Programs Self-Healing over a termites simulator based on language games and evolutionary computing

Arles Rodríguez Portela[1,2], Jonatan Gómez Perdomo[1,2]

[1]Alife Research Group
[2]National University of Colombia
aerodriguezp@unal.edu.co,jgomezpe@unal.edu.co

Abstract

This paper presents a mechanism of programs self-healing in an environment of agents looking for food. The failure system is defined based on initial failures that each agent (termite) has on their programs. By using language games concepts and the Q-learning algorithm, termites diagnose failures on their programs. Termites also have enough information to determine if their programs are failing based on a simple voting system that is the result of language games of diagnosis. The proposed self-healing mechanism was tested on virtual worlds with 100 and 200 termites and a different failure per termite. The results show that the proposed approach is capable, from local interactions, of building a set of very specific diagnosis questions, allowing the system to diagnose more than one type of failure at the same time, while the accounted number of diagnosis questions for instructions with low failure probability is reduced. By using the voting system and storing a ranking of possible missing code lines, mutations are induced on the code and the system is capable of recovering the programs.

Introduction

Self-healing is based on the ability to detect software and hardware components that are failing. Systems must detect failures on components and then replace, eliminate or repair them without disrupting the system operation (Nami and Bertels, 2007).

Self-healing involves: the design and verification of an autonomic system which has some of the complexity of a real system in order to locate functions and services offered by an autonomic element in an efficient manner (Kephart and Chess, 2003), to make an abstraction of behaviors to obtain emergent properties and global behaviors from local actions (Kephart and Chess, 2003; Bicocchi and Zambonelli, 2007; De Wolf and Holvoet, 2003), to reallocate resources (Arora et al., 2006) and to locate faults as fast as possible (Gao et al., 2004).

An important part of the problem is to develop a virtual organization in an area where certain items may have certain types of failures and to reduce the risk of large losses by getting a reconfiguration that ensures the continuity of the system and the potential generation of learning about corrective actions (Nami and Sharifi, 2007; Gao et al., 2004).

Some works try to find the cause of failures on distributed transaction environments with good times of response (Gao et al., 2004). One of them is about failure detection on heterogeneous environments as a NP-hard problem. Its approach is based on a dependency matrix of transactions versus resources and consider only binary dependencies i.e. a 0/1 matrix. Another work deals with the concept of self-regeneration introduced as a survival mechanism of systems that reduces the role of human experts. This work is focused on security and shows as an application the project CSISM, which implements multi-layer reasoning with fast reaction rules designed to take effective defensive actions within 250 ms after the initiation of the attack (Atighetchi and Pal, 2009).

Self-healing has been proposed for operating systems and distributed network environments (Rott, 2007). Rott decomposed this process in four main components: Monitoring, Adaptation, Interpretation and Resolution. By adopting the behavior of human administrators, also defined an optimal self-healing process in a computer environment into three stages: prevention, first aid and immunization. Rott considered as an example the ability to restore a service from an XML policy, which was implemented in Solaris 10. Some research demonstrated that it is possible to build self-healing operating systems through simple and effective techniques such as code reloading, component isolation and automatic restarts (David and Campbell, 2007).

A code injection mechanism for Java to introduce self-healing in object-oriented applications also has been proposed (Fuad et al., 2006). The model includes sensors that capture the state of the variables before calling the functions and encapsulating the exceptions. When any runtime failure occurs, the failure is notified and the system tries to reconstruct the unsuccessful method, so that it could be restarted at the point where the failure occurred. Otherwise, the system notifies the system administrator and some actions are executed like log generation. Fuad remarked that the code must be analyzed and the autonomic functionality should be inserted in such a manner that it is separated from the service functionality of the legacy system. Also, a framework based on Java annotations was presented. This framework creates and builds applications with self-healing using a simple language of annotations (Breitgand et al., 2007).

Self-healing over networks was performed by injecting different types of faults to a network during training using cost-sensitive fault remediation (Littman et al., 2004). In cost-sensitive fault remediation, a decision maker is responsible for repairing a system when it breaks down. To narrow down the source of the fault, the decision maker can perform a test action, at some cost, and repair the fault if a repair action can be carried out.

