A Probabilistic Model Checking Technique for the Verification of Self-Organising Emergent Systems*

Lucio Mauro Duarte[1], Luciana Foss[2], Flávio Rech Wagner[1], Tales Heimfarth[3]

[1]Instituto de Informática – UFRGS – Porto Alegre – RS – Brazil

[2]Instituto de Física e Matemática – DINFO – UFPel – Pelotas – RS – Brazil

[3]Departamento de Ciência da Computação – UFLA – Lavras – MG – Brazil

{lmduarte,flavio}@inf.ufrgs.br

luciana.foss@ufpel.edu.br, tales@dcc.ufla.br

Abstract. *Developing self-organising emergent systems (SOESs) poses a great challenge to current software engineering approaches. Model-checking such systems is difficult as their behaviour cannot be linearly described and their structure can change dynamically. We propose a technique to model and verify SOESs applying probabilistic model checking. We applied our technique to a problem involving an SOES and obtained verification results about relevant properties of the system. These results demonstrate that our abstractions are appropriate for describing the behaviour of this SOES and indicate that they may be applied to other similar systems.*

Resumo. *O desenvolvimento de sistemas emergentes auto-organizáveis (SEAOs) constitui-se em um grande desafio para as atuais abordagens de engenharia de software. Realizar verificação de modelos para estes sistemas é difícil, visto que seu comportamento não pode ser descrito linearmente e sua estrutura pode mudar dinamicamente. Neste artigo, propomos uma técnica para modelar e verificar SEAOs que utiliza verificação de modelos probabilística. Esta técnica foi aplicada a um problema envolvendo um SEAO, possibilitando a obtenção de resultados de verificação de propriedades relevantes. Tais resultados mostram que nossas abstrações são adequadas para descrever este SEAO e indicam que a mesma ideia pode ser aplicada a outros sistemas similares.*

1. Introduction

Self-organising emergent systems (SOES) [De Wolf and Holvoet 2005a] are one of the most promising answers to the development of massive distributed systems with decentralised control. An SOES comprises a set of quite simple, and not necessarily reliable, components that interact using local rules and autonomously adapt their behaviour to changes in the environment. As opposed to conventional systems, the global behaviour of such a system (macroscopic layer) cannot be simply defined as a linear combination of the behaviours of all its individual components (microscopic layer); it actually *emerges* from the local rules. For this reason, it is difficult to infer global properties just by looking at local properties of individual components.

*This work was supported by grant MCT/CNPq/CT-INFO 551031/2007-7.

The complexity of SOESs lies not on their components, but on the difficulty of predicting the behaviours that can emerge from local interactions between these components. Therefore, the system does not necessarily follow a linear behaviour, which means that SOESs cannot be designed by just dividing the problem and then assigning one part of the solution to each basic element of the system. Characteristics such as emergent behaviours and self-organisation have to be considered throughout the software development process so as to make it possible the early identification of undesirable behaviours and improve the safety and correctness of the system. Furthermore, when it comes to an SOES, even the idea of a correct behaviour changes, since many questions do not have just "true" or "false" as possible answers, but might require a probabilistic or stochastic analysis.

As a result, conventional software engineering techniques are, in general, not appropriate for the development of SOESs [De Wolf and Holvoet 2005b]. In particular, modelling the dynamic change of behaviour according to environment conditions that characterises an SOES is difficult, due to the requirement of predicting all possible behaviours beforehand. Moreover, the complexity resulting from the heterogeneity of system components and their interactions generally trespasses the limits of current software engineering paradigms and methodologies. Therefore, new techniques and tools need to be provided to support the development of this new type of systems.

1.1. Motivation

During the development of the project "A Methodology for the Development of Computational Systems Considering the Transition From Silicon to New Technologies", supported by the Brazilian Research Council (CNPq), we have studied a possibly complete methodology for the design and implementation of SOESs. This is part of the problem presented in [Wagner et al. 2009] as a refinement of two of the Grand Challenges in Computer Science Research in Brazil[1]. Our general idea is to provide a whole set of techniques and tools to cope with the characteristics of these new systems. Our study involves identifying conventional approaches that could be extended and/or adapted to give the necessary support, thus developing new techniques and tools from existing ones.

In this paper, we focus on a particular part of the methodology, which is the *verification of SOESs*. This verification involves modelling the system behaviour, specifying properties that it should preserve, and checking that the necessary properties indeed hold in the model. In this context, we apply *model checking* [Clarke et al. 1999], which is one of the most important approaches for the analysis of properties of systems. It provides an automatic way of checking that a system, represented as a finite-state model, preserves some temporal properties. More specifically, we use *probabilistic model checking* [Vardi 1985], which is an extension of model checking that allows the assignment of probabilities and timing information to transitions.

Applying model checking is usually not trivial because it requires the thorough exploration of a model of the system, which means that the model size has to be controlled via abstractions precise enough to guarantee confidence on the results, but coarse enough to not be intractable by model checking tools [Duarte et al. 2008]. This is even more

[1] Available from http://www.sbc.org.br/index.php?language=1&subject=8&content=downloads&id=231.

challenging when one considers the non-linearity of an SOES behaviour. Furthermore, the dynamic aspects of the system must also be represented in the models.

1.2. Objectives

Essentially, engineering SOESs has two big challenges [Gardelli 2008]: *how to design the individual entities to produce a target global behaviour* and *how to provide guarantees of some sort about the emergence of a specific behaviour*. In this paper, we focus on trying to find an answer for the second question by applying probabilistic model checking.

There have been a few attempts to model and verify an SOES using probabilistic model checking, such as [Casadei and Viroli 2009] and [Gardelli 2008], where they verify properties of coordination systems and properties of multi-agent systems, respectively, using the Probabilistic Symbolic Model Checker (PRISM) [Kwiatkowska et al. 2002]. They use simulation to help the verification process, either by using it to adapt the model to the abstractions necessary to check some property or by applying simulation to produce results to complement verification. Unlike these approaches, we do not apply simulation as part of our technique. Hence, the results of our analysis are more precise, since they are not based on approximations considering a set of scenarios obtained from an abstraction of the system. Rather, our results come from a complete exploration of this abstraction, thus increasing our confidence on the correctness of the model.

