta parte del suo Discorso sul Metodo, dove l'ha presentata in maniera ipotetica. Egli ha successivamente rielaborato l'idea in un libro chiamato Le Monde che non fú mai pubblicato.

Nell'antichitá gli anatomisti credevano che progettare l'intelligenza non fosse necessario, perché data una quantitá opportuna di tempo e spazio, una sorta di organizzazione emerge sicuramente, anche se non esiste nessuna particolare tendenza che questo accada. Quello che Descartes introdusse fú l'idea che le ordinarie leggi della natura hanno la tendenza a produrre organizzazione.

Dal 1700 in poi la comunitá scientifica ha provato a capire la "legge universale della forma" con l'obiettivo di spiegare la forma degli organismi, ma nessun significativo passso in avanti fú compiuto. All'inizio del ventesimo secolo D'Arcy Wentworth Thompson ed altri ricercatori hanno ripreso queste idee ed hanno scoperto che esiste una legge universale che determina la forma e lo sviluppo di tutti gli esseri viventi.

Il termine "auto organizzante" è stato introdotto nel 1947 da W. Ross Ashby. Successivamente è stato utilizzato da Heinz von Foerster, Gordon Pask, Stafford Beer e Norbert Wiener nella seconda edizione di "Cybernetics: Control and Communication in the Animal and the Machine". Auto organizzazione come parola e concetto è stato usato da questi scienziati in associazione con la teoria dei sistemi negli anni sessanta, ma non divenne popolare nella letteratura scientifica finché non fu adottato da fisici e da ricercatori nell'ambito dei sistemi complesi negli anni settanta e ottanta.

Motivazioni

L'industria del software si stá confrontando con un problema di difficile soluzione: le applicazioni stanno diventando sempre piú complesse a causa sia dell'integrazione che del bisogno di scambio di informazioni attraverso la rete. Questo rende anche le procedure di ordinaria amministrazione piú complesse da eseguire e sembra essere un fattore di limitazione nello sviluppo

economico.

La presenza di questo problema ha motivato la comunitá scientifica a proporre sistemi che si auto organizzano. Sfortunatamente peró una loro classificazione non esiste ed in molti casi la proprietá *auto organizzazione* non è sufficentemente provata.

Nel nostro punto di vista questa situazione porta inutile confusione nel settore e puó esere migliorata.

Per questi motivi abbiamo deciso di presentare un nuovo metodo per la valutazione di sitemei auto organizzanti, basato sulla valutazione delle performance dei sistemi tradizionali, e lo abbiamo successivamente testato su un sistema reale.

Contributo

Lo scopo di questa tesi è duplice: presentare una procedura ragionevole per valuare sistemi auto organizzanti e provare che il famework Cultura Implicita puó essere usato per la loro costruzione. La tesi è divisa in quattro capitoli:

Nel primo abbiamo focalizziamo sui sistemi auto organizzanti. Diamo alcune definizione operative che ci permettono di distinguere i software auto organizzanti dagli altri. Inoltre presentiamo alcuni principi e suggerimenti da seguire durante la costruzione di un SOS, nella parte finale del capitolo.

Il secondo capitolo è sulla valuatazione di un sistema auto organizzante in generale. Prendiamo un semplice caso di studio ed estendiamo le definizioni prese dall'anallisi tradizionale delle performance .

Nel terzo capitolo riassumiamo il famework Cultura Implicita con una breve descrizione dei suoi componeti e delle loro funzioni. Successivamente motiviamo la scelta della Cultura Implicita per implementare un sistema auto organizzante.

Nel quarto capitolo descriviamo il software che abbiamo costruito utilizzando il supporto alla Cultura Implicita. Descriviamo gli agenti, i loro comportamenti, i protocolli di interazione e i diagrammi delle classi. Nelle ultime sezioni dimostriamo che il nostro sistema è auto organizzante e successivamante ne analizziamo le prestazioni.

Chapter 2

Self-organizing systems

There are many definitions of self-organizing systems, and we use them in many differnt contexts: cybernetics, information theory, thermodynamics. In agreement with [7], we can split the problem of defining a self-organizing system in three:

- how to define *self*,
- how to define an *organization*,
- how to define a *system*.

Self is a concept that comes from psychology, it is the set of an entity's features perceived as continue in time, in relation to the external world. An *organization* is a set of entities and resources that operate in an ordered and functional way. The concept of *system* is similar to the previous one, but more emphasis is given to the belonging of a single group of entities and the interrelations between elements. We can apply this definition to software systems to distinguish, in an effective and uniform way, the self-organizing systems from the others, as presented in [8].

2.1 Operative definitions

Definition 1 (Software system) *A software system is defined as a set of interacting and well-defined components.*

Definition 2 (Adaptive software system)
Let: S *a system,*
W *a criterion of acceptability,*
$\{I_\gamma\}$ *a family of input functions,*
Γ *a selection function,*
$P(\gamma)$ *a performance funtion.*

A software system is adaptive iff $S : \Gamma \to W$, so in accordance with [9], we can describe the following model for adaptivity (Figure 2.1).

A subset of the input functions is selected by Γ, and the system S is under the influence of it. The behaviour of the whole system must be observable and measurable, and indicates the effect produced by the given input. Now we evaluate the behaviour using the performance function, and using the criterion W we can accept or reject the adaptivity of the system for the given input.

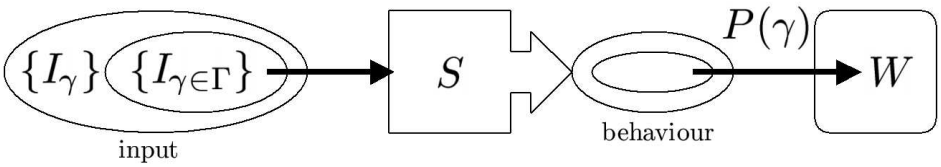

Figure 2.1: Zadeh's model for adaptivity

Definition 3 (Structure) *A structure is the property of a system by which it constrains the degrees of freedom of its components.*

Proving that a system is structured is a hard task if we use this definition alone, because it is too abstract. The concept of entropy give us a great help, in fact entropy is a measure of the degree of disorder in a general system. If a system is free to assume any state of the state space the entropy is maximal, while if there are some constraints that limit the configurations to assume, the entropy decreases. A definition of entropy is given by Shannon in [10].

Definition 4 (Entropy) *The information entropy is given by*

$$H(P) = -K \sum_{s \in S} P(s) \cdot logP(s)$$

with: K *a constant, to decide the measurement unit,*
S *the state space*
s *a state of the state space*
$P(s)$ *the probability, according to the distribution P for the system to be in the state s.*

The entropy is strictly related to the definition of the state space, thus its definition is a critical task for detecting structures and how varies in time.

Figure 2.2 represent an unstructured ant colony on the left and a structured one on the right. In this case the state space is given by the cartesian product of the state space of each ant. The state space of an ant is given by its position and the orientation of its head. In the right part of the figure we can note that the ants are located near the segment $\overline{HF}$ and the orientation of heads is towards H or towards F. Obviously the entropy is decreased.

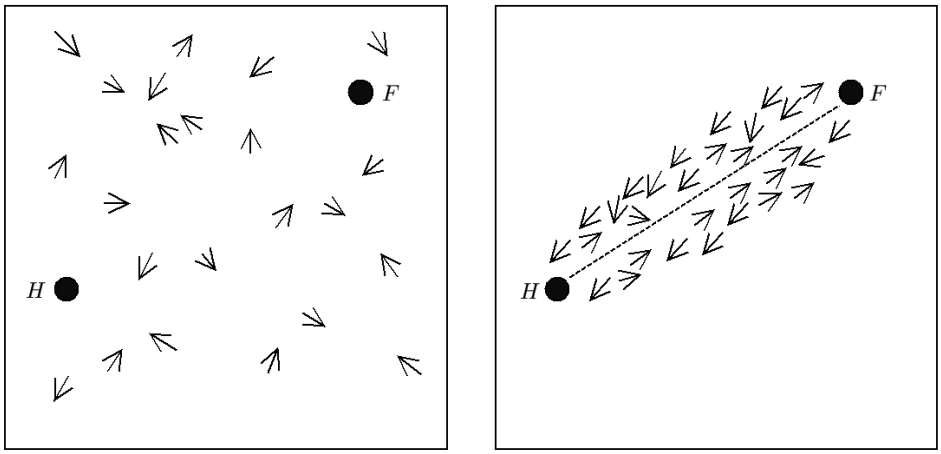

Figure 2.2: An ant colony

Definition 5 (Central controller) *A central controller for the software system S is a subset C of the components of S able to control the execution of the components in $S \setminus C$.*

If a system has a group of components like C, the system looses the ability of self-organizing: it rather consists, in two subsystem, where one organize the other. In order to proove that a paricular system lacks of a central controller, one must show that it is impossible to decompse it destructively. The decomposition theorem helps a lot in this task.

Theorem 1 (Decomposition) *If a software system S has a central controller, then S can be decomposed in two subsystems C and $S' = S \setminus C$, such that neither C nor S' can perform the original function of the whole system S.*

Note that the reverse implication is not always true.

Definition 6 (Single point of failure) *A system has a single point of failure if a problem in a single component is sufficent to inabilitate the system to accomplish tasks.*

Definition 7 (Self-organizing software) *A Self-organizing software is a software system satisfying the following:*

- *it is adaptive, according to the Zadeh's model for adaptivity,*
- *its entropy is not maximal and changes in time,*
- *it does not have a single point of failure.*

2.2 Design principles

In this section we present a set of guidelines to be taken into account when planning a new self-organizing system. This collection, given in [11], comes from the analysis of natural systems where the emergent property is easy to detect, for example ant or bee colonies.

2.2.1 On agent population

- If actors are limited in the actions, they can be simple and functional at the same time. A distributed environment induce this limitation enhancing the localization of each agent and increasing the scalability of the system.
- The developer has to implement an automatic procedure to remove obsolete informations. This allows the environment to be more dynamic and prevent actors from worrying about information cleaning.
- The developer has to prefer many small agents instead few large ones. If angents are limited in number, they have to cover more problem space. On the contrary if there are many agents the localization of interaction is better supported.
- The developer has to map components to agents and not functions to agents. This allows to keep an agent responsible only for a small part of the problem. Conversely, each agent need much more informations to produce output, and trying to include special cases its complexity increases.

2.2.2 On agent interactions

- Do not think in term of transition between discrete and autonomous state, but in term of information flows. To fulfill the requirement of multiple interacions among the agents, it is also necessary to include closed loops of informations.

- The mechanism of moving information from an agent to others needs some constraints to prevent from redundancy. It can be done in two ways: implementing positive and negative feedback loops, or terminating every agent after a specific time period. This second option is called programmed agent death.

- Randomize the agents' behaviour. Sometimes the localization of agents is sufficient to act in different way, but a general rule is to incorporate stochastic elements in the agent's decision tree. In this way identical code will diversify its run, so the decision process becomes a random variable. It follows a probability distribution and changes in time related to experience.

2.2.3 On supporting the emergence of desired functions

- Let each agent support multiple functions, and provide a mechanism for switching among them. In this way an agent can contribute to the solution of a wider part of the problem.

- Let each part of the problem to be solved, be composed of many interconnected agents. This avoids single point of failure and keeps agents simpler.

- Give agents the possibility to measure their level of task completion, to receive feedback on their duties. One of the major challenges is finding local interactions strictly correlated with the general behaviour of the system. Finding such interactions allows to make intelligent decisions based on a subset of the state space.

- Provide an automatic mechanism for selecting among alternative behaviours. In this way agents can modify their behaviour or change the composition of the population.

2.3 How to guarantee a high task completion rate

To be sure that a self-organizing sistem performs the right tasks, a control mechanism is needed. This control should be distributed within the system in agreement with the self-organizing framework.

The control mechanism can be seen as a mediator which on the one hand guarantees the right interation of the element of the system, on the other hand helps producing the desired performance. However a strict control over the system is not possible without loosing the self-organizing property.

In order to build a control, the designer have to distinguish between friction elements and elements increasing the synergy. The former should

be minimized while the latter should be maximized. The performance of a general mediator can be measured using the following equation:

$$\sigma_{sys} = f(\sigma_1, \sigma_2, \sigma_1, \ldots, \sigma_n, w_0, w_1, w_2, \ldots, w_n)$$

with:
- σ_{sys} the satisfaction of the whole system,
- f $f : [0,1]^n \cup R^{n+1} \rightarrow [0,1]$,
- σ_i the individual statisfaction of agent i,
- w_0 a constant value,
- w_i the weight of agent i,
- n number of agents in the system.

If the system is homogeneous f is the weighted sum of the individual satisfaction, it can be any function for heterogeneus system.

Another property of a control mechanism is the adaptivity. Since the system changes in time, the control mechanism must deal with those changes without external input.

The following mechanisms can be applied when an agent A is negatively affected by another agent B [12].

Tolerance A can tolerate B by modifying itself, for example move to another location, find other resources, or modify its behaviour. After an application of the tolerance method σ_b remains unchanged, the friction between A and B decrease and σ_a increases.

Courtesy It is the opposite of tolerance. Agent B have to modify itself in order to not reduce σ_a.

Compromise It is a combination of tolerance and curtesy. Here both A and B modify their behaviour in order to reduce friction. The application of the compromise method is common when actors are similar.

Imposition A kind of courtesy forced by the system. The controller can apply imposition by limiting B actions or imposing internal changes.

Eradication A special case of imposition: the controller kill the agent B.

Apoptosis The suicide of B, without input by the controller.

2.3.1 Metods for increasing synergy

The methods proposed in this section are useful to increase the satisfaction of the system σ_{sys}, reducing some individual satisfactions.

Cooperation It is similar to courtesy but the target is different. Here two or more agent change for the advantage of the system.

Individualism An actor can try to rise its satisfaction if it increases the satisfaction of the system. Conversely a controller must deny that increasing same individual satisfaction will decrease σ_{sys}.

Altruism An altriustic agent A reduces its satisfaction to increase σ_{sys}. The role of the controller here is to guarantee that the relative incrase in σ_{sys} if greater than the decrease in σ_a.

Exploitation The controller decrease an agent satisfaction to increase σ_{sys}.

2.4 Organization detection

It is part of the common experience that social networks form communities of strictly correlated entities. This structure is called *modular* (Figure 2.3). Moreover it is widely belived that the modular structure of a networks affect the functionality of the network itself [38].

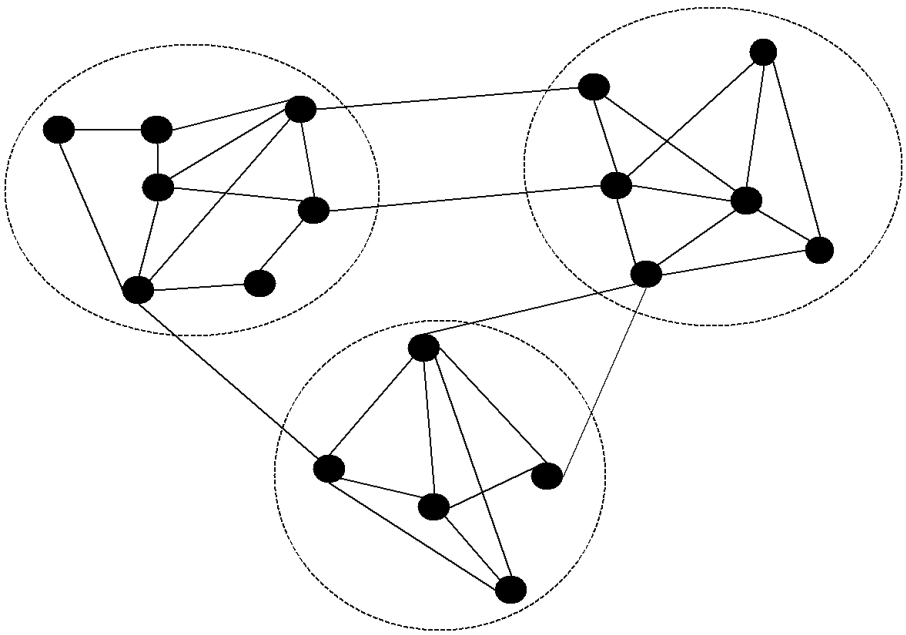

Figure 2.3: An example of modular structure

The mechanisms by which the modular structure naturally emerges in complex systems have not be discovered so far, but some results are known. Recently Solé and Fernández have pointed out that networks without any input from extern are able to build a modular network [39]. The need of finding algorithms to identify modules inside a network is high and some new algorithms have been presented recently. Those algorithms are based

on many different ideas. In table 2.1 we present a list of them. The features and how the details on each algorithm works are presented after the table.