A framework to specify, validate and generate autonomic systems, called Autonomic System Specification Language (ASSL) presented concrete results to specify a self-healing behavior model for NASA swarm-based exploration missions. The system send messages from a worker similar to heartbeats, or messages with a diagnosis (Vassev and Hinchey, 2009; Vassev and Paquet, 2007). A mechanism of self-healing for resource allocation using Ant Colony Optimization is also presented (Zhou et al., 2008). The obtained results are scalable to different kind of problems.

NASA has an initiative to carry out explorations in asteroid belts in a project called ANTS (Autonomic Nano Technology Swarm), based on autonomic computing (Truszkowski et al., 2004). The system has specialized workers to obtain information about asteroids, a central agent that gives a global goal and some messengers that send signals between the agents and the spatial station. NASA prototypes offer autonomic properties and these are defined in the architecture design that implements a wide level of autonomous and intelligent agents. These prototypes manage concepts, like specialization, in which an agent is designed to carry out a specific work and can redefine its task, it can also adapt itself to the environment and learn from its work or be easily replaced for other, if it has high-level failures. A concept mission is currently being planned to be launched between the years 2020 and 2030 as functional prototypes (Truszkowski et al., 2006).

In order to work self-healing from a generic perspective, taking some of the previous works as that of space exploration (ANTS) and the challenge of reproducing a system that captures part of the complexity of the real world, a virtual world in which termites that can carry out a task in a given environment is created. The termites simulator is an environment that includes the interaction among multiple elements, providing a more general solution instead of defining self-healing over operating systems or software, providing a motivation and a possible future extension not only for an application for self-healing in software but also for being extended to a hardware one.

Swarms have self-organization that makes them interesting. Considering that from self-organization it can be obtained self-administration like an emergent property (Bicocchi and Zambonelli, 2007), not only it is possible to obtain self-administration but also self-healing. In this paper self-healing is studied from a perspective of artificial life that is based on the emergency and self-organization ideas, with many elements that interact with others through local rules and a synthetic approach would be adopted in which behaviors are understood throughout the construction of the same ones, using computer simulations (Langton, 1989).

Agents are called ”termites” because they have social organization (only the worker termites are modeled). Feeding of termites is carried out for trophallaxis, it means that food is stored in their stomach and it is transfered among members of a community through mouth-to-mouth or anus-to-mouth feeding (Wikipedia, 2011). In this case also the pheromones define a communication mechanism.

In this paper, a simulator of a termite's swarm, a failure system, and a self-healing mechanism are introduced. Termites were modeled as agents with a virtual machine that execute instructions about motion and diagnosis. An Ant Colony System algorithm (ACS) was used to locate two points of food in the space. A failure was defined as a bad copy of a base program of a termite. Agents also diagnose others using language games and Q-learning.

Language games involves local interactions between two agents (a speaker and a hearer), in an environment with other agents, objects and situations. Some games allow the speaker to make the hearer perform an action (Steels and Vogt, 1997). The language game of diagnosis in this paper consists of one question of diagnosis about the programs of the termites; if the hearer termite does not have the code line that the speaker is expecting, the speaker rewards the diagnosis question. After recognizing the error, a voting parameter about failures is updated on each hearer termite, using a vector called belief vector of failures per each termite.

This paper is organized in the following way: first, the agents and the binary programs of the termites are described. The second part deals with the failure system, the diagnosis mechanism and the self-healing algorithm. Finally, the experiments with 100 and 200 termites are performed, with 10, 30, 50 and 70 percent of the sick termites at the beginning. Results are organized in terms of sick termites and termites that were healed.

The Termites System

Agents and ACS

Agents are termites that look for food, carry it, take it to the nest and continue searching for more food using Ant Colony System (Dorigo and Gambardella, 1997). The world is a toroidal space initialized with a pheromone value of zero for the termite nest and all termites start from this position with the simulation. Two points of food were defined with a pheromone value of one and the other world positions have a pheromone value of 0.5. Termites are represented as white squares if they are looking for food and blue squares if they are carrying food and have eight possible movements: none, down, left, right, up, upleft, upright, downright and downleft (Fig 1).