We present an approach focused on modelling and verifying an SOES. This approach involves the use of the PRISM tool to model the system, specify the relevant properties using a probabilistic temporal logic, and check these properties against the model using probabilistic model checking. To demonstrate this approach, we describe results of a case study involving the *ant colony optimisation* (ACO) [Dorigo and Gambardella 1997] applied to the *travelling salesman problem* (TSP) [Applegate et al. 2006]. The ACO defines a bio-inspired self-organisation algorithm and has been used to guide the solution for the TSP. We combined the TSP and the ACO in a small scenario aiming to investigate how to model the problem and, in particular, how to model self-organisation and deal with emergent behaviours.

We describe how we modelled the problem as a discrete-time Markov chain (DTMC) in the PRISM language and present some probabilistic properties we checked. The results indicate that the application of the ACO to the TSP eventually leads to a behaviour that complies with requirements for solving the problem. Therefore, the results show that our abstractions successfully capture the desired characteristics of the scenario, in particular those related to emergent behaviours and self-organisation. This is demonstrated by the fact that, as expected, the system converges to the expected solution. This result supports our confidence that our modelling ideas could be applied in other scenarios to describe self-organising mechanisms and check emergent behaviours.

1.3. Paper Structure

This paper is organised as follows: Section 2 presents background information on probabilistic model checking; Section 3 describes how we modelled and verified the problem and the results we obtained; Section 4 presents a discussion about related work regarding the verification of SOESs; and Section 5 contains the conclusions.

2. Background

This section presents the concepts involving probabilistic model checking and a brief description of the PRISM tool. We discuss why probabilistic model checking is more appropriate for verifying properties of an SOES than traditional model checking and present the basic ideas on how to model behaviours and specify probabilistic properties.

2.1. Probabilistic Model Checking

Probabilistic model checking [Vardi 1985] is a model checking technique for the analysis of systems that exhibit probabilistic or stochastic behaviour. It mainly differs from traditional model checking in that it involves additional information on probabilities or timing of transitions between states. There are several commonly used model representations for probabilistic and stochastic systems, most of them based on Markov chains, which are models traditionally used for performance and reliability analysis. In Markov chains, transitions between states depend on some probability distribution, where only the current state of the system influences the probability of the next transitions. Depending on how time is treated, a Markov chain can be a *discrete-time Markov chain* (DTMC) or a *continuous-time Markov chain* (CTMC). A DTMC model is represented by a transition system that defines the probability of moving from one state to another considering discrete steps, whereas in a CTMC system state changes can occur at any arbitrary time and the probability of moving to a next state depends on transition rates. We concentrate on DTMCs, as we deal with discrete time events.

Properties of a DTMC are specified using the *Probabilistic Computation Tree Logic* (PCTL) [Hansson and Jonsson 1994]. PCTL extends the temporal logic CTL [Ben-Ari et al. 1983] with discrete time and probabilities. It is used to express properties over states or paths. The syntax of PCTL is as follows:

state formulas: $\phi ::= true \mid a \mid \phi \ \wedge \ \phi \mid \neg\phi \mid \mathcal{P}_{\bowtie p}[\psi]$
path formulas: $\psi ::= \mathcal{X}\ \phi \mid \phi\, \mathcal{U}^{\leq k}\ \phi \mid \phi\, \mathcal{U}\ \phi$

where $\bowtie \in \{\leq, \geq, <, >\}$, $0 \leq p \leq 1$ and $k \in \mathbb{N}$. A property of a model will always be expressed as a state formula. Path formulas only occur as the parameter of the probabilistic path operator. A state formula can be an atomic proposition a, formulas $true$, $\phi \ \wedge \ \phi$ or $\neg\phi$, having their usual meaning, or the probabilistic path operator $\mathcal{P}_{\bowtie p}[\psi]$. Intuitively, a state s satisfies $\mathcal{P}_{\bowtie p}[\psi]$ if the probability of taking a path from s satisfying ψ is in the interval specified by $\bowtie$. Path formulas contain the next ($\mathcal{X}$), the bounded until ($\mathcal{U}^{\leq k}$) or the unbounded until ($\mathcal{U}$) operator, all of which are standard in CTL. Intuitively, $\mathcal{X}\phi$ is true if ϕ is satisfied in the next state; $\phi_1 \mathcal{U}^{\leq k} \phi_2$ is true if ϕ_2 is satisfied within k time-steps and ϕ_1 is true up until that point; and $\phi_1 \mathcal{U} \phi_2$ is true if ϕ_2 is satisfied at some point in the future and ϕ_1 is true up until then.

2.2. Probabilistic Symbolic Model Checker - PRISM

The Probabilistic Symbolic Model Checker (PRISM) [Kwiatkowska et al. 2002] is a tool used for verifying quantitative properties. The tool takes two inputs: a model description (written in the PRISM language) and a property (e.g., specified in PCTL). It parses this description, constructs a model of the appropriate type (either a DTMC or a CTMC) and then determines the set of reachable states of this model and checks whether these states preserve the required properties.

A model in the PRISM language is composed by modules that can interact with each other. A module contains a set of local variables whose values constitute the state of the module. The global state of the model is determined by the combination of the local states of all modules, along with the values of global variables. All variables are typed as integer or boolean. Integer variables are defined as a range of values, so as to allow bounded model checking. Besides variables, constants of type integer, boolean or double can be defined.

The behaviour of each module is described by a set of commands of the form: $[]g \rightarrow \lambda_1 : u_1 + \ldots + \lambda_n : u_n$. The guard g is a predicate over all the variables in the model (including those belonging to other modules). Each update u_i describes a transition that the module can make if the guard is true. A transition is specified by giving the updated values of the variables in the module, possibly as an expression formed from other variables or constants. The expressions λ_i are used to assign probabilistic information to the transitions.