Algorithm's authors	Reference	Time Complexity	Year
Eckmann & Moses	[16]	$O(m \cdot k^2)$	2002
Girvan & Newman	[22]	$O(n^2 m)$	2002
Bagrow & Bollt	[27]	$O(n^3)$	2004
Capocci et al.	[28]	$O(n^2)$	2004
Donetti & Muñoz	[25] [26]	$O(n^3)$	2004
Fortunato et al.	[23]	$O(n^4)$	2004
Guimerá et al.	[20]	param. depend.	2004
Latapy & Pons	[17]	$O(n^3)$	2004
Newman	[19]	$O(n \cdot \log^2 n)$	2004
Newman & Girvan	[18]	$O(m^2 n)$	2004
Radicchi et al.	[24]	$O(n^2)$	2004
Reichardt & Bornholdt	[31]	param. depend.	2004
Wu & Huberman	[29]	$O(n+m)$	2004
Zhou & Lipowsky	[33]	$O(n^3)$	2004
Duch & Arenas	[21]	$O(n^2 \log n)$	2005
Palla et al.	[30]	$O(\exp(n))$	2005

Table 2.1: Algorithms for finding modules inside a social network: n number of nodes, m number of links, k average degree.

2.4.1 Eckmann & Moses

This algorithm was devaloped to find the *meta-informations* localized in the connectivity of the World Wide Web. Meta-informations are those based only on the structure of the graph of connections. This algorithm is the composition of four well-known concepts:

- clustering,
- co-links,
- triangles,
- curvature.

In detail the procedure is the following:
`1. Group in a sigle node the pages connected to a home page.`
In this way all the pages of a site are gathered together, the size of the graph is reduced, and proximity links are erased.
`2. Find co-links between nodes.` Two nodes A, B are co-linked only if A is linked to B and viceversa. The presence of such a relation implies

mutual recognition.
3. `Find triangles.` A trinagle is a set of three nodes co-linked one to each other. Since this relation is extremely rare in random network with bounded cardinality, is a good signal of cooperative content.
4. `Quantify the nodes' local curvature parts of one triangle or more.` The local curvature is defined as

$$c_n = \frac{2t_n}{(v_n - 1) \cdot v_n}$$

with: t_n number of triangles containing n,
v_n is the number of links leaving the node.

2.4.2 Girvan & Newman

This algorithm instead of finding and measuring the most central nodes by adding them progressively, starts from the whole graph and removes the less central nodes, one to one. The key idea is:

"If a network is organized in communities, all shortest paths between the communities connect only few edges".

Moreover removing those edges, we are able to separate each group rom the other. The details of the procedure are the following:

1. `Calculate the betweenness for all edges in the network.`
2. `Remove the edge with the highest betweenness.`
3. `Recalculate the betweenness for all edges affected by the removal.`
4. `Repeat from step 2 until no edges remain.`

2.4.3 Latapy & Pons

The approach of Latapy and Pons follow the well documented intuition that small lenght random walks in a graph tend to get trapped inside the densely connected parts of it. The innovation is the implementation of a measurement of the structural similarity between nodes based on the random path created. The procedure in detail is the following:

1. `Start with a partition where each vertex is alone.`
2. `Compute the distance between all adjacent vertices.`
3. `Choose two comunities`, according to a criterion based on the distance between comunities.
4. `Merge this two comunities and create a new partition.`
5. `Update the distance between communities.`
6. `Repeat 3-5 for n-1 step.`

To evaluate the quality of the partition found, the ratio η_k is used.

$$\eta_k = \frac{\Delta\sigma_k}{\Delta\sigma_{k-1}} = \frac{\sigma_{k+1} - \sigma_k}{\sigma_k - \sigma_{k-1}}$$

with: k a step in the process of clustering,
σ_k the mean of the squared distances between each vertex and its community. See [17] for more details.

2.4.4 Duch & Arenas

This algorithm is a divisive one, if fact it starts with the whole graph and progressively divides it in smaller and densely connected subgraph. It uses a heuristic search to find the optimal modularity value Q. According to [18] the definition of Q is the following:

$$Q = \sum_r (e_{rr} - a_r^2)$$

with: r a comunity,
e_{rr} the fraction of links connecting two nodes inside r,
a_r the fraction of links that have one or both vertices inside r.

In detail the heuristic proposed evolves as follows:
`1. Split the nodes in two random partitions.` The subdivision creates two comunities, understood as connected components.
`2. The system itself moves some node from a partition to the other, increasing` Q`.` Every movement causes the recalculation of the fitness measure.
`3. Repeat 2 until the maximal` Q `is reached.` Then deletes all links between the partitions and proceeds recursevely with each partition.
`4. The algorithm ends when` Q `cannot be improved any more.`

2.4.5 Palla et al.

The main advantage of this algorithm is that it allows to find overlapping communities. Two communities overlap if some nodes belong to two or more communities at the same time, and this characteristic is present in many actual networks.

The algorithm finds the groups of nodes fully interconnected (cliques) in the graph of relations, but this restriction can also be relaxed allowing groups not fully connected.

2.5 Examples of self-organizing systems

2.5.1 Ants

Those animals are interesting from a computer science' point of view because they are social insects and form highly organized colonies of millions of

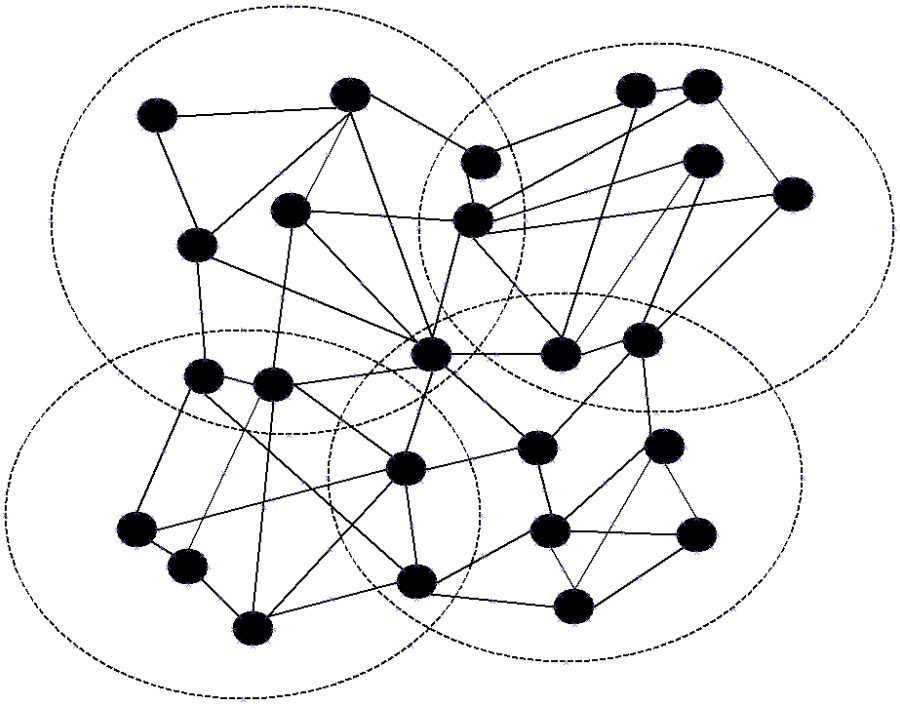

Figure 2.4: An example of overlapping comunities

individuals. The interaction between organisms is so high that sometimes a colony is described as a unique superorganism.

Communication

The main channel of comunication is through pheromones. They are chemical substances left on the ground during the transit. The concentration of those pheromones decreases in time, and needs an uninterrupted supply of new material to maintain or increase its level. For example when an ant finds a new source of food, she will leave pheromones on her way to the nest. After a short period other ants will follow the pheromones trail and reinforce it returning home. When the source is depleted, the pheromones trail is no longer reinforced and decrease its intensity. In this way dissipation of pheromones acts as a cleaner for information out of date.

Behaviour

Ants are the only group of animals except primates and some other mammals which interactive teaching behavior has been observed so far [15]. The process of learning is called *tandem running*. In a queue, each entity tend

to maintain the same distance between the previous and the following one, leading the swarm to the source of food.

Cooperation and competition

The aggressiveness and propension to expansion vary in each specie. Some are known for attacking and taking over the colonies of others species, while others attack colonies to steal eggs or larvae. Amazon Ants depends on captured worker ants to find and transport food to the nest. Some species engage huge battles between colonies in order to expand their territory and control the food supply. Those battles often cause thousands of deads and they performed a central role in the development of ants' cleverness.

2.5.2 Pollination

It is an important biological process, where the male gametes present in a flower are transferred to another flower which contains the female gametes. In every process, pollination is the result of the interation of two kind of actors [13].

Flower can be a source of pollen or a receiver. In a source a variable quantity of tranferable material is present, while a receiver can collect this material.

Vector is responsible fpr the movement of pollen. In nature there are many situation where the pollination is done by insects, and only a few cases where the random pollination of the environment is used. In some cases the dependency between the plant and the pollinator is so high that each of them depend from the other for surviving.

The central problem of the pollination process is how to attract the pollinator to the right flowers. In nature it is done using combination of colors and various fragrances. Every kind of vector, in fact, is attracted by a specific combination of the two. Moreover insects have generaly limited visual capabilities and strong smell so the effect of the two actractors changes in space.

In the pollination process a method for rewarding vectors is also present. Flowers give nectar and other substances to vectors as a result of its visit, and vectors visit only similar flowers during a trip. In this way the possibility of pollination is high and the reward collected is high too.

This system, autonomously evolved in milions of years, is characterized by six useful properties.

Self-configuration. When adding new flowers or vectors, the system is able to configure itself without human activity.

Self-optimization. Faster vectors will collect more rewards, and flowers providing more food will be visited more.

Self-healing. If some flowers acts no more, the system is able to cope with this change, the vectors simply search for new flowers to visit.

Self-protection. The reward is only provided to vectors carrying pollen, there is no way to be rewarded without carrying it.

Self-adaptation. If a plant is not able to attract vectors it is going to die over a long run.

Self-organization. The system presents all three elements of self-organizing systems: it is adaptive, it has not a single point of failure, and its entropy changes in time.

2.5.3 Modular robots

In systems composed of modular robots, each of them is a set of module able to reconfigure and assembly itself [14]. One of the first applications of modular robots was building a machine to operate in autonomous way inside a nuclear plant. Since robots can modify their structure, the domain of the application is wide. The intelligence of a robot is decentralized and located in each module.

Some variations of the system were proposed in the past years.

Manually assembled vs. self reconfigurable. In the first case the robots need external intput to modify their structure, in the second they are more skilled and each set of modules is able to assemble itself.

Homogeneity vs. heterogeneity. Each module must be able to cope with a subset of tasks called basic tasks (first case), on the contrary a module is specialized in only one of them.

The rules to follow for making modular robots are quite simple.

- The artificial intelligence is the result of the linear combination of many elementary behaviours.
- Each intelligence is evaluated according to progress and results.
- Each elementary behaviour has at least one capability not performed by any combination of other elementary behaviours.

The set of all elementary behaviours of a modular robots system is called a substrate. A possible substrate is presented in table 2.2.

Behaviour	Goal
Safe Wandering	The ability of moving without colliding with obstacles
Following	The ability of retracing a path of another module
Dispersion	The ability of maintaining a minimum distance with another module
Aggregation	The ability of maintaining a maximum distance with another module
Homing	The ability of finding a particular location in the state space
Connecting	The ability of interconnecting one to each other
Disconnecting	The ability to terminate the connection with a previously connected module

Table 2.2: An example of substrate

Chapter 3

The evaluation of a self-organizing system

3.1 Our procedure for the evaluation

We can model a self-organizing system at least in two ways:

- the first focusing on the internal mechanisms of the software, it is useful to control how decisions are made;
- the second representing the whole system as a function and analyzing its inputs and outputs.

If we adopt a white box view (Figure 3.1), we are interested in knowing if the internal organization of our software changes in time, and in agreement with the definition, if it has a single point of failure.

If we adopt a black box view, our system is depicted as in figure 3.2 and can be evaluated as a traditional software system.

Following four aspects are of our interest:

Task completion: the probability that a user will receive the required service at the desired quality.

Availability: the fraction of time in which the system is able to receive and process inputs.

Survivability: is measured when the system operates in unusual conditions, is high if it responds well.

System performance: measures the utilizzazion of each resource and the throughput of the system.

We propose now a two-phases procedure which allows us to build and evaluate a general self-organizing system. This procedure is based on the tautology: *"any system tends to its more probable state"*.

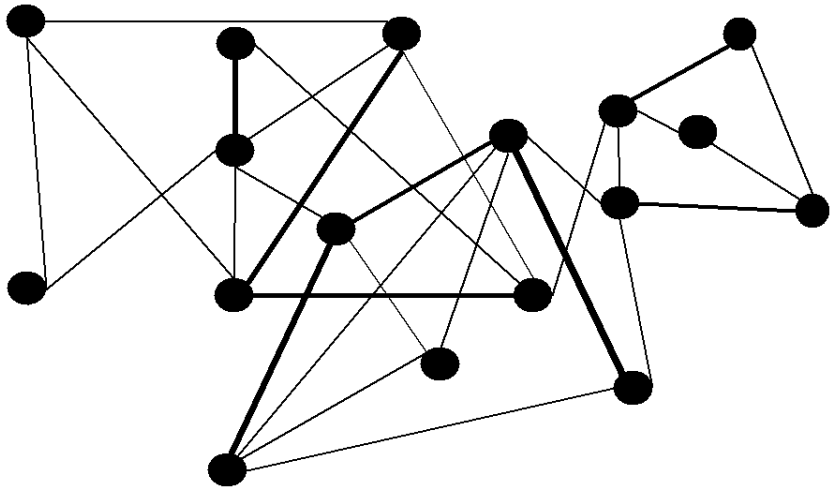

Figure 3.1: A white box view of a self-organizing software, if the connection is stronger, the connecting line between two agents is thicker.

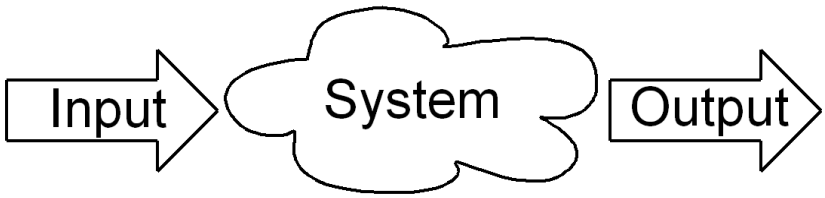

Figure 3.2: A black box view of a self-organizing software.

Requirements of the procedure A software system and its requisites, the knowledge of the domain of the application, and a list of cases that can generate problems in the software.

Phase 1: Here the system processes a huge amount of inputs, with the objective of building up the internal structure. This phase takes a variable time to be performed: it begins with the first input and terminates when a reasonable stability in the organizzation is reached, and the task completion rate is acceptable. Our goal is building up an efficent software able to cope our inputs. In this phase the following indexes are of our interest:

- the organization stability,
- the task completion rate,
- the availability of the system,
- the system performance.

Phase 2: We use now the list of cases that will probably generate problems at runtime. This phase ends when a reasonable stability in the organizzation is reached. An useful method to follow, comes from a review of a widely-used one. The original procedure is taken from [35], and it is used for finding errors in a general application.

1. Take the probable problems and group similar ones.
2. For each group of probable problems P_i select the problem p_i where the highest probability to fail is expected.
3. Set a list of requirements R_i for each p_i,using the requisites of the software.
4. Set two naturals $m_i \leq n_i$ for each p_i.
5. Run each p_i exactly n_i times, in random order.
6. Store the outputs in chronological order.
7. Control for each i: if the last m_i outputs for p_i are similar and satisfy the requisites R_i then we claim that the software is able to cope with all kinds of problems in P_i, otherwise not.

In this phase we are interested only in the survivability of the system.

3.2 A Simple case study: traffic lights

Steering vehicles in a city is a very complex task, because to improve the efficency there is the need of coordination of many actors. The traditional approach for this kind of problem is to maximize the efficency for a particular distribution of traffic, hoping that the model will not work so bad if the distribution changes. The main limitation of this approach is that traffic canges distrubution constantly. For those reasons many studies have proposed self-organizing systems that handle this problem as an adaptation problem.

Some rules have been introduced to limit the behaviour of the vehicles, for example the side of the street where driving is possible, the rules for precedence, the signals, and trafic lights to regulate the crossroads.

Unfortunately, if the car density is too high, there is no solution for the traffic congestion problem. Thus our goal becomes finding solution to improve the traffic management.