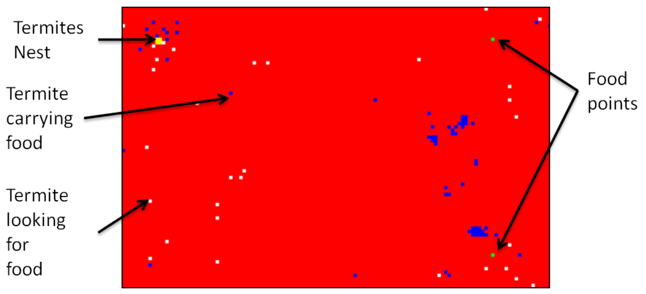


Figure 1: The Termites World

Termites start making random movements when looking for food and when they finally reach the food its color changes from white to blue and the pheromone production starts. Pheromone values in vicinity of termite and the search status (seeking, carrying) are the input of algorithm to select an action (Fig 2). If termite is looking for food, the first direction with the less amount of pheromone is chosen and if the termite is carrying food, the termite chooses the first direction with more pheromone. Then the termite moves in this direction, and the pheromone of termites and world pheromone are updated (Eq 1 and 2).

$$tph = (tph + 0.01 * (0.5 - tph)) \quad (1)$$

$$wph(x, y) = wph(x, y) + 0.01 * (tph - wph(x, y)) \quad (2)$$

Where:

- tph is the pheromone of the termite.
- $wph(x, y)$ is the pheromone of world in the new location of the termite (x,y).

If the termite reaches its nest, its pheromone value is updated to 0, if the termite reaches a food point the pheromone of the termite gets a value of 1.

The Termites Programs

Each agent has a simple program, which is executed line by line, that encapsulates the Ant Colony System algorithm and the mechanism of diagnosis based on language games. Each program is a vector of binary values that represent the sensations and actions to be performed by the termite. The base program of termites is exposed in Table . `sSeek` is a sensor that indicates if the termites are looking for food, `sCarry` indicates if the termites are carrying the food and `sNeigh` indicates if termites have only one neighbor. Neighbor and

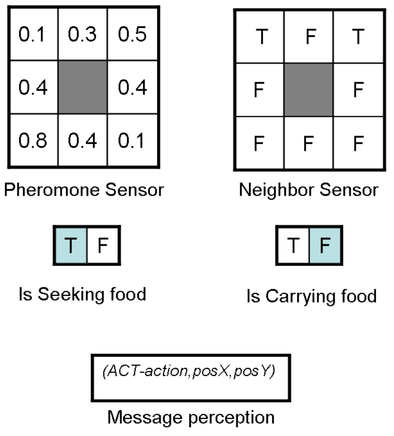


Figure 2: The Termites Sensors

pheromone sensors are defined in the Moore neighborhood $r = 1$ with center in the termite location (Fig 2). `acSeek` and `acCarry` are simple instructions that execute the Ant Colony System algorithm and `acDiag` starts a diagnosis. The first instruction of the base program (Table), is generated based on the rule: ”if the termite is looking for food and the termite does not have one neighbor, then the termite has to look for food”.

sSeek	sCarry	sNeigh	acSeek	acCarry	acDiag
1	0	0	1	0	0
0	1	1	0	0	1
0	1	1	0	1	0
0	1	0	0	1	0
1	0	1	0	0	1
1	0	1	1	0	0

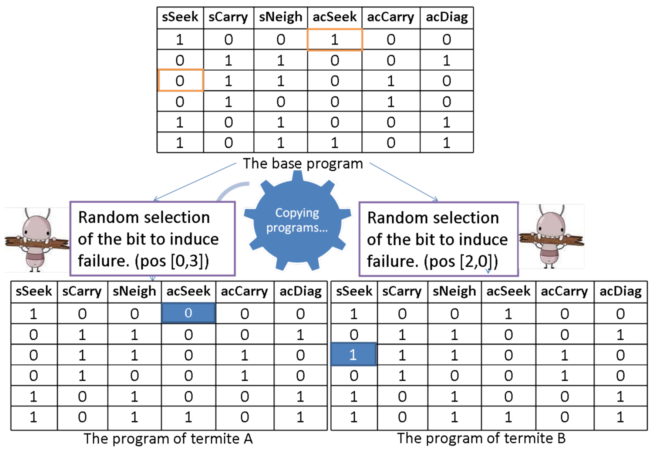
Table 1: Base program for a termite

Each termite has an interpreter for its program. The interpreter takes each line of code and compares it with the perception of each sensor. If the line of code matches the perceptions, then the action indicated in the code line is performed. If more than one action is specified, the interpreter returns the action with the greatest priority. Priority is defined in the following order: `acSeek` $>$ `acCarry` $>$ `acDiag`.