3. Verifying a Self-Organising Emergent System

This section presents the results of a case study on verifying properties of an SOES. This case study shows the application of our ideas for modelling and verifying SOESs. The problem we chose to model was the travelling salesman problem, based on the criteria that this is a famous complex problem and, yet, easy to understand. To drive the solution to the problem, we applied the ant colony optimisation, which is a bio-inspired mechanism that has already been used to speed up the process of finding a solution to this problem [Dorigo and Gambardella 1997]. This mechanism adds the necessary characteristics of emergence and self-organisation to the problem scenario, thus allowing us to apply our technique and demonstrate that it is appropriate as part of a methodology for the development of SOESs.

3.1. Problem Description

The *travelling salesman problem* (TSP) [Applegate et al. 2006] is a classic and well-studied problem in Theoretical Computer Science that can be used to formalise a number of real-world problems [Cormen et al. 2001]. It involves determining the shortest tour for a salesman through a given finite set of n cities. The salesman starts his tour in some city and must visit all other $n-1$ cities exactly once before returning to the initial city. More formally, this problem consists in finding the shortest Hamiltonian circuit [Gould 1991] in a fully-connected graph $G = (N, E)$, where N represents the set of n cities and E describes the set of routes between pairs of cities.

As a metric to determine the shortest tour, each route $(i, j) \in E$ is assigned a cost $cost(i, j)$, describing the distance between city i and city j. Thus, the total cost of a certain tour $t = (c_0, c_1, ..., c_n)$ is given by the sum presented in (1), which represents the sum of the costs of every route included in the tour. Therefore, the shortest path would be the one with the lowest total cost.

$$\sum_{i=0}^{n-1} cost(i, (i+1)\ mod\ n) \ . \tag{1}$$

One way proposed to help solve the TSP is the use of the *ant colony optimisation* (ACO) [Dorigo and Gambardella 1997], which is an algorithm that reproduces the behaviour of ants looking for food sources. As they move around, ants mark the paths they take by leaving a trail of *pheromone*, which is a natural hormone. Paths more frequently used are those that contain good food sources, thus having higher concentrations of pheromone (positive feedback). Hence, paths leading to food sources close to the nest tend to be more used, concentrating higher levels of pheromone and becoming more attractive to other ants. After some time, the pheromone on paths decays, which guarantees that paths not used very often will have less probability of being taken in the future (negative feedback). This means that the pheromone mechanism provides all the information ants need to choose paths to be taken and this interaction through concentrations of pheromone is enough for them to achieve the necessary organisation. Therefore, their system involves the idea of self-organisation and the behaviour of the whole colony emerges from the interactions through pheromone, thus characterising an SOES.

The algorithm of the ACO applied to the TSP described in [Dorigo and Gambardella 1997] proposes to use an abstraction of pheromone to guide the choice of paths between N cities in order to find the shortest tour. The probability of taking a certain route at each part of the path is based on the length of the route and on the concentration of pheromone on that route, which is called the *desirability* of the route. Therefore, the objective is that, after some iterations of the algorithm, shorter paths should have higher probabilities of being taken. The decay of pheromone is represented by a decrease of pheromone concentration and recalculation of probabilities of all paths after some time units, which ensures that long paths may eventually be completely avoided.

3.2. Modelling

We modelled the symmetric version of the TSP (i.e., for a route (i, j), $cost(i, j) = cost(j, i)$) using the PRISM language. To control the state space we decided to work with a graph composed by 4 cities, where each city is identified by a sequential number and city number 1 is always the initial city (which simplifies the modelling of the problem but does not affect the analysis). The number of cities is related to the minimum number of vertices required to introduce some complexity to the problem of choosing paths.

Instead of modelling several individual ants, we constructed a model with one ant that iteratively travels along the edges of the graph. This way we simulate with one ant the work of many, each one represented by a tour executed by this single ant. Note that this reduces the complexity of the problem, but does not influence the results of any analysis on the model. The final result is essentially the same as having multiple ants moving around simultaneously.

Although we followed the original algorithm when modelling the ACO, calculating the probabilities of paths based both on costs and desirability (amount of pheromone), we adopted different formulas to simplify the model and the verification. Our desirability component, called *preference*, refers to the amount of pheromone associated to each route between two cities and ranges from 1 (MIN_PREF) to 10 (MAX_PREF). All routes are initialised with MIN_PREF and, after each cycle, are updated according to usage and path lengths. The local update of preferences after a tour is calculated as presented in

(2), where p_{ij} denotes the preference of the route (i, j) and tot_dist is the sum of the costs of all routes comprising the most recent tour taken. The function inc_factor determines by how much the preference of a route will be increased depending on the total cost of the complete path. It assigns an increase value to paths according to the group they belong to (shortest, mid-length or longest), which is determined by the definition of the costs assigned to each route.

$$p'_{ij} = min(p_{ij} + inc\_factor(tot\_dist), MAX\_PREF) \ . \tag{2}$$

The probability of taking a certain route (i, j) is given by the formula in (3), where N_CITIES is the number of cities involved (in our case, 4) and $visited\_cities$ is the set of cities already visited during this tour.

$$prob_{ij} = p_{ij} / \sum_{k=1}^{N\_CITIES} p_{ik} \quad \text{s.t. } k \in \{1, ..., N\_CITIES\} - visited\_cities \ . \tag{3}$$

Consider, for instance, a scenario with four cities. From the initial city 1, the probability of taking, for example, route $(1, 2)$ is given by the formula $prob_{12} = p_{12}/(p_{12} + p_{13} + p_{14})$.

Figure 1 presents the PRISM model of the TSP with the ACO. We chose to model the problem as a DTMC, since our focus is on determining that our modelling correctly abstracts a behaviour that leads to the solution of the problem and not on how long it takes for this solution to be found. Although the system modelled has only four cities, the PRISM model is quite large since we need to define the system behaviour for each sequence of visited cities. This is because the PRISM language does not provide any abstract structure, such as arrays, that could allow us to generalise these behaviours. Note, however, that a tool to automatically generate models like this one could be easily developed, but our focus is on the technique and not on the case study.