In the past, to solve those problems, the consuetude was to find the most appropriate timing in traffic lights, but this method does not count on the current state of the traffic, and is unable to handle abnormal situations in a suitable way. In real life abnormal situations are quite frequent, for example queues near a stadium before and after an important match, queues

in the morning to reach the workplace or queues on Sunday near commercial facilities.

To build the basis for a self-organizing system that tries to solve traffic congestions is quite easy because it is simply composed of two classes of actors.

Car: its goal is to travel from a place to another minimizing the time spendt.

Traffic light: its goal is to avoid long queues in the crossroads.

Our model for traffic is based on the following assumptions.

- The number of streets and the position of traffic lights is constant.
- The total number of vehicles in the city is constant but their localization varies in time.
- Collisions can happen.

To proceed with the two-phases evaluation we have to provide the requirements of the application and a list of possible problems where we can test our system. Our requirements are basically two.

- In general we want to ensure an acceptable speed during movement.
- Our system must respond well in abnormal conditions, since they are frequent in real traffic.

We have selected the following possible problems:

- many casualities happen in the same time,
- a specific part of the city is crowded while the others are empty,
- many cars have to reach the same place,
- for a while a part of a street is closed.

3.3 Stability in similar tasks

Finding stability in a self-organizing system means that similar tasks given to the system are performed using similar actions. As for every organized system, the evolution in time of the agents' behaviour and their actions allows to point out some mechanisms of action-effect. A useful mechanism to find stability is composed of two steps:

```
1.  find two or more actions with similar initial situation
    and final situtation;
2.  verify if the task done is similar.
```

An important part of the algorithm is the definition of the similarity function. Our first choice was considering two task similar if they are equal, but finding different tasks with the same initial and final situations is extremly rare in a self-organzing system. So we have chosen a new similarity function depending on the application context.

- The initial situation is well represented with the current position of the car inside the city grid.
- The final situation is the arrival position after the journey.
- The task done is the path followed to reach the arrival position from the departure position.
- Two positions are similar if are reasonably close one to each other.
- Two paths are similar if have similar lenght and reach similar cells in the same order.

Using the current implementation of the system for implicit culture support we can easily implement this similiarity function. we only need to set properly the file *similarity_config.xml*.

3.4 How to guarantee the absence of a single point of failure

Here we want to apply the decomposition theorem previously stated, for showing that our system has not a single point of failure. A common way to operate is the following:

```
1. repeat
2.    take a component of the system and remove it;
3.    show that the system performs the same tasks as before;
4.    re-add the component to the system;
5. until each component has been tested.
```

The result of the procedure is a boolean value. If after removing some component the system is unable to perform some tasks then our system is not self-organizing, in agreement with the definition.

In our case study the system fails for example if removing a traffic light some cars stop for a too long time or if causalities happen frequently.

3.5 Task completion

As explained before, calculating the task completion rate of a system consists in finding the proportion of tasks correctly processed which the time inside

the system does not exeed a threshold given. Clearly this rate canges in time depending on the system' ability to solve the input given. Since the output of our system depends strictly on its experience, we can apply several methods for forecasting. In particoular we have selected three of them that suit well to our problem.

Linear moving average. The most simple of the three. It simply compute the average of the performance measured in a fixed interval.

Logistic fit. We try to model the performance measued with logistic a well-known curve that has some good properties:

- The explicit formula of the derivative in time is known.
- The derivative depends on two parameters only: the improovement rate that measures the aptitude of the system to learn and to organize and the limiting rate that measures the limitations of our system in solving problems.
- Has an horizontal asymptote, it identifies the steady state.

Smoothing functions. They are a wide class of curves that try to clean up the data from rumors. In this class we have selected *lowess*: its degree of smooth is controlled by threee intuitive parameters and is already implemented and well tested in our evironment for analysis. For more complete imformation about this curve see [3];

In the case study, the task completion measures the fraction of vehicles reaching their destination in acceptable time. Our method allows to distinguish between systems with same task completion rate at steady state (Figure 3.3). Simply comparing the parameters we can say that one of the systems is twice fast as the other in the learning. In the general case, we are able to quantify how better a system learn than another.

3.5.1 The process of logistic fitting

Logistic is a well-known ordinary differential equation, which is also known the analitycal solution. It is suitable for us because like task completion, it is always included in a closed interval and allows to explicit easily the improovement rate and the limiting rate. The logistic model is the following. Let A be the improvement rate, B the limiting rate, and α the initial value:

$$\begin{cases} y' = Ay - By^2 \\ y(0) = \alpha \\ A, B, \alpha > 0 \end{cases}$$

If $\alpha < \frac{A}{B}$ the shape of the curve is like figure 3.4. Moreover we derive from the model that the initial value is α while in the final stage the curve

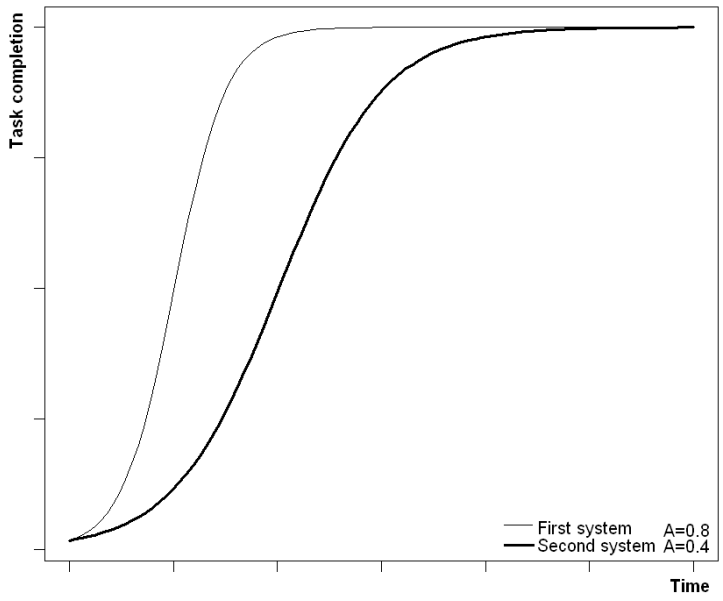

Figure 3.3: The task completion trend for two different As

is limited by $\frac{A}{B}$. We use now some values obtained from the linear moving average method:

$\alpha \leftarrow$ the initial value,
$\frac{A}{B} \leftarrow$ the final value.

We use the following definition for the error:

$$\varepsilon = \sum_{i=1}^{n} \left(\frac{A\alpha}{B\alpha + (A - B\alpha)e^{-Ai}} - y_i \right)^2$$

with: n number of measurements,
y_i real values,
A improvement rate,
α initial value,
B limiting rate.

To find the values for A and B that minimize the error we use a binary search algorithm, when we first need to set a range for A. The solutions found are similar to figure 3.5.

3.6 How to determine the availability

The procedure for analyzing the availability of the system is quite similar to the task completion one, because both of them uses information about

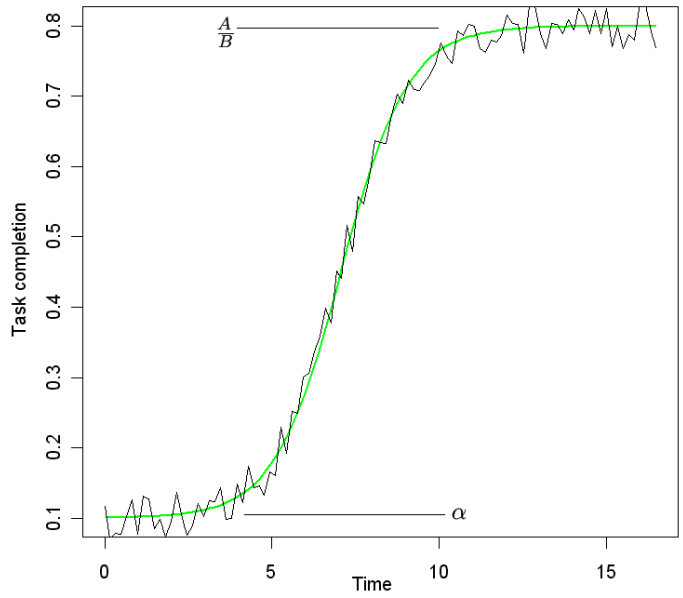

Figure 3.4: A very well fitting to a logistic curve

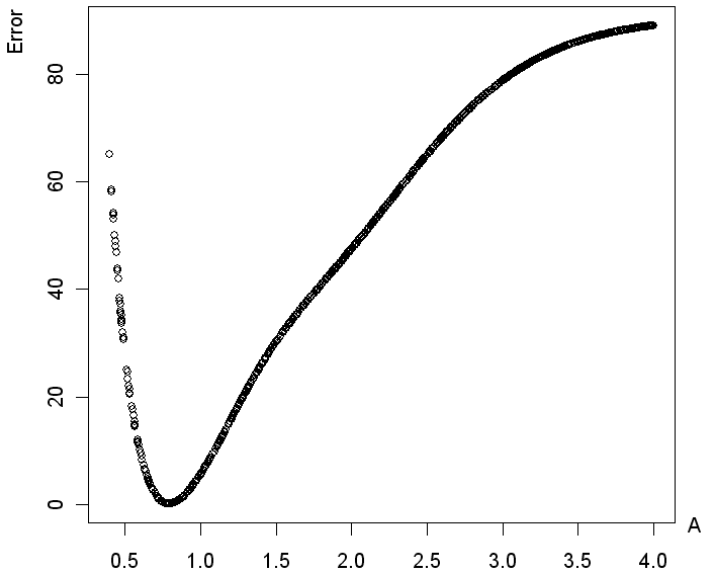

Figure 3.5: The trend of the error for different As

past experience to try to predict how the system will behave in the future. Moreover the availability changes in time, depending on the input and the experience of the system. For these reasons we can apply exactly the same

methods used for determining the task completion rate, in the same way.

The availability is the fraction of time where the sytem is ready to process new inputs. The results of this analysis is the availability at steady state, calculated using linear moving average methods. If the trend of availability follow a logistic law then we can find the improvement rate, the limiting rate and the error of the process.

In the case study example we say the system is available if new cars can start immediately to travel in the city.

3.7 How to evaluate the survivability

The survivability of a system expresses its ability to react well in abnormal conditions. To control if our system has such a property, we test our system in the abnormal condition we have previously found. We choose the possible problems that have the highest probability to crash the system and we repeat them n times, with n fixed.

A similar method is described in [35], when a system is tested seaching for errors. There are five rules to follow to execute a survability evaluation:

- a test case is valid if it has a high probability to fail,
- a test case has to be the best of its category,
- a test case has to be neither too simple nor too complex,
- we do not have to use many test case, but we have to consider all the functionality,
- we want to minimize the redundancy.

The result of the survability analysis can be efficiently summarized in a table. Note that the value n need to be set before the evaluation, and a bad choice of it can cause a negative result even if the system is able to cope with the input.

3.8 System performance

Measuring the perfomance of the system often means measuring the utilization of the resources or the system throughput. Both of them are interesting for our evaluation because they focus on different aspects:

- the utilization of the resources is useful to control if our system uses a constant amount of resources during executions.
- the system throughput indicates the improvement of the system; in our case study it is the number of cars reaching the arrival position in a fixed time slot.

3.9 Useful characteristics of the method

- Guide the developer from the first simulation to the performance evaluation.
- Allows us to apply some concepts coming from the traditional performance evaluation: task completion, survivability, system performace, throughput, availability and stability.
- Gives the possibility to evalutate how the system works in the presence of abnormal conditions, evaluating its survivability.
- Keeps track of the time needed to reach a stable organization, gives us the organizing rate.
- Allows us to judge if the system has a single point of failure or none.
- Allows us to quantify the availability of the system in the general case and in particular cases.
- Needs few simple data structures to be completed.
- Uses the variations of the system' behaviour to determine if the system is sufficently stable.
- All the algoritms needed have polinomial complexity.

Chapter 4

Implict culture

The culture of a group is a set of notions, traditions and behaviours that characterize that group in a unique way. A culture is implicit if some entities behave like the group without explicitly knowing the rules of the comunity. The primitive concepts of implicit culture are agents and object, and we refer to them with strings.

The agentss operate in the set of possible actions, those actions are chosen depending on the state of the environment and to the state of the agent itself.

The objects are the target of the actions.

We give now some definitions, to formally define the meaning of implicit culture.

4.1 Definitions

Definition 8 (set of agents) *A set of agents $\mathcal{P}$ is a set of agent-name strings.*

Definition 9 (set of objects) *A set of objects $\mathcal{O}$ is a set of object-name strings.*

Definition 10 (environment) *The environment $\mathcal{E}$ is a subset of the union between the set of agents and the set of object. $\mathcal{E} \subseteq \mathcal{P} \cup \mathcal{O}$*

Let *action-name* be a type of strings, and let $E \subseteq \mathcal{E}$ and let s an action-name.

Definition 11 (action) *An action α is the pair $\langle s, E \rangle$,where E is the argument af $\alpha(E = arg(\alpha))$*

Let $\mathcal{A}$ be a set of actions, $A \subseteq \mathcal{A}$ and $B \subseteq \mathcal{E}$.

Definition 12 (scene) *A scene σ is the pair $\langle B, A\rangle$ where for any $\alpha \in A$, $arg(\alpha) \subseteq B$; α is said to be possible in σ. The scene space $\mathcal{S}_{\mathcal{E},\mathcal{A}}$ is the set of all scenes.*

Let T be a numerable and totally ordered set with the minimum t_0; $t \in T$ is said to be a *discrete time*. Let $a \in \mathcal{P}$, α an action, and σ a scene.

Definition 13 (situation) *A situation at the discrete time t is the tuple $\langle a, \sigma, t\rangle$: "a faces σ at time t".*

Definition 14 (execution) *A situation at the discrete time t is the tuple $\langle a, \alpha, t\rangle$: "a performs α at time t".*

Definition 15 (situated executed action) *An action α is a situated executed action $\iff$ exists a situation $\langle a, \sigma, t\rangle$, where a performs α at the time t and α is possible in σ.*

The function $F_{\mathcal{E}} : A \times \mathcal{S}_{\mathcal{E},\mathcal{A}} \times T \to \mathcal{S}_{\mathcal{E},\mathcal{A}}$ describes how the situation at time $t+1$ is determined:

$$\sigma_{t+1} = F_{\mathcal{E}}(\alpha_t, \sigma_t, t)$$

with $\sigma_t, \sigma_{t+1} \in \mathcal{S}_{\mathcal{E},\mathcal{A}}$; $\alpha \in A$.

Let the action performed by the agent a at time t, $h_{a,t}$ be a random variable that assumes values in $\mathcal{A}$

Definition 16 (expected action) *The expected action of the agent a is the expected value of the variable $h_{a,t}$, that is $E(h_{a,t})$.*

Definition 17 (expected situated action) *The expected situated action of the agent a is the expected value of the variable $h_{a,t}$ conditioned by the situation $\langle a, \sigma, t\rangle$, is $E(h_{a,t}|\langle a, \sigma, t\rangle)$.*

Definition 18 (party) *A set of agents $G \subseteq \mathcal{P}$ is said to be a party.*

Let $\mathcal{L}$ be a language to express the environment, and let G be a party.

Definition 19 (cultural constraint theory) *The cultural costraint theory for G is a theory expressed in $\mathcal{L}$ that predicts the expected situated actions of the members of G.*

Definition 20 (group) *A party G is a group if there exists a cultural constraint theory Σ for G.*

Definition 21 (cultural action) *An action a is a cultural action for the group G if exists $b \in G$ and $\langle b, \sigma, t\rangle$ such that*

$$\{E(h_{b,t}|\langle b, \sigma, t\rangle) = \alpha\}, \Sigma \nvdash \bot$$

with Σ cultural constraint for G.

Definition 22 (implicit culture) *G ang G' are in the implict culture relation* $\iff$ *G is a group and the expected situated actions of G' are cultural actions of G.*

Culture is represented by a theory. The theory consists of one or more rules of the form

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

with: A_i antecedents of the rule,
C_j consequences of the rule.

Definition 23 (implicit culture phenomenon) *the implicit culture phenomenon is the pair (G,G') where G and G' are related by the implict culture.*

To let implicit culture phenomenon happen, we need a software that implements the concepts given. It is called Ic-Service and it is developed in java [4].

4.2 The system for implicit culture support

It provides a set of primitives to implement in a simple way agents with implicit culture capabilities. Basically, it is composed of three sub-systems (Figure 4.1).

The observer allows to store the observation about action performed by the agents and guarantees the access to the observations.

The inductive module analyzes the data base of the observations and, applying data mining techniques, tries to discover rules among data.