Failures definition

Program failures are simulated as bad coding from the beginning. Each termite has a variation of the program that the ”queen” has (the base program). The programs are copied with a failure probability, it means not all termites will have programs with failures. For example, a failure probability of 0.1 means that approximately the 10% of the population have a failure.

A failure is a change in a random bit of the code per termite, so each termite has a different failure and it makes that the termites act in unexpected ways (Fig 3).



The base program

sSeek	sCarry	sNeigh	acSeek	acCarry	acDiag
1	0	0	1	0	0
0	1	1	0	0	1
0	1	1	0	1	0
0	1	0	0	1	0
1	0	1	0	0	1
1	0	1	1	0	1

The program of termite A

sSeek	sCarry	sNeigh	acSeek	acCarry	acDiag
1	0	0	0	0	0
0	1	1	0	0	1
0	1	1	0	1	0
0	1	0	0	1	0
1	0	1	0	0	1
1	0	1	1	0	1

The program of termite B

sSeek	sCarry	sNeigh	acSeek	acCarry	acDiag
1	0	0	1	0	0
0	1	1	0	0	1
1	1	1	0	1	0
0	1	0	0	1	0
1	0	1	0	0	1
1	0	1	1	0	1

Random selection of the bit to induce failure. (pos [0,3])
Copying programs...
Random selection of the bit to induce failure. (pos [2,0])

Figure 3: Failure selection for two termites

Diagnosis Mechanism

Diagnosis based on Language Games

A Language game is a sequence of local interactions between two agents (a speaker and a hearer) located in a specific environment (Steels, 2001). Some language games allow agents to identify objects in the environment using linguistic means and others allow the speaker to obtain actions from the hearer (Steels and Vogt, 1997). Some of language games were taken to design the mechanism of diagnosis for the programs.

A termite can send messages to a world location, if there is a termite in this place and the termites are neighbors. A diagnosis is started if a termite receives a message in its current position. This termite has to remain at this location, to clean the message from the world location and to reply the message. The diagnosis is encapsulated in the `acDiag` instruction in the termites program and was modified following the process below (Diagnose instructions are defined on Table 2):

- Making contact: Two agents are physically close (they are neighbors) and make contact with each another. One assumes the role of speaker and the other is the hearer.
- Start Diagnosis: The speaker chooses one line from its program (using Q-learning) and sends the codeline to the hearer location using the `RUNINSTR` instruction.
- Action: The hearer reviews its program, in this case compares its code with the line of code given by the speaker, and reports weather its program has this instruction or not (`INSTRRES` instruction). If the hearer does not have this line, the code line and a vote are added to a vector of possible code lines.

Instruction(syntax)	Definition
RUNINSTR *(RUNINSTR-codeline,x, y)*	Indicates to the hearer a code line of the program from the speaker *codeline* and the position of the speaker *(x, y)*
INSTRRES *(INSTRRES-result,x, y)*	Indicates to the speaker if the hearer has the codeline or not and the current position of the hearer *(x,y)*

Table 2: Diagnose instructions

- Feedback: If the hearer has this instruction, the diagnosis ends in failure (it does not discover a possible failure), the question of diagnosis about this line is punished using Q-learning. Otherwise, the hearer receives a positive vote for this code line, and the diagnose question is rewarded using Q-learning.

Q-learning (Watkins, 1989), is used to optimize the questions of diagnosis. There are questions of diagnosis about each code line per agent and weights associated with each code line which are stored in a vector of diagnosis questions.

If an error is detected (the hearer does not have the speaker's line), the question of diagnosis about this line of code receives a reward and otherwise the question receives a punishment. The goal of the agents in Q-learning is to maximize their total reward (Alpaydin, 2004). Questions of diagnosis with the greatest value are selected; if there is more than one question with the same greatest value, we choose the first one in the diagnosis vector.

The following equation is the reward when a failure is diagnosed:

$$d[c] = d[c] + \eta * (r + \gamma * Max_i(d[i]) - d[c]) \qquad (3)$$

If a failure is not found, the following punishment for the question of diagnosis is given:

$$d[c] = d[c] - \eta * (r + \gamma * Max_i(d[i]) - d[c]) \qquad (4)$$

Where:

- d is the vector of weights about diagnosis questions.
- c is the selected codeline for the diagnosis.
- η is the learning rate $(0 < \alpha < 1)$.
- r is the reward for taking the action.
- γ is the discount factor for the maximum of the weights.