Each constant `Dij` determines the distance (or cost) between cities `i` and `j`. Constants `EVAP_RATE` and `N_CYCLES` are used to set the rate at which the pheromone on routes evaporates and the number of cycles to be executed, respectively. Their values are assigned at the beginning of the verification, so that it is possible to analyse different scenarios with only one model just by varying the values of these parameters.

Based on the defined constants, Figure 2 shows an illustration of the scenario considered in our experiments. Each circle represents a city and each edge determines the route between two cities. Distances are displayed next to each route.

Variable `cont` in the model (see Figure 1) controls the number of cycles, acting as a counter, whereas each variable `pij` defines the preference of a route (`i`,`j`). Lines 20-30 describe the formulas used to calculate the probability `probij` of each route (i, j). The distances defined for each route in the graph create 3 groups of path lengths: the longest paths have a total distance of 19, mid-length paths have a cost of 15, and the shortest paths have a total distance of 10. We simplified the calculation by testing, according to the total

```
1  dtmc
2
3  const int N_CITIES = 4;
4  const int MIN_PREF = 1; const int MAX_PREF = 10;
5  const int D12 = 1; const int D23 = 8;
6  const int D34 = 2; const int D14 = 4;
7  const int D13 = 4; const int D24 = 3;
8  const int MAX_DIST = (D12+D23+D34+D14+D13+D24);
9  const double EVAP_RATE;
10 const int N_CYCLES;
11
12 global cont : [0..N_CYCLES] init 0;
13 global p12 : [MIN_PREF..MAX_PREF] init MIN_PREF;
14 global p13 : [MIN_PREF..MAX_PREF] init MIN_PREF;
15 global p14 : [MIN_PREF..MAX_PREF] init MIN_PREF;
16 global p23 : [MIN_PREF..MAX_PREF] init MIN_PREF;
17 global p24 : [MIN_PREF..MAX_PREF] init MIN_PREF;
18 global p34 : [MIN_PREF..MAX_PREF] init MIN_PREF;
19
20 formula prob12 = (p12/(p14+p13+p12));
21 formula prob13 = (p13/(p14+p13+p12));
22 formula prob14 = (p14/(p14+p13+p12));
23 formula prob123 = (p23/(p23+p24));
24 formula prob124 = (p24/(p23+p24));
25 formula prob132 = (p23/(p23+p34));
26 formula prob134 = (p34/(p23+p34));
27 formula prob142 = (p24/(p24+p34));
28 formula prob143 = (p34/(p24+p34));
29 formula inc_factor =
30 (tot_dist<15) ? 7 : ((tot_dist=15) ? 4 : 1);
31
32
33 module traveller
34
35   loc : [1..N_CITIES+1] init 1;
36   path : [0..6] init 0;
37   tot_dist : [0..MAX_DIST] init 0;
38
39   [] loc=1 & path=0 -> prob12 : (loc'=2) +
40                        prob13 : (loc'=3) +
41                        prob14 : (loc'=4);
42
43   [] loc=2 & path=0 ->
44              prob123 : (loc'=3) &
45                        (path'=1) &
46                        (tot_dist'=D12+D23+D34+D14) +
47              prob124 : (loc'=4) &
48                        (path'=2) &
49                        (tot_dist'=D12+D24+D34+D13);
50   [] loc=3 & path=0 ->
51              prob132 : (loc'=2) &
52                        (path'=3) &
53                        (tot_dist'=D13+D23+D34+D14) +
54              prob134 : (loc'=4) &
55                        (path'=4) &
56                        (tot_dist'=D13+D34+D24+D12);
57   [] loc=4 & path=0 ->
58              prob142 : (loc'=2) &
59                        (path'=5) &
60                        (tot_dist'=D14+D24+D23+D13) +
61              prob143 : (loc'=3) &
62                        (path'=6) &
63                        (tot_dist'=D14+D34+D23+D12);
64
65   [] (loc=2|loc=3|loc=4) & path!=0 -> 1.0 : (loc'=5);
66
67   [] loc=5 & path=1 -> 1.0 :
68              (path'=0) &
69              (p12'=min(p12+inc_factor,MAX_PREF)) &
70              (p23'=min(p23+inc_factor,MAX_PREF)) &
71              (p34'=min(p34+inc_factor,MAX_PREF)) &
72              (p14'=min(p14+inc_factor,MAX_PREF)) &
73              (tot_dist'=0);
74   [] loc=5 & path=2 -> 1.0 :
75              (path'=0) &
76              (p12'=min(p12+inc_factor,MAX_PREF)) &
77              (p24'=min(p24+inc_factor,MAX_PREF)) &
78              (p34'=min(p34+inc_factor,MAX_PREF)) &
79              (p13'=min(p13+inc_factor,MAX_PREF)) &
80              (tot_dist'=0);
81   [] loc=5 & path=3 -> 1.0 :
82              (path'=0) &
83              (p13'=min(p13+inc_factor,MAX_PREF)) &
84              (p23'=min(p23+inc_factor,MAX_PREF)) &
85              (p24'=min(p24+inc_factor,MAX_PREF)) &
86              (p14'=min(p14+inc_factor,MAX_PREF)) &
87              (tot_dist'=0);
88   [] loc=5 & path=4 -> 1.0 :
89              (path'=0) &
90              (p13'=min(p13+inc_factor,MAX_PREF)) &
91              (p34'=min(p34+inc_factor,MAX_PREF)) &
92              (p24'=min(p24+inc_factor,MAX_PREF)) &
93              (p12'=min(p12+inc_factor,MAX_PREF)) &
94              (tot_dist'=0);
95   [] loc=5 & path=5 -> 1.0 :
96              (path'=0) &
97              (p14'=min(p14+inc_factor,MAX_PREF)) &
98              (p24'=min(p24+inc_factor,MAX_PREF)) &
99              (p23'=min(p23+inc_factor,MAX_PREF)) &
100             (p13'=min(p13+inc_factor,MAX_PREF)) &
101             (tot_dist'=0);
102  [] loc=5 & path=6 -> 1.0 :
103             (path'=0) &
104             (p14'=min(p14+inc_factor,MAX_PREF)) &
105             (p34'=min(p34+inc_factor,MAX_PREF)) &
106             (p23'=min(p23+inc_factor,MAX_PREF)) &
107             (p12'=min(p12+inc_factor,MAX_PREF)) &
108             (tot_dist'=0);
109
110  [] loc=5 & path=0 & cont<N_CYCLES -> 1.0 :
111     (p12'=(max(p12-ceil(EVAP_RATE*p12),MIN_PREF))) &
112     (p23'=(max(p23-ceil(EVAP_RATE*p23),MIN_PREF))) &
113     (p34'=(max(p34-ceil(EVAP_RATE*p34),MIN_PREF))) &
114     (p14'=(max(p14-ceil(EVAP_RATE*p14),MIN_PREF))) &
115     (p13'=(max(p13-ceil(EVAP_RATE*p13),MIN_PREF))) &
116     (p24'=(max(p24-ceil(EVAP_RATE*p24),MIN_PREF))) &
117     (loc'=1) & (cont'=cont+1);
118
119  [] loc=5 & path=0 & cont=N_CYCLES -> 1.0 : true;
120
121 endmodule
```