The composer uses the informations stored by the observer and the rules discovered by the inductive module to produce suggestions. Following one of those suggestions means having a beaviour consistent with the culture.

The first definition of the framework is given in [1].

4.2.1 The observer in detail

The current version of SICS allows to store observation in both databases and xml files. By modifying few parameters it is possible to switch among them simply. Every observation stored is composed of:

- the agent who performed action;

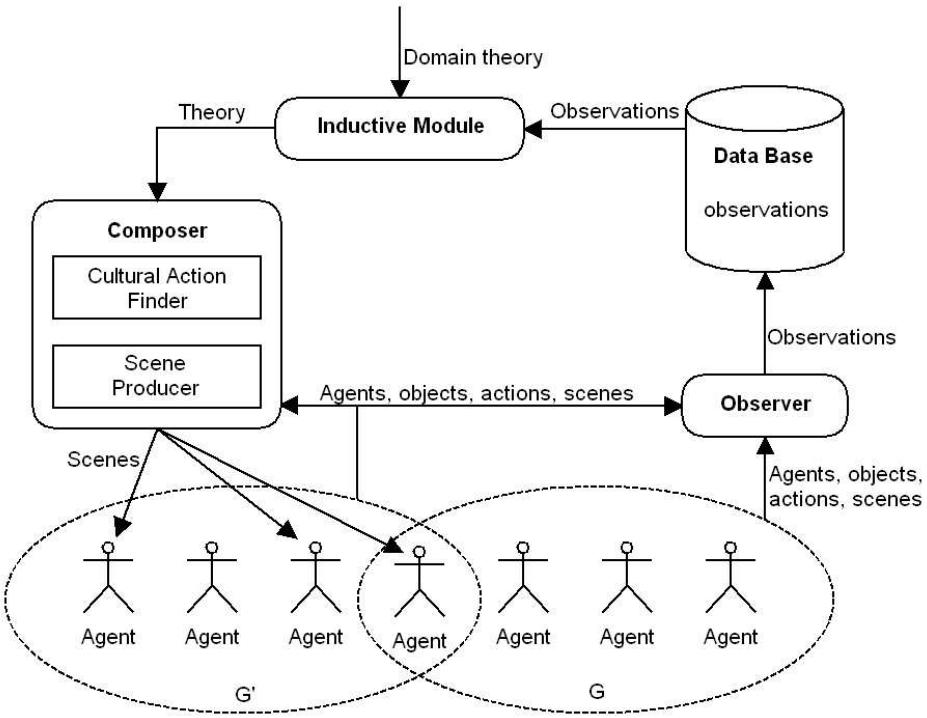

Figure 4.1: The architecture of the System for Implict Culture Support

- the scene faced by the agent;
- a set of possible actions;
- a set of objects;
- the action performed;
- the time when the action was executed.

4.2.2 The inductive module in detail

It is composed of two subsystems (Figure 4.2).

- The first extracts new associations from the observations. The current implementation the Apriori algorithm to produce associative rules.
- The second updates the old culture with new rules just discoverd. Moreover it allows to manually insert an initial set of rules and clear automatically out of date ones.

The implementation of the inductive module is presented in [4].

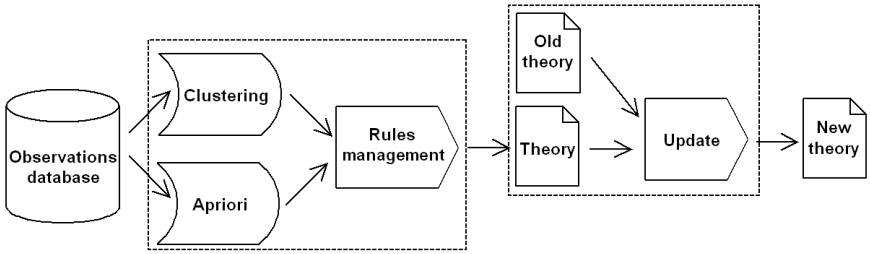

Figure 4.2: The structure of the inductive module

4.2.3 The composer in detail

Its goal is to change the current scene in such a way that the agents are more likely to perform cultural actions. It is made of three subsystems (Figure 4.3).

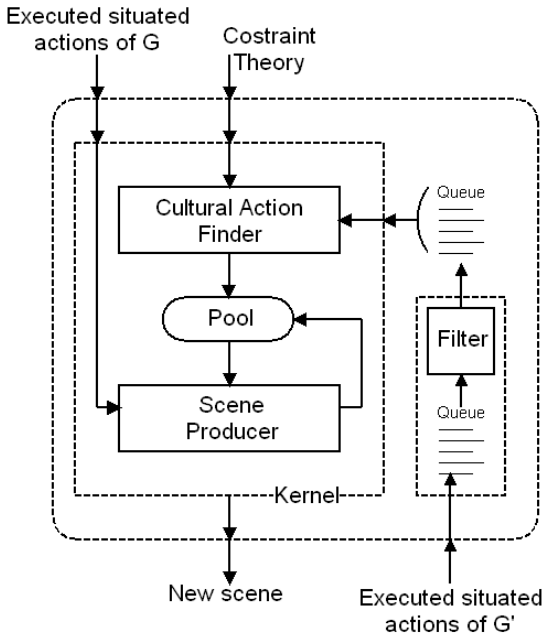

Figure 4.3: The structure of the composer

The cultural action finder: it selects from the database of observations, actions that match antecedents of the rule. Once found such actions, it returns the consequent of the rule as cultural action.

The pool: it manages cultural actions taken as input from the cultural

action finder and from the scenes producer. It solves possible conflicts among them.

The scenes producer: it takes as input the output of the cultural action finder and searches in the database for actions similar to the cultural action found. After that it tries to define which scenes lead to that action. Those scenes are subsequently returned.

4.3 Why implicit culture

We have chosen the implicit culture for reasons related to the easiness of use and to the possibility of personalization.

In the implicit culture framework the environment is dynamic because agents change it by performing actions. Also their actions are recorded as observations which take part in the next decision processes. This behaviour is well-suitable for self-organizing systems because they need a mechanism for building up a stable organization starting from disorder.

We consider the procedure of learning from observations very difficult to implement. Fortunately it is already implemented in the implicit culture framework. Moreover some software systems have been already implemented with implict culture capabilities, and they accomplish well the tasks given. For example:

- Implicit: An AgentBased Recommendation System for Web Search [5]
- A MultiAgent System that Facilitates Scientific Publications Search [6]

The procedure for testing stability can be performed with few lines of code because a similarity function is already implemented in the SICS core, it is easily configurable by xml files.

In the system we are going to build, all the agents are at the same experience level. In the IC framework there are two distinct groups of agents with different capabilities and experience. We resolve this possible problem defining the two identical groups.

Finally we want to test the framework in the field of self-organizing systems to promote the software and to improve its usefulness.

Chapter 5

A Self-organizing system based on the implict culture framework

5.1 Description of the System

Our system is composed of three classes of agents:

- car,
- semaphore,
- city.

An agent of type car have the goal of reaching its destination as faster as possible, avoiding collisions in the meantime. It moves through adjacent cells if the target cell is free respecting the indications of the semaphores. If a casuality occours, the car has to restart its journey from the beginning (Table 5.1).

Objectives	Actions
Reach the destination Avoid casualities	Move through adjacent cells Inform the city of its journey Learn from other agents Restart in case of crash

Table 5.1: Objectives and actions of Car agents

An agent of type semaphore manages the traffic in a specific location of the city and switches its light in an endless loop. Its goal is maximizing the speed of the cars in its intersection, avoiding long queues. The main actions

to achieve it is to change light periods (Table 5.2). Every semaphore has only a limited view of the traffic inside the city, but in general should be aware of the activities of the other semaphores.

Objectives	Actions
Manage the traffic in a single intersection	Change position in the city (not implemented)
Be useful for the traffic management	Change light periods

Table 5.2: Objectives and actions of Semaphore agents

The system must ensure that exactly one agent of type city is active on the platform. For this reason an agent of this type has to control its uniqueness on startup. Moreover an automatic procedure to restart crashed city must be present. The city moves the cars and updates the semaphores (Table 5.3). The average speed of all the car inside the system is used to control the performance of the system, so one of the objectives is maximize it.

Objectives	Actions
Be always active	Send update to semaphores
Be unique	Send movements to cars
Manage cars' movements	Control its uniqueness
Manage semaphores' updates	Receive informations from cars and semaphores

Table 5.3: Objectives and actions of City agent

5.1.1 Behaviours

In the following section we present how agents of different classes behave, during the simulation.

The behaviour of the city is quite simple at high level (Figure 5.1) and is composed by two sub-behaviours running in parallel.

Traffic simulator: it manages the traffic in the city grid. At fixed time interval it starts a new turn. In each turn the cars move in the grid trying to reach their arrival position. A turn ends when none of the cars can move. After that the semaphores receive an evaluation on their actions.

Messager: it replies to incoming messages and is responsible of the deletion of the agents if a terminate message arrives. New cars are added to the city when a proper message arrives and this behaviour is executed.

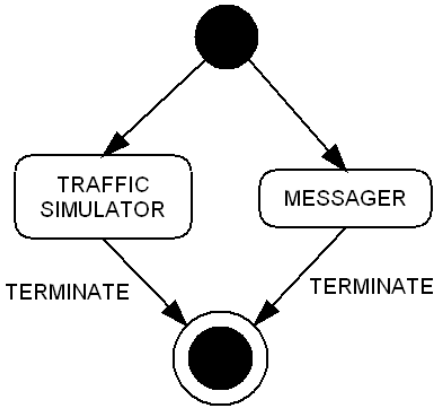

Figure 5.1: The behaviour of a city

The behaviour of a semaphore can be well described with a finite state automata (Figure 5.2).

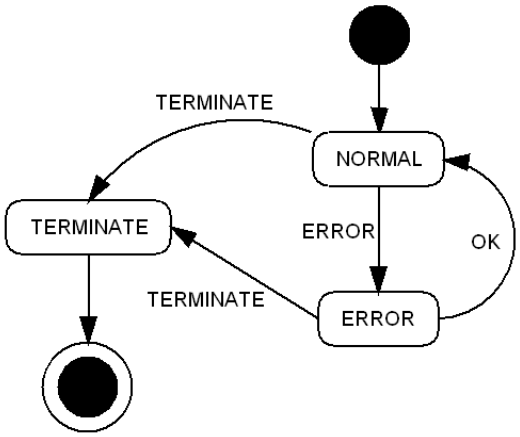

Figure 5.2: The behaviour of a semaphore

Normal: it characterizes the regular execution of the agent. It basically waits for evaluations sendt by the city. From time to time a function for auto-evaluation is called. The agent uses the evaluations collected to better manage the traffic.

Error: it is called when an error occurs during the execution and here is managed. Some actions that can be done for the correction are: ask for a new city grid and create a new agent city.

Terminate: it is called when a message of termination arrives. The doDelete() method is called and the agent terminates.

The behaviour of a car is the most complex of the three and is composed by five sub-behaviours. Like for semaphore, it can be represented with a finite state automata (Figure 5.3).

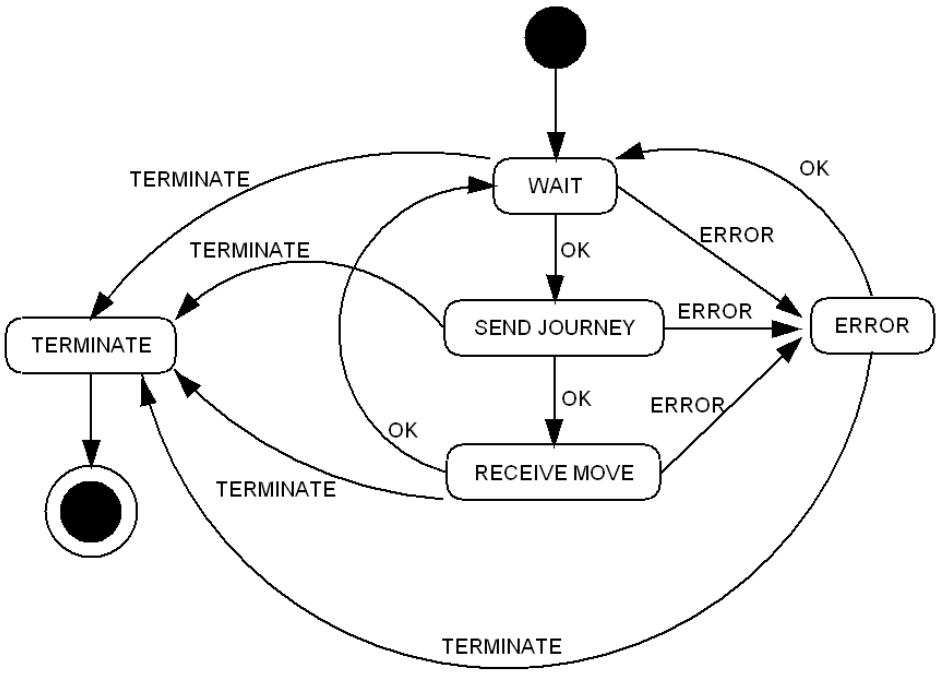

Figure 5.3: The behaviour of a car

Wait: characterizes the car when is parked, it does not move and does not receive any move messages from the city. This sub-behaviour is called when the agent is created and after the completion of the journey. After the behaviour for error management, wait is chosen in many cases.

Send journey: is the act of sending a new journey to the city. If it ends with ok the city accepts the journey, while ends with an error if the time for reply expires or the city rejects the journey.

Receive move: in this, the car waits messages of movement from the city. It terminates with OK when the car reaches the arrival position, with ERROR if some error occurs and with TERMINATE if a message of termination arrives.

Error: it is the behaviour for errors management. Here errors are analyzed and handled.

Terminate: like for semaphores, characterizes the agents when a message of termiation arrives and the agent has to be deleted.

5.1.2 Protocols of interaction

The protocols of interaction show which messages are sended during the execution.

The following sequence diagrams represent such protocols.

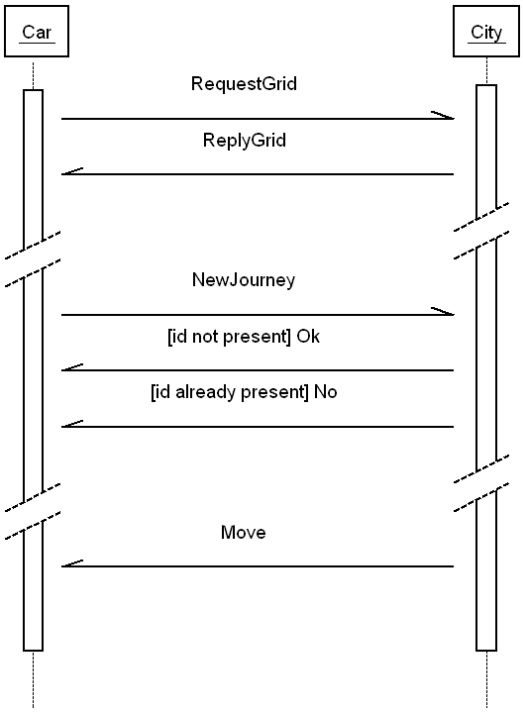

Figure 5.4: Interactions between car and city

The first operation to perform after the creation is obtainig the city grid. Both semaphores and cars need it and utilize the same interactions. The grid sendt is represented by a matrix and two integers representing the number of columns and the number of rows.

After the completion of any turn the city sends a move message to all car present in the city grid. The car does not have to reply. We have chosen this simple protocol to reduce the number of messages exchanged between agents. This kind of interaction is in fact the most common. The move message received represents the position reached in the city grid.

When a car wants to travel from a departure to an arrival position, it has to send a new journey message to the city. It returns an ok message if the car is not already present in the city grid, or a *no* message otherwise. The *no* message comes with a numerical value that explain the reason for the rejection (Figure 5.4).

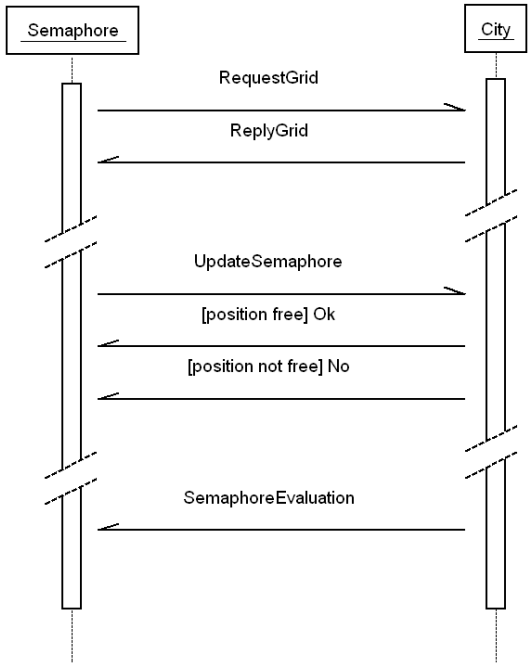

Figure 5.5: Interactions between car and city

Like for cars, the first operation to perform is requesting the city grid. The protocol is the same.