Voting system

Each termite has a vector called belief vector of failures which stores the feedback of the diagnosis based on language games (a vote is added if the hearer does not have the code line that the speaker indicated). In this case, a value of 1 is added for this line of code if it belongs to the vector, otherwise the code line is added to this vector with a vote equal to 1. Table , shows four votes for the code line `100100`, and three votes for the line `101011`.

codeline	votes
100100	4
101011	3
101100	2
100101	1
110001	1

Table 3: belief vector of failures for a termite

Self-healing

Self-healing is defined using some concepts of evolutionary algorithms. Evolutionary algorithms (EA) are optimization techniques based on the principles of natural evolution (Holland, 1992). First, a threshold was defined for the code lines in the belief vector of failures. If a code line of the belief vector of failures reaches this threshold (five votes in this case), it is introduced in a random position of the termite program, instead of adding another line. It could be considered like an operator of an EA. When this operator is applied, the introduced code line and its votes are removed from the belief about failures vector of this termite and the termite will disable the diagnosis instruction (Termite is sick so it cannot diagnose others), which is useful for avoiding failure propagation (Fig 4).

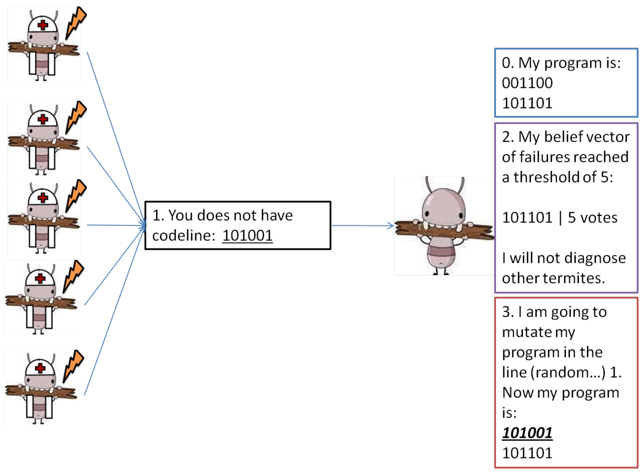

0. My program is:
001100
101101
2. My belief vector of failures reached a threshold of 5:
101101 | 5 votes
I will not diagnose other termites.
1. You does not have codeline: 101001
3. I am going to mutate my program in the line (random...) 1. Now my program is:
101001
101101

Figure 4: Self-healing process

Dynamic of the process

Each termite gets their programs from the queen (base program). The base program is copied to all the termites and some termites of the population get bad copies of their programs (see failures definition section to get details). Some termites will be healthy and others will be sick and will act in unexpected ways. After that, the termites load and execute their programs. Thanks to the program, the termites know that they must look for food, carry food or make diagnostics.

Sick termites can diagnose healthy termites, so if a healthy termite receives bad diagnosis from sick termites (reach the threshold of the belief vector of failures), a code line of the healthy termite would be replaced and the healthy termite can get sick (see Self-healing section to get details) and disable its diagnosis instruction. In the same way a sick termite that is diagnosed by healthy termites, change their code, disable their diagnose instruction to avoid failure propagation and can be healed. With the time, the self-healing mechanism of programs avoid failure propagation and to induce changes in the lines of code of the sick termites decreasing disease.

Experiments and Results

A virtual world was defined and each agent and food point were given a size of 1x1. For the Q-learning equations (Eqs 3 and 4) the following parameters were set: $\eta = 0.01$, $\gamma = 0.06$, and a $r = 1$. Each question of diagnosis has an initial weight of $1/codelines$. There was a population of 100 and 200 termites with 0.1, 0.5 and 0.7 as the probability of failure (pf) in the program for the population at startup (see the faiures definition section for details). Code to validate if a program has been healed was introduced, but the agents have no knowledge about it.

Each experiment was performed 30 times, with 100000 iterations (movements per termite) per experiment. Data in Tables 4 and 5 presents the mean and the standard deviation of the experiments in terms of:

- PF = probability of Failure
- TS = Termites Sick at the Beginning are the termites that get sick by bad copy of their programs.
- TSBD = Termites Sick by Bad Diagnosis are the termites that get infected by bad diagnosis.
- TH = Termites Healed are termites which changed their code and got a code with the same instructions of the base program.
- TSDS = Termites Sick During Simulation are all the termites that got sick during the simulation (TS+TSBD).
- TSAS = Termites Sick After Simulation (TSDS - TH).