Figure 1. PRISM model of TSP-ACO.

distance travelled to complete a path, which group this path belongs to. The shortest paths receive an increase of 7, mid-length paths have their preference incremented by 4, and only 1 is added to preferences of the longest paths[2].

Module `traveller` (lines 33-121) describes the behaviour of the artificial ant moving from city to city to execute tours on the graph. Variable `loc` determines which city the ant is currently in, variable `tot_dist` determines the total distance travelled during the tour and variable `path` determines which out of the 6 possible paths has been taken during the current tour. For our experiments, we assigned a numerical identification to each possible path as follows, where the sequences of numbers describe sequences of visited cities: *Path 1* = 1-2-3-4-1, *Path 2* = 1-2-4-3-1, *Path 3* = 1-3-2-4-1, *Path 4* = 1-3-4-2-1, *Path 5* = 1-4-2-3-1 and *Path 6* = 1-4-3-2-1.

[2]Command $\langle cond \rangle ? val1 : val2$ represents a selection operation where value $val1$ is returned in case the boolean expression $cond$ is evaluated as true and $val2$ is returned otherwise.

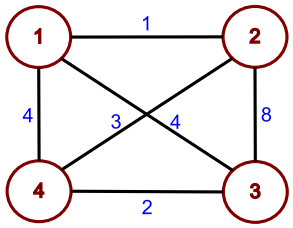

Figure 2. Scenario of experiment with TSP-ACO.

As mentioned before, the tour always starts in city number 1. Therefore, the initial state is that described in line 39. Consequently, there are three possible routes to take (lines 39-41), each one with its own probability. Because all preferences are initialised with the minimum preference (see lines 13-18), at the beginning, probabilities are the same for all possible routes. Depending on the next location, which is determined probabilistically, the choices for next route are different, so that we comply with the requirement that all cities should be visited only once during a tour. For instance, consider that a transition to city number 2 has been selected. Then lines 43-49 describe the behaviour at this location. If we reached city 2 from city 1, then there are still two cities to visit (3 and 4). Because there are only four cities involved, once the third city to be visited is chosen, we can already identify which path was taken. For example, if city number 3 is selected, we know for a fact that the next city is necessarily city number 4 and that, from there, we will move back to city number 1 to complete the tour. Hence, the path taken was 1-2-3-4-1, which is path 1 (line 45). Since we know the path, we can determine the total distance travelled (line 46).

We defined a special location (city 5) that is used to apply the necessary updates and is reached whenever a tour is completed (line 65). Depending on the path taken, the preferences of the route involved are updated according to the formula in (2) (lines 67-108). Variables `path` and `tot_dist` are reset to signal that the preference update has been executed, which allows the evaporation update to happen (lines 110-117). Using the defined evaporation rate, preferences are updated by removing part of the abstract pheromone. Function $ceil$ is used to guarantee that the result of the operation is an integer so that it falls into the range of the corresponding preference variable.

The evaporation rate occurs at the end of every tour until `cont` reaches the predetermined number of cycles. When all programmed cycles have been executed, the model enters a sink state (line 119). This limit on the number of cycles is necessary to avoid state-space explosion and to allow the execution of bounded model checking.

3.3. Property Specification

For the presented problem, we can define the following four requirements to guarantee that the system works properly:

1. If ant a_1 starts a tour with probability p_1 of finding one of the shortest paths, and ant a_2 starts its tour after a_1 with probability p_2, then $p_1 \leq p_2$;
2. The majority of ants will eventually follow one of the shortest paths;
3. Routes that compose the shortest paths tend to have their preferences increased after each cycle;
4. Routes belonging to the longest paths tend to become unused over time.

Requirement 1 defines that ants starting tours later must "learn" from previous tours through pheromone concentration to avoid long paths, increasing the probability of them taking one of the shortest paths, whereas requirement 2 ensures the emergence of a global behaviour that satisfies the collective aim, which is to turn the shortest paths into the most likely to be taken when a new tour begins. Requirement 3 determines that routes composing the shortest paths should become more attractive to ants after each cycle, while requirement 4 defines that routes in the longest paths tend to have a minimum probability of being taking after some cycles.

Properties of the TSP-ACO experiment were specified using PCTL based on the requirements presented above. The checked properties are presented below, where "stopped" is a label that represents the formula `(cont=T & loc=5)`, which defines the end of cycle `T`, where `T` is a parameter used during verification. Label "R12" represents the formula `((p12>p14)&(p12>p23))`, "R13" the formula `((p13>p14)&(p13>p23))`, "R24" the formula `((p24>p14)&(p24>p23))`, and "R34" the formula `((p34>p14)&(p34>p23))`. Labels "shortest_paths", "mid-length_paths" and "longest_paths" represent formulas `((path=2) | (path=4))`, `((path=1) | (path=6))` and `((path=3) | (path=5))`, respectively.