To modify its behaviour in the traffic management, a semaphore has to send an update message and then wait for the response. The response can be negative if the new position to occupy is already occupied by another semaphore, and is positive otherwise.

An evaluation for a semaphore is sended at the end of each turn. With this message the city comunicates how many cars were stopped and passed in both directions Similarly for move message, we decided not to reply to this message for speeding up the simulation (Figure 5.5).

5.1.3 Class diagrams

In this section we show the internal structure of the classes through some class diagrams. All the agents are impemented using JADE. In JADE each agent uses behaviours to perform its duties. They represent the logical

threads of a software agent implementation.

Car

The class Car (Figure 5.6) extends jade.core.Agent, the basic class of a JADE agent and it is abstact. The following methods must be implemented

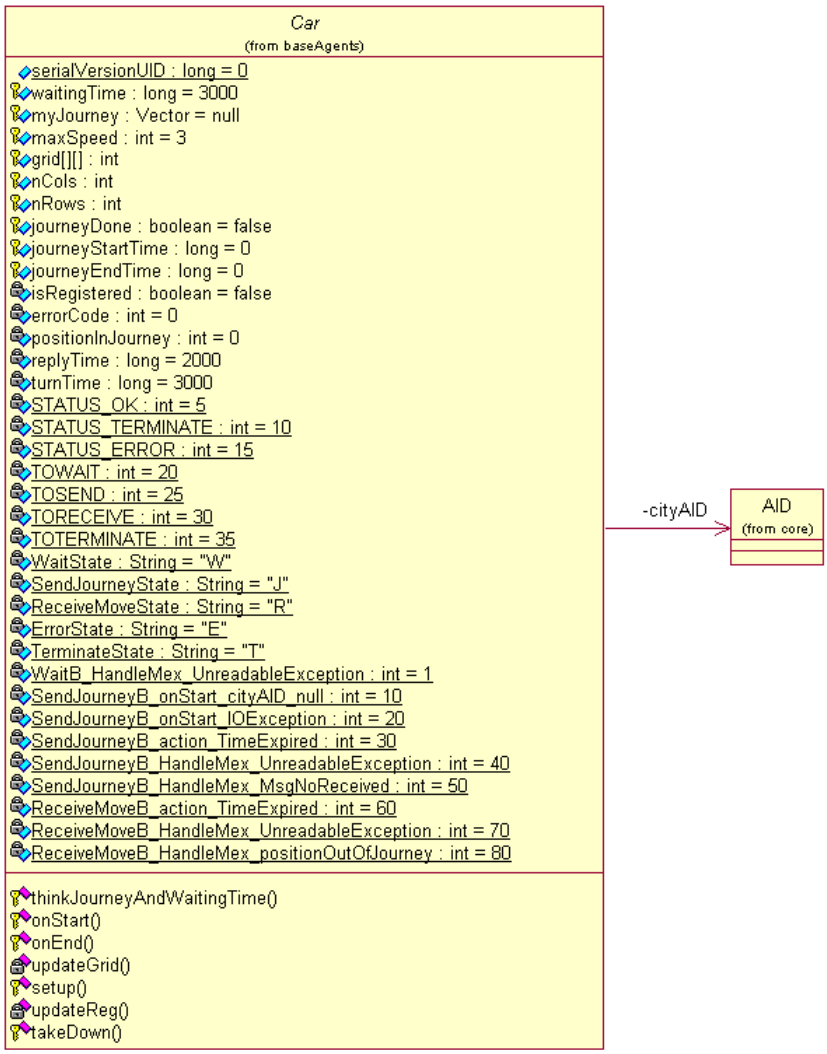

Figure 5.6: Class diagram of Car

in the subclasses:
thinkJourneyAndWaitingTime() is called only in the SendJourney Behaviour, it allows to set the new journey to do and the time to wait after the completion.

onStart() is called only when the agent starts in the setup method, it allows to set the initial values.
onEnd() is called only in Terminate Behaviour, it allows to do some operation before finishing.
The following methods are implemented:
updateGrid(): updates the representation of the city in the agent, sends a RequestGridMsg to the city.
setup(): called for the initialization of the agent. Registers the agent to the DF and adds the behaviours.
updateReg(): updates the registration to the DF.
takeDown(): deregister the agent from the DF.
The class has the following attributes:
serialVersionUID: used to implement correctly the class Serializable.
waitingTime: the time in milliseconds to wait after the completion of the journey.
myJourney: a list of adjacent cells.
maxSpeed: the maximal speed of the car, in cells/turn.
grid, nCols, nRows: the representation of the city grid.
journeyDone: true when the journey comes to the end, false otherwise.
journeyStartTime, journeyEndTime: used to calculate the duration of the journey.
isRegistered: keeps track if the agent is registered to the DF or not.
cityAID: the agens identifier of the city.
errorCode: keeps track of the last error occurred.
positionInJourney: the curent position of the car.
replyTime: milliseconds of a reply, maximum.
turnTime: mlliseconds of a turn, maximum.
STATUS_*, TO*, *State: used to control the behaviour of the agent.

Semaphore

The class Semaphore (Figure 5.7) extends jade.core.Agent and it is abstact.

The class has the following abstract methods:
evaluateAndThink() here we evaluate the managemend done by the sempahore. If the actor decide to change some parameters, the city has to be informed.
onStart() is called only when the agent starts in the setup method, it allows to set the initial values.
onEnd() is called only in Terminate Behaviour, it allows to do some operation before to end.
The class implements the following methods:
sendUpdateToCity(): sends an updateSemaphoreMsg to the city.
updateGrid(): updates the representation of the city in the agent, sends

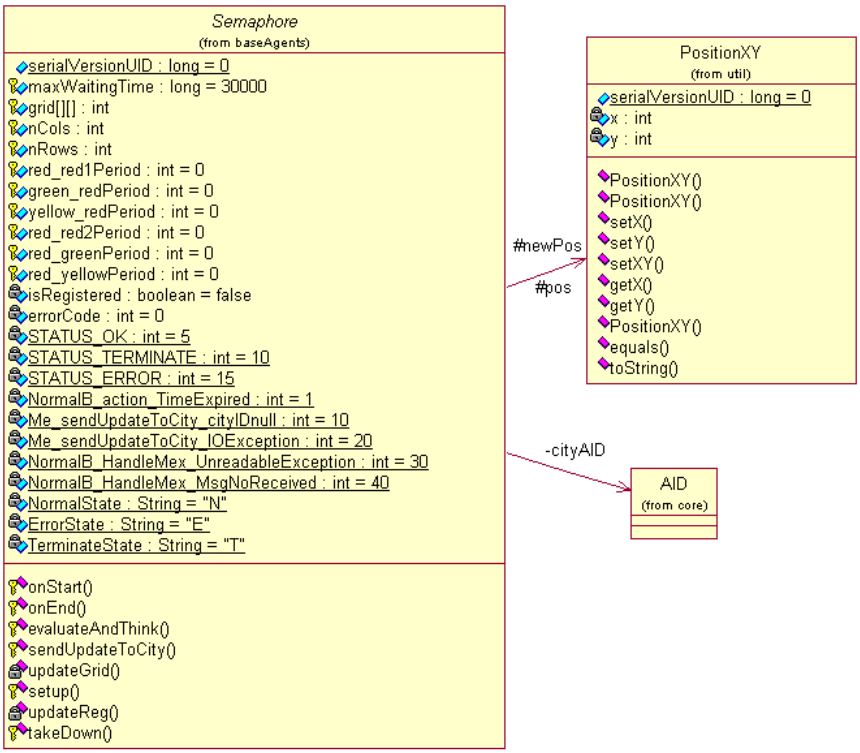

Figure 5.7: class diagram of Semaphore

a RequestGridMsg to the city.
setup(): called for the initialization of the agent. Registers it to the DF and add the behaviours.
updateReg(): updates the registration to the DF.
takeDown(): deregisters the agent from the DF.
The class has the following attribures:
serialVersionUID: used to implement correctly the class Serializable.
maxWaitingTime: maximum time passed waiting an update from the city.
grid, nCols, nRows: the representation of the city grid.
pos, newPos: point to the curent position of the semaphore and the new position to occupy, if it is free.
***Period:** number of turns for each combination of lights.
isRegistered: keeps track if the agent is registered to the DF or not.
cityAID: the agent identifier of the city.
errorCode: keeps track of the last error occurred.
STATUS_*, *State: used to control the behaviour of the agent.

City

The class City (Figure 5.8) extends jade.core.Agent and has the following methods:
setup(): called for the initialization of the agent. Registers it to the DF and adds the behaviours.
updateReg(): updates the registration to the DF.
takeDown(): deregisters the agent from the DF.
The class has the following attribures:
serialVersionUID: used to implement correctly the class Serializable.
isRegistered: keeps track if the agent is registered to the DF or not.
tc: this instance of the class trafficController manages every movement inside the city grid.
isFinished: controls the end of the agent, is setted in the behaviour for message management.

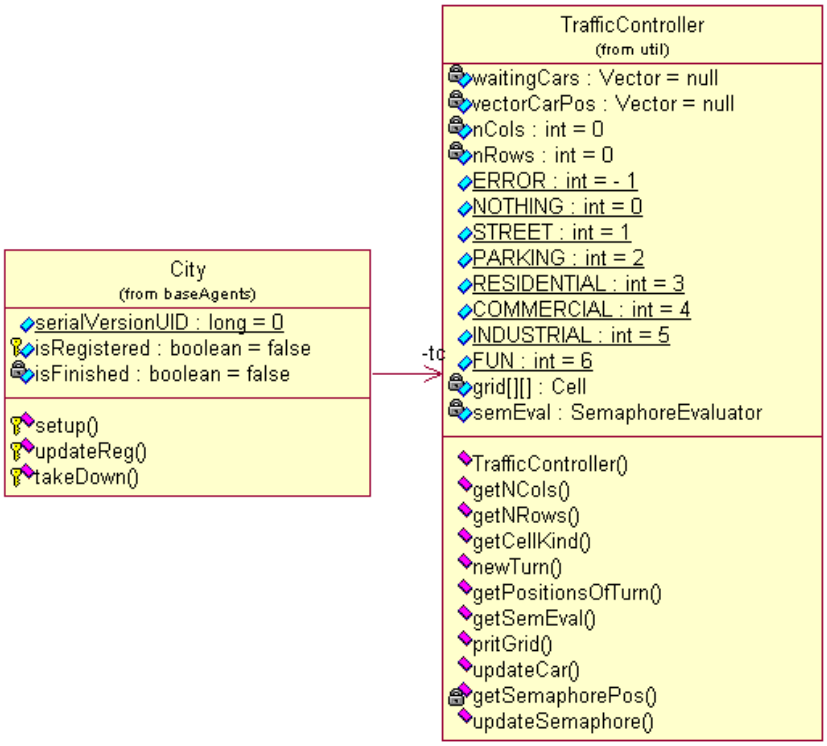

Figure 5.8: class diagram of City

Messages

The class MoveMsg represents a move and is sendt only by cities to cars (Figure 5.9). It has two attributes:

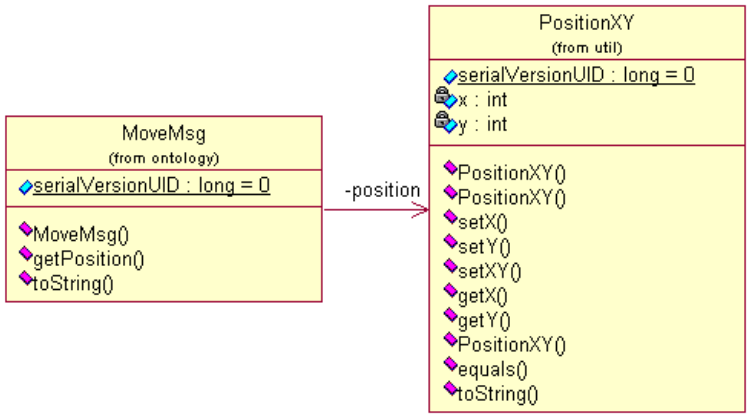

Figure 5.9: class diagram of MoveMsg

serialVersionUID: it is used to implement correctly Serializable.
position: is the position to communicate.

The class NewJourneyMsg is used to exchange information about journeys between cars and the city (Figure 5.10). It has three attributes:

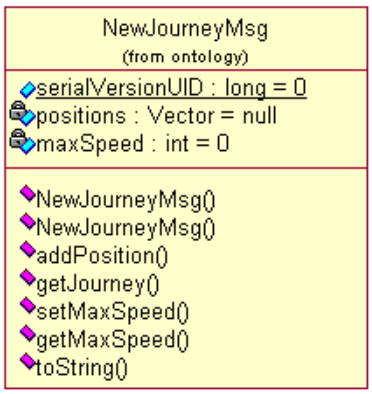

Figure 5.10: class diagram of NewJourneyMsg

serialVersionUID: it is used to implement correctly Serializable.
positions: the journey to be done.
maxSpeed: the max speed of the car, in cells/turn.

The class NoMsg is used to communicate a rejection (Figure 5.11). It has two attributes:
serialVersionUID: it is used to implement correctly Serializable.
errorCode: the reason of the rejection.

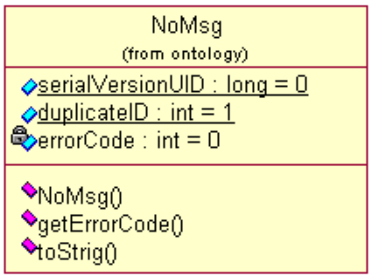

Figure 5.11: class diagram of NoMsg

The class OkMsg is used to comunicate an accept (Figure 5.12). This

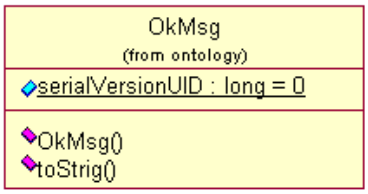

Figure 5.12: class diagram of OkMsg

class has only one attibute: **serialVersionUID:** it is used to implement correctly Serializable.

With a RequestGridMsg cars and semaphores can ask the city for the grid (Figure 5.13). This class has only one attibute: **serialVersionUID:** it

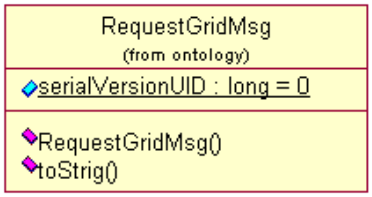

Figure 5.13: class diagram of RequestGridMsg

is used to implement correctly Serializable.

The class ReplyGridMsg is used to communicate the city grid (Figure 5.14). It has four attributes:
serialVersionUID: used to implement correctly Serializable.
grid: a matrix of integer values, it represents the city grid.
nCols: the number of columns of the matrix.
nRows: the number of rows of the matrix.

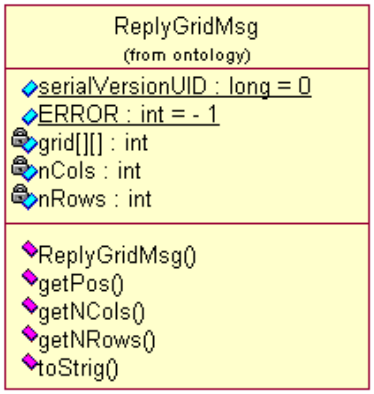

Figure 5.14: class diagram of ReplyGridMsg

The UpdateSemaphoreMsg is sendt by semaphores to the city. It includes new position to occupy and new periods for lights (Figure 5.15). It has eight attributes: **serialVersionUID:** it is used to implement correctly Serializable.
***Periods:** the new durations of lights
pos: new position to occupy, if it is free.

The SemphoreEvaluationMsg is sendt by the city to semaphores to let them know of how many cars are stopped or passed in the last turn (Figure 5.16). It has five attributes:
serialVersionUID: it is used to implement correctly Serializable.
topDownStopped: the number of cars stopped along y-axis.
topDownPassed: the number of cars passed along y-axis.
leftRightStopped: the number of cars stopped along x-axis.
leftRightStopped: the number of cars passed along x-axis.

Receiving the TerminationMsg causes the termination of the agent. It has only one attribute (Figure 5.17).