PF	TS	TSBD	TH	TSAS
0.1	9.76 ± 2.31	7.13 ± 4.59	16.4 ± 5.92	0.5 ± 0.68
0.3	30.2 ± 4.32	18.6 ± 5.44	42.43 ± 5.70	6.4 ± 4.07
0.5	$49,6 \pm 7.43$	24.23 ± 6.60	47.06 ± 7.89	26.7 ± 6.14
0.7	68.5 ± 10.02	20.1 ± 5.89	$20,2 \pm 11.34$	68.4 ± 13.92

Table 4: Experiments with 100 termites.
PF = probability of Failure, TS = Termites Sick at the Beginning, TSBD = Termites Sick by Bad Diagnosis, TH = Termites Healed, TSAS = Termites Sick After Simulation

PF	TS	TSBD	TH	TSAS
0.1	20.26 ± 4.64	9.63 ± 5.54	29.26 ± 7.89	0.63 ± 1.12
0.3	57.30 ± 7.42	27.53 ± 9.37	72.53 ± 8.67	12.30 ± 8.73
0.5	97.87 ± 15.24	53.43 ± 9.86	97.70 ± 15.59	53.60 ± 24.23
0.7	134.97 ± 9.52	32.27 ± 8.03	48.17 ± 13.52	119.07 ± 23.72

Table 5: Experiments with 200 termites.
PF = probability of Failure, TS = Termites Sick at the Beginning, TSBD = Termites Sick by Bad Diagnosis, TH = Termites Healed, TSAS = Termites Sick After Simulation

Pf		Mean	Std. Deviation.	Std. Error mean
0.1	TSDS	16.167	5.977	1.091
	TSAS	.70	.952	.174
0.3	TSDS	48.833	7.368	1.345
	TSAS	6.40	3.490	.637
0.5	TSDS	73.833	10.952	1.999
	TSAS	26.77	15.542	2.838
0.7	TSDS	88,633	7,513	1.372
	TSAS	68.40	13.922	2.542

Table 6: Paired Samples Statistics (100 termites, N = 30)

Pf	Correlation	Sig
0.1	.045	.812
0.3	.660	.000
0.5	.879	.000
0.7	.582	.001

Table 7: Paired Samples Correlations between TSDS and TSAS (100 termites, N = 30)

To determine if the algorithm is efficient, a t-test for related samples was performed with the following hypothesis. Results are organized in terms of total sick termites that got sick during the simulation (TSDS) and termites sick at the end of the simulation (TSAS):

- H_o : the mean of the termites that got sick during the simulation TSDS ($TSDS = TS+TSBD$) is equal to the mean of sick termites at the end of the simulation TSAS ($TSAS = TSDS - TH$).
- H_a : the mean of the total of termites sick $>$ the mean of the termites sick at the end of the simulations ($TSDS > TSAS$).

A value of $\alpha = .05$ is selected for the tests (this value is the most used in social sciences), this means that five times out of a hundred a statistically significant difference between the means is found even if there was none.

For experiments with 100 termites, the means showed a difference between the termites sick during the simulation and the termites sick at the end of the simulation. The difference between the means is 15.467, the value of t is 14.096 for experiments with 0.1 as the failure probability. In the experiments with 100 and 0.3 of failure probability the difference between the means is 42.433, the value of t is 40.750. With 100 and 0.5 of failure probability the difference between the means is 47.067, the value of t is 32.651. For 0.7 we have a difference between means of 20.233 and a t value of 9.773. For 100 termites and pf = 0.1, Table 7 presents a sig value greater than .05 but the Paired Samples Test of table 8 reveal a statistically reliable difference between the means. The null hypothesis is rejected in all cases, so the algorithm is efficient for the 100 termites and pf (0.1, 0.3, 0.5, 0.7) (Tables 6 and 8).

For experiments with 200 termites, the means also showed a difference between the total sick termites and the termites sick at the end of the simulation. For $pf = 0.1$ the difference between the means is 29.266, the value of t is 20.314. In the experiments with $pf = 0.3$ the difference between the means is 75.533, the value of t is 45.848. With $pf = 0.5$ of failure probability the difference between the means is 97.700, the value of t is 34.325. For $pf = 0.7$ we have a difference between means of 48.167 and a t value of 19.518. The null hypothesis is rejected in all cases, so the algorithm also is efficient for 200 termites and pf (0.1, 0.3, 0.5, 0.7) (Tables 9, 10 and).