P1: `P=? [F (``stopped''&``R12''&``R13''&``R24''&``R34'')]`
P2: `P=? [F (``stopped'' & ``shortest_paths'')]`
P3: `P=? [F (``stopped'' & ``mid-length_paths'')]`
P4: `P=? [F (``stopped'' & ``longest_paths'')]`
P5: `P=? [F (``stopped'' & (p12=MAX_PREF) & (p34=MAX_PREF))]`
P6: `P=? [F (``stopped''& (p14=MIN_PREF) & (p23=MIN_PREF))]`

Property P1 asks what is the probability of the preferences of routes that compose the shortest paths being higher than those of the other routes when cycle `T` ends. Properties P2, P3 and P4 ask what is the probability of taking the shortest, the mid-length and the longest paths, respectively, when cycle `T` ends. Finally, properties P5 and P6 define, respectively, the probability of routes composing the shortest paths ($(1, 2)$ and $(3, 4)$) reaching saturation and the probability of routes belonging to the longest paths ($(1, 4)$ and $(2, 3)$) having the minimum preference.

3.4. Verification

Equipped with the model presented in Figure 1, considering the scenario shown in Figure 2 and having the properties described in Section 3.3, we were able to carry out some experiments using the PRISM tool. Our first experiment was to compare different evaporation rates and analyse how they affect the results of our properties. The objective was to check that in fact the pheromone correctly influences the preference for shorter paths and to determine what is the most appropriate evaporation rate to guarantee that our requirements are fulfilled. For this analysis, we used a model with 20 cycles, which was enough to detect a pattern of increase or decrease of probabilities, and `T` ranging from 0 to 20. To determine the number of cycles to be used, we executed a simulation of the results of property P2 considering 1000 cycles, which showed that the model reaches a firm majority (approx. 55%) after 20 cycles and remains around this value from that point on. It is worth emphasising that we applied simulation just as a means of accelerating the process, since we only needed a rough idea of how large our model should be to guarantee that the analyses would not be biased by the number of cycles considered. The use of verification confirms the stability of the results for property P2 from the 20th cycle on.

Table 1 presents the results for the properties for evaporation rates ranging from 0 to 0.8. Values 0.9 and 1 are ignored as they have the same results as those of value 0.8. Properties P5 and P6 were not relevant for this analysis.

Table 1. Property results for different evaporation rates.

	0.0	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**
P1	0.9927	0.8068	0.8406	0.7933	0.7134	0.4841	0.3455	0	0
P2	0.3459	0.4796	0.5006	0.5483	0.5616	0.5583	0.4805	0.4166	0.3333
P3	0.3386	0.3295	0.3547	0.3264	0.3014	0.2784	0.3031	0.2917	0.3333
P4	0.3155	0.1909	0.1447	0.1253	0.137	0.1633	0.2164	0.2917	0.3333

Analysing the results, we can see that the pheromone abstraction correctly guarantees that the shortest paths (probability of property P2) are, in almost all cases, more likely to be taken than longer paths (properties P3 and P4). Moreover, we can notice that extreme values, such as 0 and 0.8, reduce the pheromone effect as they either define no decay (value 0), causing preference to accumulate in all routes (that is why property P1 reaches nearly 100%), or determine a decay of a great amount of preference (value 0.8) that ends up balancing the probabilities of all paths. The values that guarantee the best results are those between 0.3 and 0.5, where a probability above 51% is achieved. However, amongst these values, it is with value 0.3 that we obtain the lowest probability result for the longest paths and the highest value for property P1.

Adopting value 0.3 as the evaporation rate, we produced a model with 20 cycles and verified the results for properties P2, P3 and P4 at each cycle. The goal was to obtain evidence that indeed the shortest paths tend to be taken with a higher probability after each cycle at the same time that the longest paths are bound to become less used. The results are displayed on the graph of Figure 3.

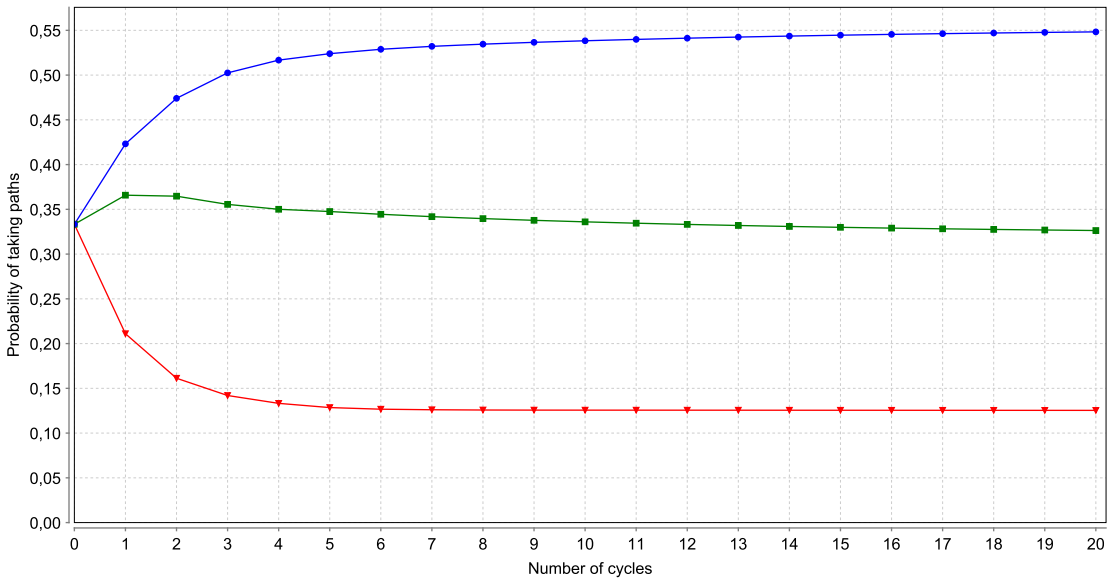

Figure 3. Properties P2 (line with circles), P3 (line with squares) and P4 (line with triangles upside down), with evaporation rate 0.3 and 20 cycles.