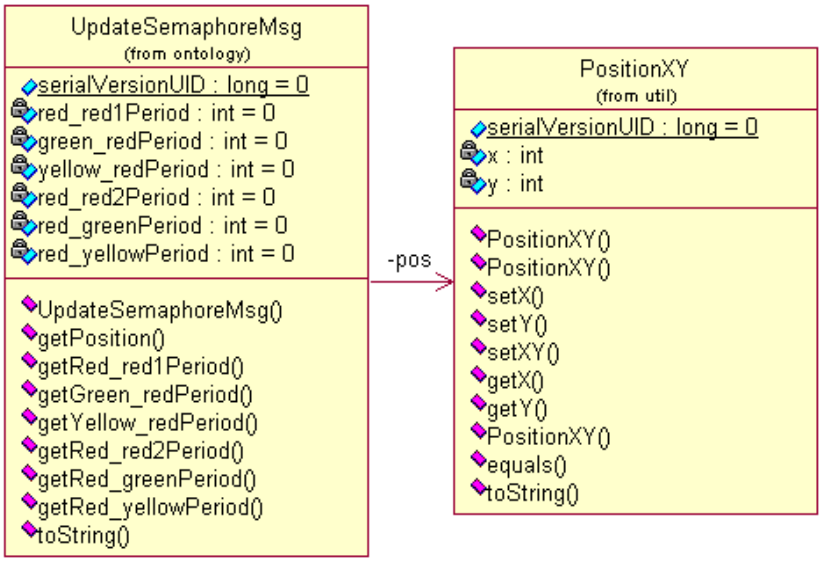

Figure 5.15: class diagram of UpdateSemaphoreMsg

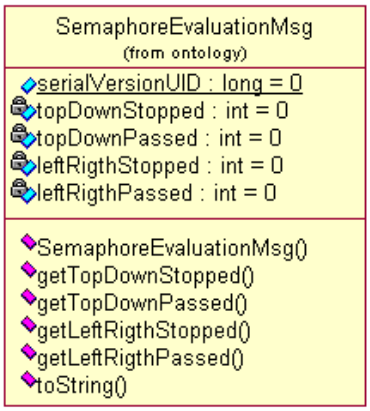

Figure 5.16: class diagram of SemaphoreEvaluationMsg

5.2 The creation of the implicit culture phenomenon

The observations are stored in the in the database of observations for a limited time. This automatic procedure allows the environment to be more dynamic and prevents agents from worrying about information cleaning.

We have identified two kind of high-level actions performed by cars.

WantMove: the car is searching a path to connect departure to arrival position without colliding with others vehicles. The path has to be

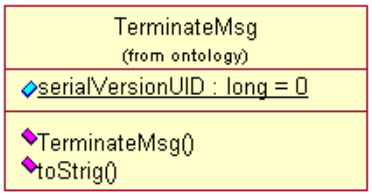

Figure 5.17: class diagram of TerminateMsg

as fast as possible. An observation containing departure and arrival positions and the desire to move is then sendt to the SICS.

Journey: the car has already done a journey from the deparure positon to the arrival position. An observation containing departure, arrival position, the path followed and the duration in time is sendt to the SICS.

The cultural theory in our application has the following form:

$$WantMove(actor, departurePos, arrivalPos)$$

$$\Downarrow$$

$$Journey(actor, departurePos, arrivalPos, path, time)$$

with the intuitive meaning of: *"If a car wants to move from a departure position to an arrival position, then it travel from the same departure to the arrival position following a certain path and spending an amount of time"*.

The decision process is defined by the following pseudocode:

```
1.   if (journeyDone equal true) then
1.2.   sendJourneyObsToSics(departure,arrival,journey,duration);
2.   end if;
3.   waitingTime = random between 1000 and 5000;
4.   journeyDone=false;
5.   decide departure position;
6.   decide arrival position;
7.   sendWantMoveObsToSics(departure,arrival);
8.   askSicsForSuggestions;
9.   evaulate the suggestions, return a number in (0,1);
10.  compare the evaluation with a random value in (0,1);
11.  if(rndValue < evaluation) then
11.1.  follow the suggestion;
12.  else
12.1.  make a self-decision;
13.  end if;
```

The evaluation of a suggestion is performed in this way:

```
1.   calculate the length of the path;
2.   if (lenght <10) then
2.1    return 0.95 - (length)*0.05/10.0;
2.2  end if;
3.   if (lenght <20) then
3.1    return 0.90F - (length-10)*0.10F/10.0F;
3.2  end if;
4.   if (lenght <40) then
4.1    return 0.80F - (length-20)*0.20F/20.0F;
4.2  end if;
5.   if (lenght <80) then
5.1    return 0.60F - (length-40)*0.30F/20.0F;
5.2  end if;
6.   return a random value between 0.00 and 0.30.
```

The self-decision is performed in this way:

```
1.   Create an empty journey;
2.   lastPos = null;
3.   currentPos = departurePosition;
4.   lastPos = null;
5.   journey.add(currentPos);
6.   do {
6.1.   nextPos = selectNextPosition(lastPos,currentPos);
6.2.   lastPos = currentPos;
6.3.   currentPos = nextPos;
6.4.   journey.add(currentPos);
6.5. } while(!currentPos.equals(arrivalPos));
7.   remove repetitions from journey;
```

In the implicit culture framework it is possible to distinguish between skilled actors and beginners. Since in our system the culture is shared among all actors, each of them has the same level of knowledge. A consequence is that the group G is equal to the group G'.

5.3 Experimental results

In this section we show that our software acts as a self-organizing system. In agreement with the definition given in Section 2.1 we have to prove three statements:

1. it is adaptive, according to the Zadeh's model for adaptivity,

2. its entropy is not maximal and changes in time,

3. it does not have a single point of failure.

For simulations, and for performance evaluation our system is composed of the following agents:

- one agent of type city,
- six agents af type car,
- one agent of type semaphore.

Six streets two cells wide are present in the city, taking the shape of a square 27×29 with a cross inside.

The semaphore is positioned in the middle of the cross (Figure 5.18). The simulation was carried out on a 1.6 GHz Centrino PC with 512 Mb ram.

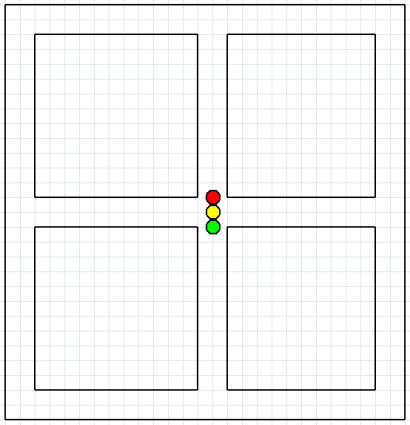

Figure 5.18: The city grid

5.3.1 On adaptivity

Our objective is to recognize if our system is adaptive or not. We observed the system for 100 minutes. We obtained the following results (Table 5.4 and Figure 5.19).

We observed that the first result is bad, three times higher than the last value. Between 20 and 40 minutes the number of collisions increases, while it decreases between 40 and 70. After 70 minutes it remains more or less stable.

We consider this result a clear signal of the adaptability of the system, because the cars have learnt how to move avoiding casualties with other

Time (from - to)	♯ of collisions for each car
0 - 10 min	2.0
10 - 20 min	$1.\overline{3}$
20 - 30 min	$2.\overline{6}$
30 - 40 min	$1.\overline{6}$
40 - 50 min	2.0
50 - 60 min	1.0
60 - 70 min	$0.\overline{3}$
70 - 80 min	$0.\overline{6}$
80 - 90 min	$0.\overline{3}$
90 - 100 min	$0.\overline{6}$

Table 5.4: Results for adaptivity test

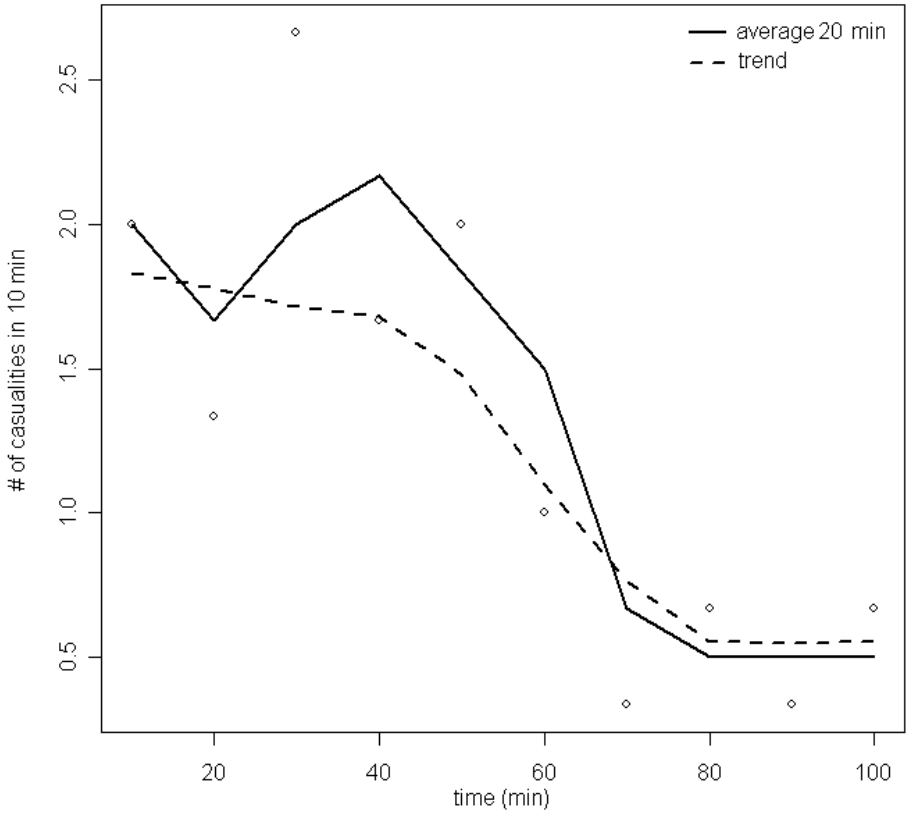

Figure 5.19: Results for adaptivity test

vehicles. The Zadeh's model for adaptability require a performance function and a criterion of acceptability. If we accept 0.6 collision per minute then

we can say that our system adapts in 80 minutes. The achievement of the steady state allows us to conclude positively.

5.3.2 On entropy

Our goal is to judge if the movement of cars is casual or if some restriction is applied on the path. The system under observation is the same as before: six cars, a semaphore, and the same city grid. At fixed time step we stopped the simulation and calculated the average evaluation of all the observation stored. This function for evaluation, in fact, represents the probability for a suggestion to be followed and measures how journeys are similar one to each other.

We found the following results (Table 5.5 and figure 5.20):

Time	Average evaluation
10	0.1224377
20	0.3473833
30	0.5432323
40	0.5394043
50	0.6794746
60	0.7244323
70	0.8774334
80	0.8424859
90	0.8334779
100	0.8498037

Table 5.5: Results for entropy test

With the progress of the simulation, the cars tend to follow the suggestion obtained to connect similar positions because SICS produces better suggestions and the evaluation increases. An effect is that the path is also similar and is clearly a signal that the entropy is not maximal. This phenomenon occurs with a high probability when the actors are skilled and is uncommon otherwise. Since we have implemented an automatic method to clean up out of date informations, an observations remain in the database for a limited period. This allows increase their quality and composition. As required for a SOS, the entrophy changes in time.

5.3.3 On failure's point

Our goal is to show that if an element crashes, the system is still able to perform the same tasks as before.

If the city crashes, all the messages sendt to the city do not receive a response. If no response is received for a prefixed time interval, then cars

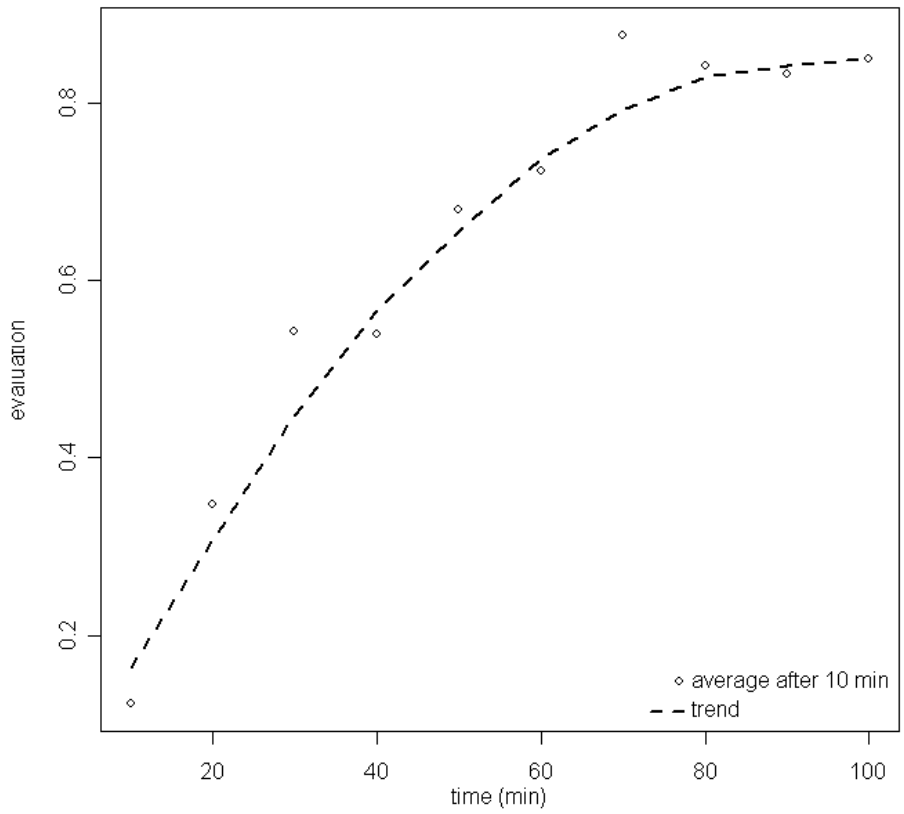

Figure 5.20: Results for entropy test

and semaphores ask the directory facilitator for the agent identifier of the city. If this identifier not valid then the receiver invoke the creation of a new agent city. If more than a city is created in the same time, the older remains active, the other terminates. After the creation the resulting system is equivalent to the previous. In this case the statement is vaild.

If an agent of type semaphore crashes, it is not automatically replaced by another equivalent agent. In the city, the semaphore keeps acting for a while and then is deleted. Since semaphores are not critical agents, the traffic management is still possible. So the statement is valid.

Like for semaphores, if an agent of type car crashes, is not automatically replaced. Cars are not critical agents so the statement is valid in this case too.

We have proved the truth of the statement in all cases, so our system does not have a single point of failure.

5.4 Performance analisys

5.4.1 System throughput

We decided to evaluate the throughput of the system, considering the average number of destinations reached by a car in a fixed time slot. We took the same target system and we obtained the following results (Table 5.6 and figure 5.21):

Time (from - to)	♯ of destinations reached
0 - 10 min	4.0
10 - 20 min	$17.\overline{3}$
20 - 30 min	$26.\overline{3}$
30 - 40 min	24.0
40 - 50 min	$23.\overline{3}$
50 - 60 min	21.0
60 - 70 min	$20.\overline{3}$
70 - 80 min	$26.\overline{3}$
80 - 90 min	$22.\overline{3}$
90 - 100 min	19.0

Table 5.6: Results for system throughput

We noted that the starting performance is low, approximately one sixth of the maximal value reached. Moreover the throughput increases fast in the first 30 minutes of simulation and then remains more or less stable. We consider this a kind of progress following a logistic law. Our best choice for the logistic fit is summarized in table 5.7.

Parameter	Value
A	0.1393311088345
B	0.0059338458317
α	3.3437667954774
$Error$	99.1620810790081

Table 5.7: Parameters for logistic fitting - throughput

We can conclude with a throughput at steady state ~ 2.348 destinations per minute, and with an improvement rate ~ 0.14.