Pf		mean	Std. Deviation.	Std. Error mean
0.1	TSDS	29.9	8.442	1.541
0.1	TSAS	.633	1.129	.206
0.3	TSDS	84.833	13.774	2.515
0.3	TSAS	12.30	8.730	1.594
0.5	TSDS	151.300	17.542	3.203
0.5	TSAS	53.60	24.234	4.424
0.7	TSDS	167.233	13.566	2.477
0.7	TSAS	119.07	23.718	4.330

Table 9: Paired Samples Statistics (200 termites, N = 30)

Conclusions and Future Work

A mechanism of programs self-healing based on language games, Q-learning and evolutionary computing was pre-

	Paired Differences pf = 0.1				Paired Differences pf = 0.3		
	Mean	Std. Deviation	Std. Error Mean		Mean	Std. Deviation	Std. Error Mean
TSDS-TSAS pf = 0.1	15.467	6.010	1.097	**TSDS-TSAS pf = 0.3**	42.433	5.704	1.041
	95% Confidence Interval for the difference	Lower	13.223		95% Confidence Interval for the difference	Lower	40.304
		Upper	17.711			Upper	44.563
	t	df	sig		t	df	sig
	14.096	29	.000		40.750	29	.000
	Paired Differences pf=0.5				**Paired Differences pf = 0.7**		
	Mean	Std. Deviation	Std. Error Mean		Mean	Std. Deviation	Std. Error Mean
TSDS-TSAS pf = 0.5	47.067	7.896	1.442	**TSDS-TSAS pf = 0.7**	20.233	11.340	2.070
	95% Confidence Interval for the difference	Lower	44.11841		95% Confidence Interval for the difference	Lower	15.999
		Upper	50.01493			Upper	24.46781
	t	df	sig		t	df	sig
	32.651	29	.000		9.773	29	.000

Table 8: Paired Samples Test (100 termites)

	Paired Differences pf = 0.1				Paired Differences pf = 0.3		
	Mean	Std. Deviation	Std. Error Mean		Mean	Std. Deviation	Std. Error Mean
TSDS-TSAS pf = 0.1	29.266	7.891	1.441	**TSDS-TSAS pf = 0.3**	72.533	8.665	1.582
	95% Confidence Interval for the difference	Lower	26.320		95% Confidence Interval for the difference	Lower	69.298
		Upper	32.213			Upper	75.769
	t	df	sig		t	df	sig
	20.314	29	.000		45.848	29	.000
	Paired Differences pf=0.5				**Paired Differences pf = 0.7**		
	Mean	Std. Deviation	Std. Error Mean		Mean	Std. Deviation	Std. Error Mean
TSDS-TSAS pf = 0.5	97.700	15.589	2.846	**TSDS-TSAS pf = 0.7**	48.167	13.516	2.468
	95% Confidence Interval for the difference	Lower	91.879		95% Confidence Interval for the difference	Lower	43.120
		Upper	103.521			Upper	53.214
	t	df	sig		t	df	sig
	34.325	29	.000		19.518	29	.000

Table 11: Paired Samples Test (200 termites)

Pf	Correlation	Sig
0.1	.539	.002
0.3	.794	.000
0.5	.767	.000
0.7	.876	.000

Table 10: Paired Samples Correlations between TSDS and TSAS (200 termites, N = 30)

sented. The system diagnoses and heals failures in an efficient way even with a 70% of the sick population. We observed that each termite is able to identify its own failures given the diagnosis of others.

Local interactions in the mechanism of diagnosis allow the system to be specialized in the detection of more than a failure at the same time even if the failure is different per termite. By running the simulation, it was observed that some sick termites caused bad diagnosis, which induced failures in other termites. However, the rule that states that a termite cannot diagnose other if a failure is detected (votes threshold = five), makes that after some iterations, the termites stop propagating the failure and the population continues evolving their code until programs are recovered and the number of programs that were bad is reduced. In all cases the healing mutations stop after several iterations (Tables 4 and 5).