The graph clearly shows three separate behaviours, one for each group of paths.

All paths start with the same probability but, as the pheromone starts acting, the shortest paths already have the highest probability of being taken. This occurs because, if picked at the beginning of the first cycle, they receive more pheromone than the other paths. Therefore, at the end of the cycle, they would be the paths most likely to be taken during the next cycle. Note that their probability increases continually from the end of cycle 1 to the end of cycle 19, when it remains the same for the next cycle (in fact, it stays at the same level of about 55% from that point on). The probability of the longest paths falls quickly during the first 5 cycles and then remains around 12% for all the others.

Using properties P5 and P6, we analysed the probabilities of maximum and minimum preference of routes. The results are presented in the graph in Figure 4, which shows that the probability of saturation of the routes belonging to the shortest paths (line with circles) increases after each cycle up to cycle 16, where it reaches a probability over 79%. On the other hand, the probability of the routes belonging to the longest paths having the minimum preference (line with squares) decreases after the first cycles, since all the preferences are initialised as minimum and then the pheromone concentration is updated. However, this probability stops dropping after cycle 3 and slowly starts to increase, reaching a value above 52% at cycle 20 and tending to grow even more. These results show that indeed routes of the shortest paths become gradually more likely after each cycle whereas routes composing the longest paths become less likely, which moves the probability of taking one of these routes towards the minimum.

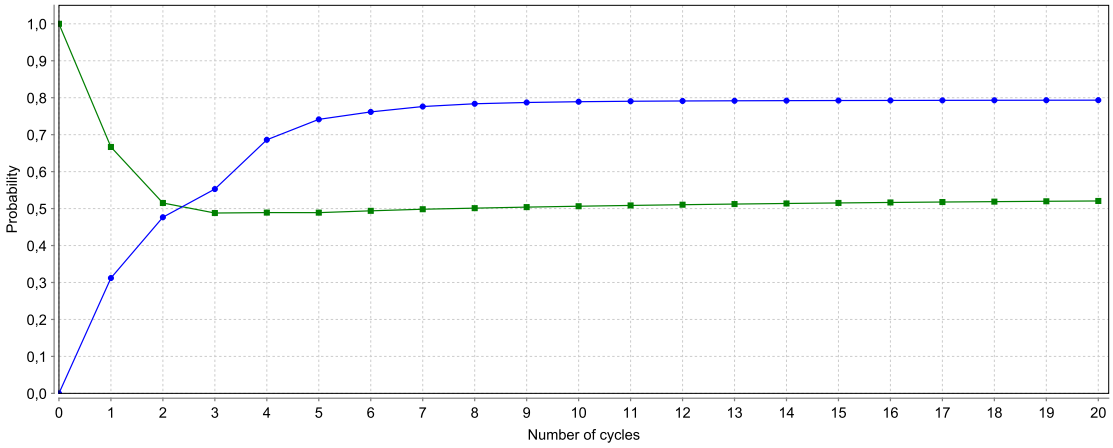

Figure 4. Properties P5 (line with circles) and P6 (line with squares), with evaporation rate 0.3 and 20 cycles.

3.5. Discussion

Our experiments showed that the model successfully fulfills the requirements of the system, considering the specification provided. Even though we used different formulas from the original algorithm, we obtained numerical accurate evidence that the majority of ants (or travellers) tend to take the shortest paths. All our results were obtained via model checking, which guarantees accuracy and confidence. Simulation was only applied to identify how many cycles were necessary to achieve a nearly constant probability for the shortest paths. In this case, simulation was very useful as it was much faster than verification and we only needed an approximate idea of the behaviour of the system during several cycles. However, its use is not a requirement of our technique.

Though we worked with only four cities, it was enough to show that model checking could be applied to the problem and that our abstractions worked as expected. Note that, although the model could be easily extended to a higher number of cities, our intention was simply to evaluate our model-checking process rather than providing a generic solution for the TSP-ACO.

The results of our experiment could be used as a basis for modelling other self-organisation mechanisms, such as other bio-inspired approaches [Mano et al. 2006]. Furthermore, the ant colony system has been used in other interesting problems (e.g., sensor networks [Hong et al. 2008]), which means that our modelling ideas could be applied to the verification of other systems involving this mechanism. Nevertheless, we still need to test the applicability of our modelling ideas in other scenarios. Our current results only show that we could successfully model a complex problem and add to it a self-organisation mechanism to allow emergent behaviours.

4. Related Work

There is not much work on verification of SOES, in particular due to the limitations of model checking regarding the complete state-space exploration. Nonetheless, we can cite a few ideas that have some relation to our work.

In [Gardelli 2008], the authors propose to use formal methods for designing self-organising systems, as well as to use model checking with the PRISM tool to verify emergent properties. Though they apply a very similar idea, their approach is specifically tailored for the multi-agent systems domain, whereas our technique is intended to be generic enough to be applied in any context. The generalisation of their approach is not sufficiently discussed by the authors.

In [Casadei and Viroli 2009], the authors propose a hybrid approach for the verification of emergent properties of self-organising systems. This approach makes use of simulation to approximate the model checking. They use the PRISM tool to perform stochastic simulation and probabilistic model checking of the collective sort problem for distributed tuple spaces. In their approach, given a model and a property to be verified, a large number of simulation runs is executed and the results are evaluated against the property. The final result is the average of all previous results. They apply verification up to the memory limit and then use simulation to provide an approximation of the behaviour of the system considering longer behaviours. During our first experiment, we essentially applied the inverse approach, using simulation to determine at which point the property would reach the expected value and then executing a verification up to that point to improve accuracy. Therefore, they trade accuracy for an extended view of the behaviour of the system, whereas we used approximation as an educated guess and then checked formally that the result was indeed real.