5.4.2 Resources' utilization

Now we want to analyze what resources are utilized more during the simulation. Since our system was the only active application, we decided to use the

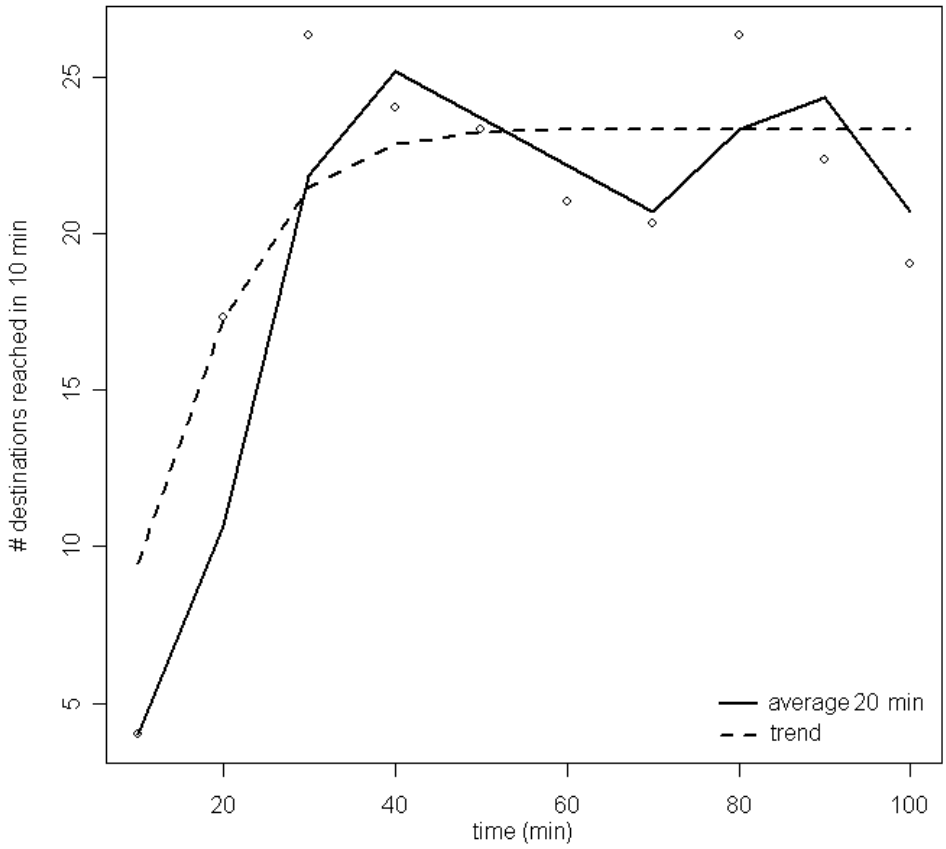

Figure 5.21: System Throughput

analysis tool included in the operative system. We made two observations on the fly:

- when an actor asks the SICS for suggestions, the utilization of the cpu is maximal for about one second;
- during all the simulation the hard disk is accessed continuosly, to store performance data and observations.

We then asked ourself why our laptop support only six cars running in parallel. We found two reasons:

- the java virtual machine needs a huge amount of memory to work and we need eight of them since each agent is created in a new container;
- the agent are coded using the jade framework: this platform wastes resources again, in fact it maps each container to a java virtual machine and each agent to a thread.

5.4.3 Task completion

We had a look on how many cars travel from the departure position to the right arrival position in less than one minute. In our target system we measured the following (Figure 5.22).

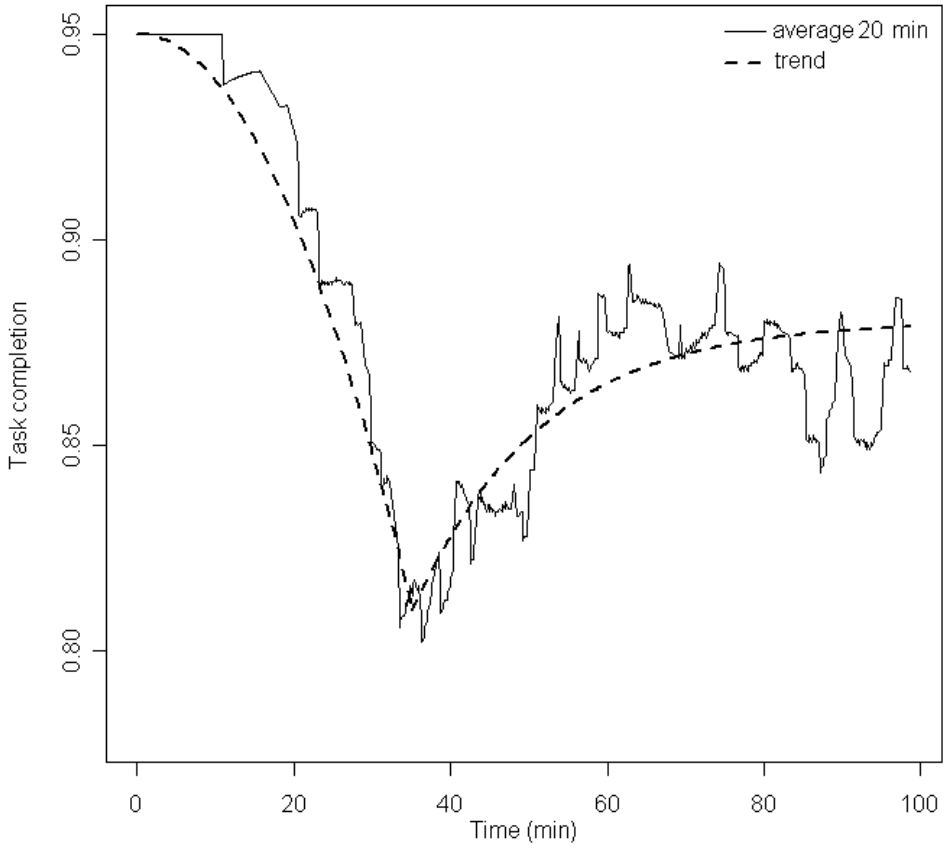

Figure 5.22: Task completion

We saw two trends. For the firsts 35 minutes the performance is decreasing. In particular we used a second order polynomial to smooth the real values. Our best choice is $y = 0.95 - \frac{1}{8859}x^2$, starts from 0.95 and ends to 0.81. After 35 minutes of running we attend to a different trend: the system quickly incrases its performance following a logistic law. We found the parameters in table 5.8:

We can conclude with a task completion at steady state ~ 0.88, and with an improvement rate ~ 0.065.

Parameter	Value
A	0.0650756238941895
B	0.0739495726070335
α	0.8111145903829003
$Error$	0.0693362493098065

Table 5.8: Parameters for logistic fitting - task completion

5.4.4 Availability

The system is said to be available if the arriving car starts moving in 700ms at maximum. We found the following results (Figure 5.23).

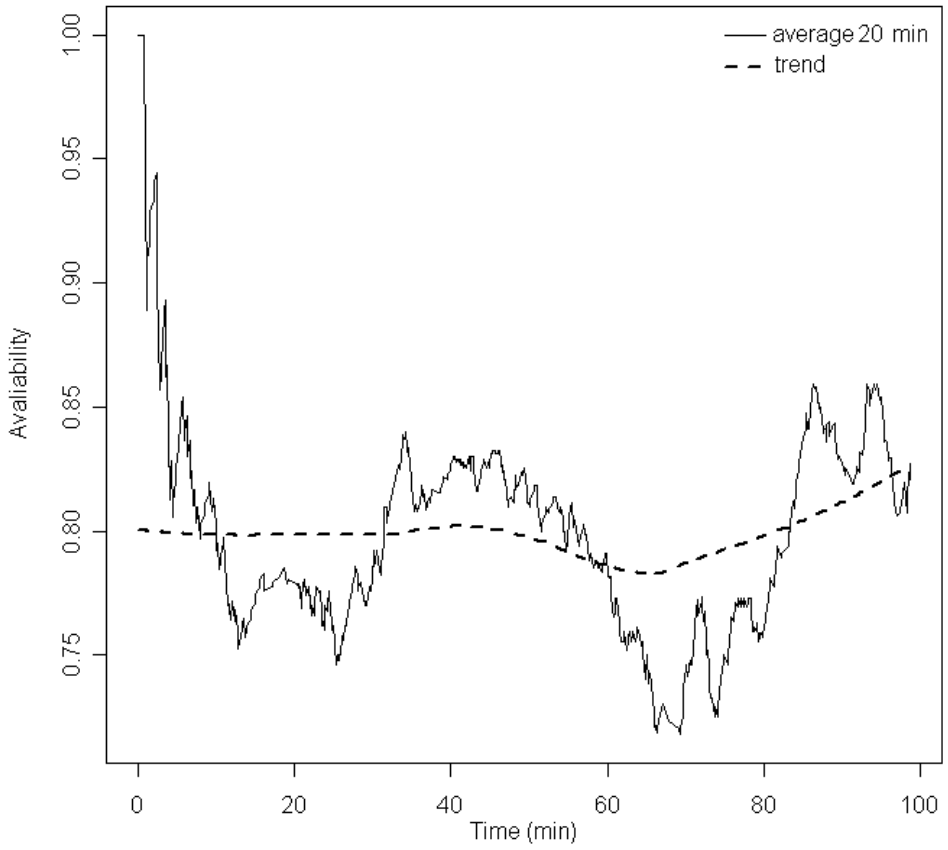

Figure 5.23: Availability

We realized that the availabilty decrease fast for 17 minutes. After, the average performance is stable at 0.81. From 65 minutes to the end, the availability seems to increase slowly. We consider this result not so

important due to the increased variance.

The availability at steady state of our system is ~ 0.81.

5.4.5 Stability in similar tasks

The results obtained for the entropy test are useful to evaluate the stability in similar tasks too, we simply consider them from another point of view.

If a suggestion has a low evaluation then it has a low probability to be followed. If another suggestion has a high evaluation its probability to be followed is high and the system is stable in this task.

Let us observe Table 5.5 and Figure 5.20 again. After 100 minutes of simulation an average observation is followed with probability ~ 0.85, and we consider stability reached in the general case.

5.4.6 Survivability

For survivability tests we use the following:

1. many collisions happen at the same time and all the city is open;
2. a specific part of the city is crowded while the other parts are empty, all the city is open;
3. many cars have to reach the same place and all the city is open;
4. for a while a part of the city is restricted.

Test 1

To do many casualties happen in the same time we simply kill the agent city. What happen is that cars and semaphore realize that the city is unreachable and some time after at least a new city starts. If more than one city is active in the system the first registered to the DF keep running while the others stop their execution.

So the system is able to cope with this kind of problems.

Test 2

To perform the second test we impose that two of five cars are limited in a specific part of the city. No difference were noted in the simulation and the others cars uses the crowded part as before.

The result is not the espected but our system is able to cope with this kind of problems.

Test 3

Here all the cars follow the same path, what happens is that the cars simply form a queue and no crash happens. This test point out that this is not a problem for the software, and the overall performance is increased.

Test 4

To close a part of the city for a while we change the city grid on the fly. Happens the following: the cars are not able to understand that a part of the city is closed and trying to reach the closed part, the cars enter in an endless loop.

So we conclude that our system crashes if a part of the city is restricted.

5.5 Organization detection and evaluation

In this section we compare the result obtained by the the analisys of task completion, system throughput and number of casualties. We note the following:

- after 35 minutes the task completion starts to increase following a logistic law (Section 5.4.3);
- after 40 minutes the system throughput reaches stability, through a logistic law (Section 5.4.1);
- after 30 minutes the number of casualties is at maximal value and start decreasing (Figure 5.19).

We recognize that the system starts acting in the desired behaviour from 35 minutes, so we say that the components of our system reaches a suitable organization approximately in that time. In section 5.3 we proved our system to be self-organizing so this oganization emerges automatically without human operations.

Moreover, by using performance meausures, we have an effective way to evaluate the order reached (Table 5.9 and table 5.10).

Parameter	Value	Measurement unit
Task completion at steady state	0.88	pure number
Availability at steady state	0.81	pure number
System performance at steady state	2.348	destinations per minute
Number of casualties at steady state	0.6	per minute
Stability in similar tasks	0.85	pure number (after 100 min)

Table 5.9: Evaluation of the organization

Problem	Coped
Many casualties in the same time	$\surd$
A part of the city crowded	partial
A part of the city unreachable	$\times$
Cars follow the same path	$\surd$

Table 5.10: Survivability of the organization

Chapter 6

Final considerations

In this thesis we presented a novel procedure to build and evaluate a general self-organizing system. It includes two phases.

- In the first many inputs are submitted to the system with the goal to increase the skills of the agents. Here we evaluate the task completion, the system throughput, the availability and the stability in similar tasks.
- In the second the system is tested under particular conditions that can decrease the performance of the system. Here the survivability is evaluated.

We applied some concepts coming from the performance evaluation to the field of self-organizing systems.

- For system throughput we focused on the number of destinations reached by a car in a fixed time slot.
- For task completion we fixed a maximum period to complete the tasks given and we verified periodically their completion.
- For the availability we said that the system is available if the task is started immediately.
- For the stability we tested if similar tasks are performed with similar actions.
- For the survivability we verified if our system is able to cope with specific tasks in controlled environment.

Later we introduced the implict culture framework and we explained why it can be used to implement a self-organizing system. A concrete system was then implemented with the support for the implicit culture: we presented the different actors and how they concur to the creation of the implicit culture phenomenon.

6.1 Results

The throughput of our system follows a logistic law with improvement rate ~ 0.14, reaching the steady state of 2.348 destinations per minute in ~ 40 minutes (Figure 6.1).

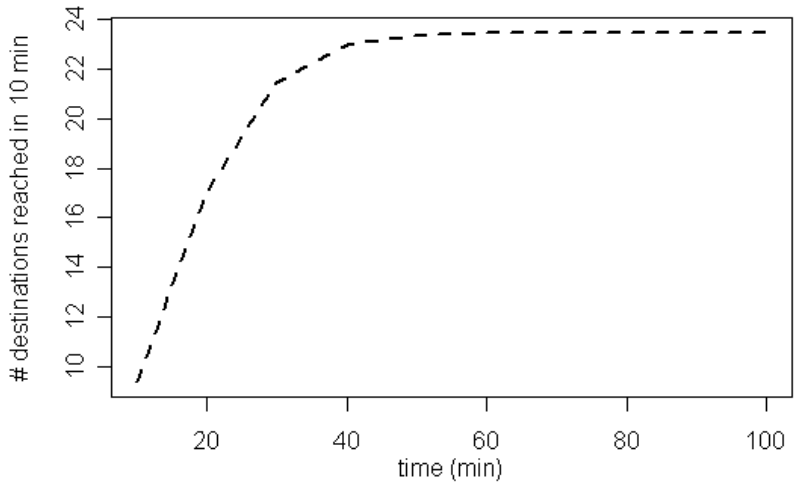

Figure 6.1: System throughput

At the beginning task completion decreases with a second order polynomial: $y = 0.95 - \frac{1}{8859}x^2$. After 35 minutes it starts increasing following a logistic law with improvement rate ~ 0.065, reaching the steady state of 0.88 (Figure 6.2).

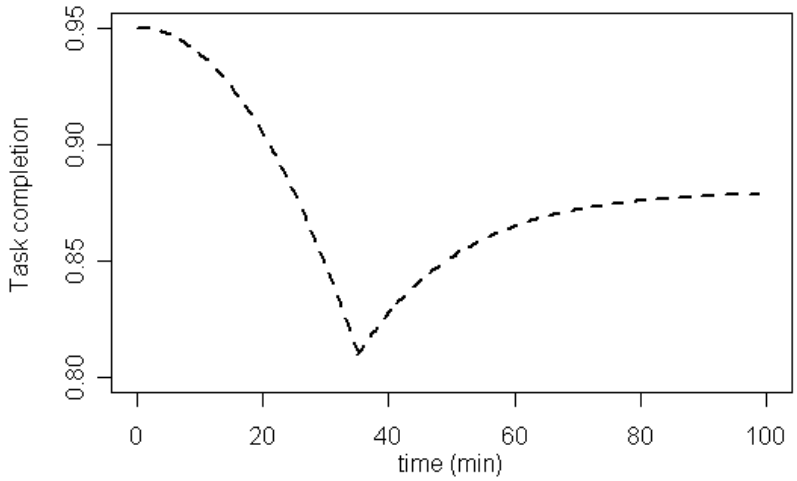

Figure 6.2: Task completion

The availability remains stable for all the simulation. It is the constant function $y = 0.81$ (Figure 6.3).