In all the experiments performed, the mean of the termites sick during the simulation (TSDS) is greater than the mean of the termites sick at the end of the simulation (TSAS) (Tables 6 and 9), so the null hypothesis (H_o : the mean of the termites that got sick during the simulation is equal to the mean of sick termites at the end of the simulation) is rejected given the statistical analysis. With the time, the self-healing mechanism avoids failure propagation and induces changes in the lines of code of the sick termites obtaining less sick termites that the termites sick during the simulation. In this way, self-healing is an emergent property that arises from local interactions between termites (diagnosis based on language games).

As future work we are going to include some improvements like allowing the termite to locate the code line of the failure and perform diagnosis to others even if a failure is detected.

Acknowledgments

This work was supported by the DIB (Dirección de investigación sede Bogotá- Universidad Nacional de Colombia) and its program ”Apoyo de la DIB a tesis de investigación en posgrados - Segundo Corte” code 12153, and by the alife research group - National University of Colombia.

References

Alpaydin, E. (2004). *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

Arora, H., Raghu, T., Vinze, A., and Brittenham, P. (2006). Collaborative self-configuration and learning in autonomic computing systems: Applications to supply chain. pages 303–304.

Atighetchi, M. and Pal, P. (2009). From auto-adaptive to survivable and self-regenerative systems successes, challenges, and future. pages 98–101.

Bicocchi, N. and Zambonelli, F. (2007). Autonomic communication learns from nature. *Potentials, IEEE*, 26(6):42–46.

Breitgand, D., Goldstein, M., Henis, E., Shehory, O., and Weinsberg, Y. (2007). Panacea towards a self-healing development framework. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 169 –178.

David, F. M. and Campbell, R. H. (2007). Building a self-healing operating system. pages 3–10.

De Wolf, T. and Holvoet, T. (2003). Towards autonomic computing: agent-based modelling, dynamical systems analysis, and decentralised control. pages 470–479.

Dorigo, M. and Gambardella, L. M. (1997). Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*.

Fuad, M., Deb, D., and Oudshoorn, M. (2006). Adding self-healing capabilities into legacy object oriented application. pages 51–51.

Gao, J., Kar, G., and Kermani, P. (2004). Approaches to building self healing systems using dependency analysis. 1:119–132 Vol.1.

Holland, J. H. (1992). *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA.

Kephart, J. and Chess, D. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Langton, C. G. (1989). *Artificial Life: proceedings of an interdisciplinary workshop on the Synthesis and Simulation of Living Systems*. Addison-Wesley.

Littman, M. L., Ravi, N., Fenson, E., and Howard, R. (2004). Reinforcement learning for autonomic network repair. pages 284–285.

Nami, M. and Bertels, K. (2007). A survey of autonomic computing systems. pages 26–26.

Nami, M. R. and Sharifi, M. (2007). Autonomic computing: A new approach. pages 352–357.

Rott, A. (2007). Self-healing in distributed network environments. 1:73–78.

Steels, L. (2001). Language games for autonomous robots. *Intelligent Systems, IEEE*, 16(5):16 – 22.

Steels, L. and Vogt, P. (1997). Grounding adaptive language games in robotic agents. In *Proceedings of the Fourth European Conference on Artificial Life*, pages 474–482. MIT Press.

Truszkowski, W., Hinchey, M., Rash, J., and Rouff, C. (2006). Autonomous and autonomic systems: a paradigm for future space exploration missions. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 36(3):279–291.

Truszkowski, W., Rash, J., Rouff, C., and Hinchey, M. (2004). Asteroid exploration with autonomic systems. pages 484–489.

Vassev, E. and Hinchey, M. (2009). Assl specification and code generation of self-healing behavior for nasa swarm-based systems. In *Engineering of Autonomic and Autonomous Systems, 2009. EASe 2009. Sixth IEEE Conference and Workshops on*, pages 77 –86.

Vassev, E. and Paquet, J. (2007). Assl - autonomic system specification language. In *Software Engineering Workshop, 2007. SEW 2007. 31st IEEE*, pages 300 –309.

Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, University of Cambridge,England.

Wikipedia (2011). Trophallaxis from wikipedia. http://en.wikipedia.org/wiki/Trophallaxis.

Zhou, R., Wei, R., Chen, G., Yang, Z., Shen, H., Zhang, J., and Luo, M. (2008). Ant colony inspired self-healing for resource allocation in service-oriented environment considering resource breakdown. In *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT '08. IEEE/WIC/ACM International Conference on*, volume 1, pages 66 –69.