Other approaches use only simulation to verify emergent behaviours of systems. In [De Wolf et al. 2006], an equation-free approach is proposed, which combines numerical analysis algorithms and simulation of individual models. The equation-based model is replaced by small simulations, considering some input parameters. The analysis algorithms guide the simulation process by adjusting the parameters of the system. In [Soares et al. 2008], the authors extend this equation-free approach to verify simulation results for online planners and self-configuration.

5. Conclusions

We presented a verification approach for SOESs. We described a case study considering the TSP with the ACO. This bio-inspired mechanism attributed a self-organising characteristic to the system as it adjusted the probabilities of paths automatically and autonomously. Results of experiments showed that our model, when checked against some quantitative properties, in fact presented the expected behaviour. Though our scenario was quite simple, it was enough to demonstrate that we produced an appropriate modelling of the emergence and self-organising features of the system. From these results it seems that in fact verification can be used for SOESs and provide relevant information for understanding the system behaviour and analysing whether it complies with a specification.

The restrictions we imposed to the scenario were necessary to control the state space and reduce the time for checking properties (in general, it took less than 3 minutes to check a property). We had to apply many simplifications on the original ACO so that it was possible to obtain verification results in a reasonable time. For instance, our choice to apply different formulas from those of the original ACO simplified the calculations, which meant that verification was simpler and faster, but produced similar results. Having results close to those of the original algorithm shows that the essential ideas remained the same, thus resulting in the same behaviour (i.e., most of the ants took the shortest paths). This demonstrate that our abstractions correctly encoded the expected behaviour.

We still have to study how to deal with different situations of self-organisation not supported by our current abstractions, such as the possibility of dynamic modifications on paths. A relevant characteristic of an SOES is that components may enter and leave the system dynamically and it has to adapt to these changes. For example, we could consider a sensor network where individual sensors may run out of energy and, thus, leave the network. At this point the system would have to reorganise itself to cover the area left by this dead sensor so as not to compromise the efficacy of the entire network. Any model of this scenario must include abstractions to describe this dynamic behaviour, which were not considered in this work.

Based on our results, we intend to apply the same mechanism to other known problems where it can be useful (sensor networks, for instance) and also use the same ideas to model other mechanisms. The objective is to investigate the possibility of defining a standard set of abstractions that could be applied to any problem involving an SOES. These abstractions would be part of a verification process to be included in a complete methodology for developing this type of system.

References

Applegate, D., Bixby, R. E., Chvátal, V., and Cook, W. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.

Ben-Ari, M., Manna, Z., and Pnueli, A. (1983). The temporal logic of branching time. *Acta Informatica*, 20(3):207–226.

Casadei, M. and Viroli, M. (2009). Using probabilistic model checking and simulation for designing self-organizing systems. In *SAC*, pages 2103–2104.

Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. The MIT Press, Cambridge, Massachusetts, USA.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001). *Introduction to Algorithms*. The MIT Press.

De Wolf, T. and Holvoet, T. (2005a). Emergence versus self-organisation: Different concepts but promising when combined. In *Engineering Self-Organising Systems: Methodologies and Applications*, volume 3464 of *LNCS*, pages 1–15. Springer-Verlag.

De Wolf, T. and Holvoet, T. (2005b). Towards a methodolgy for engineering self-organising emergent systems. In *SOAS 2005*, volume 135 of *Frontiers in Artificial Intelligence and Applications*, pages 18–34, Glasgow, Scotland. IOS Press.

De Wolf, T., Holvoet, T., and Samaey, G. (2006). Development of self-organising emergent applications with simulation-based numerical analysis. In *Engineering Self-Organising Systems*, volume 3910 of *LNCS*, pages 138–152. Springer.

Dorigo, M. and Gambardella, L. (1997). Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Trans. on Evol. Comp.*, 1(1):53–66.

Duarte, L. M., Kramer, J., and Uchitel, S. (2008). Towards faithful model extraction based on contexts. In *Proceedings of FASE 2008*, volume 4961 of *Lecture Notes in Computer Science*, pages 101–115, Budapest, Hungary. Springer.

Gardelli, L. (2008). *Engineering Self-Organising Systems with the Multiagent Paradigm*. Phd thesis, Alma Mater Studiorum-Università di Bologna, DEIS-Dipartimento di Elettronica, Informatica e Sistemistica.

Gould, R. (1991). Updating the hamiltonian problem - a survey. *Journal of Graph Theory*, 15:121–157.

Hansson, H. and Jonsson, B. (1994). A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535.

Hong, J., Lu, S., and Chen, D. (2008). Towards bio-inspired self-organization in sensor networks: Applying the ant colony algorithm. In *22nd AINA*, pages 1054–1061.

Kwiatkowska, M., Norman, G., and Parker, D. (2002). PRISM: Probabilistic symbolic model checker. In *TOOLS'02*, volume 2324 of *LNCS*, pages 200–204. Springer.

Mano, J.-P., Bourjot, C., Lopardo, G., and Glize, P. (2006). Bio-inspired mechanisms for artificial self-organised systems. *Informatica*, 30:55–62.

Soares, B., Gatti, M., and Lucena, C. (2008). Towards verifying and optimizing self-organizing systems through an autonomic convergence method. In *Proc. of 4th Workshop on Soft. Eng. for Agent-oriented Systems*, pages 73–84, Campinas, Brazil.

Vardi, M. (1985). Automatic verification of probabilistic concurrent finite state programs. In *Proc. of the 26th Annual Symp. on Found. of Comp. Sci.*, pages 327–338.

Wagner, F. R., Duarte, L. M., Heimfarth, T., and Foss, L. e. a. (2009). Uma metodologia de engenharia de software para o desenvolvimento de sistemas emergentes auto-organizáveis. In *II Seminário da SBC sobre Grandes Desafios da Computação no Brasil*.