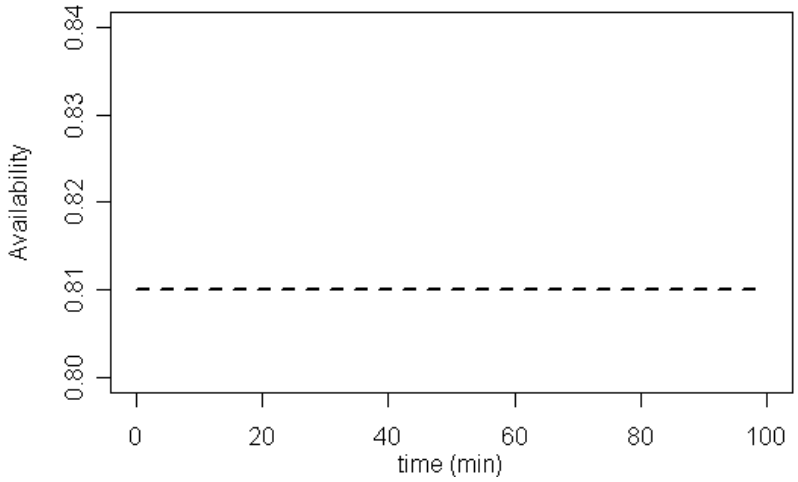

Figure 6.3: Availability

The stability in similar tasks increases from 0.12 to 0.85 and then remains more or less the same, reaching a steady state. For those reasons we can say that the stability at steady state is ~ 0.85.

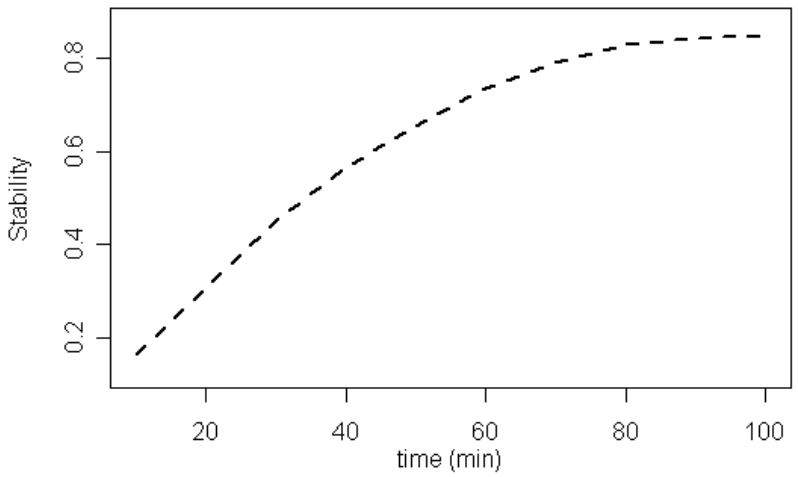

Figure 6.4: Stability in similar tasks

6.2 Future work

For the future we can develop two aspects: the first is to apply the method for evaluating self-organizing system to existing software to quantify their performance; the second is to improve the traffic simulator, to allow more cars to be active in the system at the same time.

We know that Hermann, Werner and Mühl, are currently building a classification of a certain number of existing systems in order to compile a catalog of SOSs. However their work does not focus on the performance of the systems, only on their adaptivity and their final structure.

For the improvement of our simulator we suggest not to use the jade platform any more since for this kind of systems too many hadware resources are required.

Considerazioni finali

In questa tesi abbiamo presentato una nuova procedura per costruire e valutare un qualsiasi sistema auto-organizzante. È composta da due fasi.

- Nella prima il sistema processa un notevole quantitativo di input con l'obiettivo di incrementare le capacitá degli attori. Qui vengono valutati task completion, throughput del sistema, disponibilitá e stabilitá nei task simili.
- Nella seconda il sistema viene testato in particolari condizioni che possono decrementarne le prestazioni. Qui viene valutata la survivability.

Abbiamo applicato alcuni concetti dell'analisi tradizionale delle performance al campo dei sistemi auto organizzanti.

- Per il throughput del sistema abbiamo considerato il numero di destinazioni raggiunte da un automobile in un tempo prefissato.
- Per il task completion abbiamo fissato un tempo massimo per compiere il lavoro richiesto ed abbiamo verificato periodicamente il loro completamento.
- Per la disponibilitá abbiamo considerato il sistema disponibile se il task appena immesso veniva svolto immediatamente.
- Per la stabilitá abbiamo testato se task simili erano svolti mediante azioni simili.
- Per la survivability avviamo verificato se il nostro sistema riesce a gestire specifici task in un ambiente controllato.

Successivamente abbiamo introdotto il famework Cultura Implicita ed abbiamo esposto le nostre motivazione per la sua scelta nell'implementeare sistemi auto organizzanti. Un sistema reale è stato poi implementato con il supporto alla cultura implicita: abbiamo presentato gli agenti e come concorrono alla creazione di fenomeni di cultura implicita.

Risultati

Il throughput del sistema segue il modello logistic, con improvement rate ~ 0.14 e raggiunge lo steady state di 2.348 destinazioni al minuto in 40 minuti (Figure 6.5).

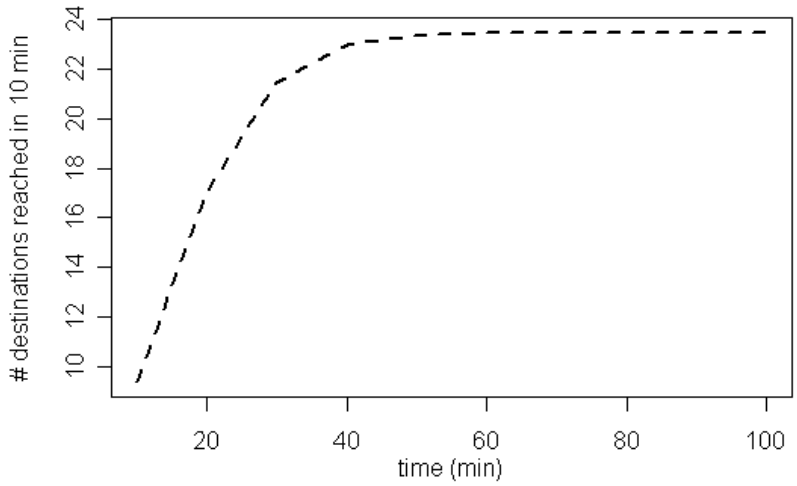

Figure 6.5: Throughput del sistema

All'inizio il task completion decresce seguendo un polinomio del secondo ordine: $y = 0.95 - \frac{1}{8859}x^2$. Successivamente comincia a crescere, seguendo un modello logistic con improvement rate ~ 0.065, e raggiungendo lo steady state di 0.88 (Figure 6.6).

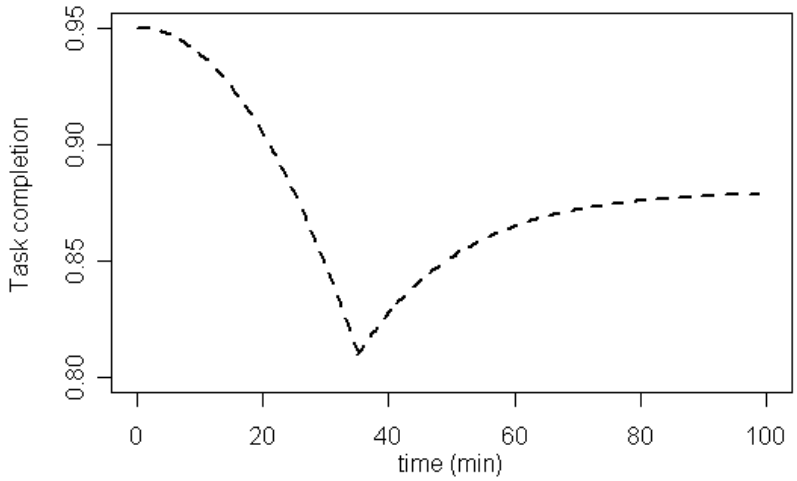

Figure 6.6: Task completion

La disponibilitá rimane stabile per tutta la durata della simulazione. È la funzione costante $y = 0.81$ (Figure 6.7).

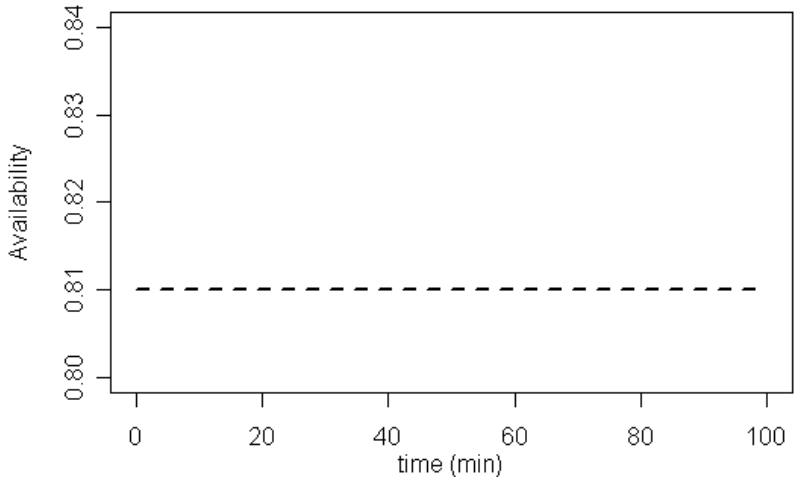

Figure 6.7: Disponibilitá

La stabilitá nei task simili cresce da 0.12 a 0.85, poi rimanne costante, raggiungendo uno stato di equilibrio. Per questi motivi possiamo dire che la stabilitá allo steady state è ~ 0.85.

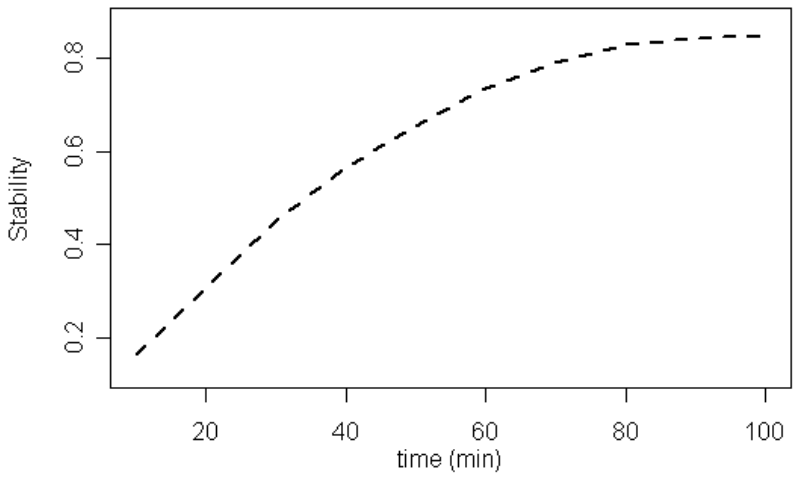

Figure 6.8: Stabilitá in task simili

Sviluppi futuri

Per il futuro è possibilie migliorare due aspetti: il primo consiste nell'applicare il metodo per la valutazione dei sistemi auto organizzanti al software esistente per quantificare le loro prestazioni; il secondo è migliorare il simulatore di traffico, per permettere a piú automobili di essere attive nello stesso istante.

Sappiamo che Hermann, Werner e Mühl, stanno classificando un certo numero di sistemi esistenti con l'obiettivo di compilare un catalogo di sistemi auto organizzanti. Tuttavia il loro lavoro non focalizza sulle performace dei sistemi, ma solo sulla loro capacitá di adattamento e sulla struttura finale raggiunta.

Per migliorare il simulatore suggeriamo di non utilizzare la piattaforma jade in quanto non garantisce prestazioni accettabili e necessita di troppe risorse hardware.

Bibliography

[1] E. Blanzieri, P.Giorgini, P. Massa, S. Recla, *Implicit Culture for multi-agent interaction support*, 2001.

[2] A. Gastaldello, P. Giorgini, E. Blanzieri, *Progettazione e sviluppo di un modulo induttivo basato su regole associative per il sistema di supporto alla cultura implicita*, Thesis, 2004-2005.

[3] W. S. Cleveland, *LOWESS: A program for smoothing scatterplots by robust locally weighted regression* 1981, The American Statistician, 35, 54.

[4] A. Birukou, E. Blanzieri, V. D'Andrea, P. Giorgini, N. Kokash, A. Modena, *IC-Service: A Service-Oriented Approach to the Development of Recommendation Systems*, 2007, Proceedings of ACM Symposium on Applied Computing. Special Track on Web Technologies, pp. 1683-1688.

[5] A. Birukou, E. Blanzieri, P. Giorgini, *Implicit: An Agent-Based Recommendation System for Web Search*, 2005, AAMAS '06: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, pp 618-624, ACM Press.

[6] A. Birukou, E. Blanzieri, P. Giorgini, *A Multi-Agent System that Facilitates Scientific Publications Search*,

[7] C. Gershenson, F. Heylighen, *When Can we Call a System Self-organizing?*, 2003.

[8] K. Herrmann, M. Werner, G. Mühl, *A Methodology for Classifying Self-Organizing Software Systems*, International Transactions on Systems Science and Applications, 2006, Vol. 2, Num. 1, pp. 41-50.

[9] L. Zadeh, *On the definition of adaptivity*, Proceedings of the IEEE, Vol. 51, 1963, p. 469.

[10] C. E. Shannon, *A mathematical theory of communication*, Bell System Technical Journal, Vol. 27, 1948, pp. 379-423 and 623-656.

[11] S. Brueckner, H. Czap, *Organization, Self-Organization, Autonomy and Emergence: Status and Challenges*, International Transactions on Systems Science and Applications, 2006, Vol. 2, Num. 1, pp. 1-9.

[12] C. Gershenson, *A General Methodology for Designing Self-Organizing Systems*, 2006, ACM Journal Name, Vol. V, No. N, M 20YY.

[13] H. Kasinger, B. Bauer, *Pollination - A Biologically Inspired Paradigm for Self-Managing Systems*, International Transactions on Systems Science and Applications, 2006, Vol. 2, Num. 2, pp. 147-156.

[14] F. Bimbard, L. George, *Real-Time Analysis to Ensure Deterministic Behavior in a Modular Robot Based on an OSEK System*, International Transactions on Systems Science and Applications, 2006, Vol. 2, Num. 2, pp. 185-190.

[15] N. R. Franks, T. Richardson, *Teaching in tandem-running ants*, 2006, Nature 439 (7073): 153.

[16] J. Eckmann, E. Moses, *Curvature of co-links uncovers hidden thematic layers in the World Wide Web*, 2002,

[17] Ma. Latapy, P. Pons, *Computing communities in large networks using random walks*, 2004.

[18] M. E. J. Newman, M. Girvan, *Finding and evaluating community structure in networks*, 2004.

[19] M. E. J. Newman, *Analysis of weighted networks*, 2004.

[20] R. Guimerá, M. Sales-Pardo, L. A. N. Amaral, *Modularity from Fluctuations in Random Graphs and Complex Networks*, 2006.

[21] J. Duch, A. Arenas, *Community detection in complex networks using Extremal Optimization*, 2005.

[22] M. Girvan, M. E. J. Newman, *Community structure in social and biological networks*, 2002.

[23] S. Fortunato, V. Latora, M. Marchiori, *A Method to Find Community Structures Based on Information Centrality*, 2004.

[24] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, D. Parisi, *Defining and identifying communities in networks*, 2004.

[25] L. Donetti, M. A. Muñoz, *Detecting Network Communities: a new systematic and efficient algorithm*, 2004.

[26] L. Donetti, M. A. Muñoz, *Improved spectral algorithm for the detection of network communities*, 2005.

[27] J. P. Bagrow, E. M. Boll, *A Local Method for Detecting Communities*, 2004.

[28] A. Capocci, V.D.P. Servedio, G. Caldarelli, F. Colaiori, *Detecting communities in large networks*, 2004.

[29] F. Wu, B. A. Huberman, *Finding communities in linear time: a physics approach*, 2004.

[30] G. Palla, I. Derènyi, I. Farkas, T. Vicsek, *Uncovering the overlapping community structure of complex networks in nature and society*, 2005.

[31] J. Reichardt, S. Bornholdt, *Detecting fuzzy community structures in complex networks with a Potts model*, 2004.

[32] R. Guimerá and L. A. N. Amaral, *Functional cartography of complex metabolic networks*, 2005.

[33] H. Zhou, R. Lipowsky, *Dynamic pattern evolution on scale-free networks*, 2004.

[34] G.Bolch, S.Greiner, H. de Meer, K. S. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*, Wiley, 1998.

[35] R. Pressman, *Principi di ingegneria del software*, Mc Graw Hill, 2000.

[36] G. Elliott, C. W. J. Granger, A. Timmermann, *Handbook of economic forecasting*, North-Holland, 2006.

[37] J. Han, M.Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann, 2005.

[38] L. H. Hartwell, J. J. Hopfield, S. Leibler, A. W. Murray *From molecular to modular cell biology*, Nature 402, 1999.

[39] R. V. Solé, P. Fernández, *Modulariy for free in genome architecture*, q-bio/0312032, 2003.