EAI Endorsed Transactions
on Industrial Networks and Intelligent Systems

Research Article ICST.ORG

Ant Colony Optimization Based Model Checking Extended by Smell-like Pheromone

Tsutomu Kumazawa[1,*], Chihiro Yokoyama[2], Munehiro Takimoto[2], Yasushi Kambayashi[3]

[1]Software Research Associates, Inc, 2-32-8 Minami-Ikebukuro, Toshima-ku, Tokyo 171-8513, Japan
[2]Tokyo University of Science, 2641 Yamazaki, Noda-shi, Chiba 278-8510, Japan
[3]Nippon Institute of Technology, 4-1 Gakuendai, Miyashiro-machi, Minamisaitama-gun, Saitama 345-8501, Japan

Abstract

Model Checking is a technique for automatically checking the model representing software or hardware about whether they satisfy the corresponding specifications. Traditionally, the model checking uses deterministic algorithms, but the deterministic algorithms have a fatal problem. They are consuming too many computer resources. In order to mitigate the problem, an approach based on the Ant Colony Optimization (ACO) was proposed. Instead of performing exhaustive checks on the entire model, the ACO based approach statistically checks a part of the model through movements of ants (ant-like software agents). Thus the ACO based approach not only suppresses resource consumption, but also guides the ants to reach the goals efficiently. The ACO based approach is known to generate shorter counter examples too. This article presents an improvement of the ACO based approach. We employ a technique that further suppresses futile movements of ants while suppressing the resource consumption by introducing a smell-like pheromone. While ACO detects the semi-shortest path to food by putting pheromones on the trails of ants, the smell-like pheromone diffuses differently from the traditional pheromone. In our approach, the smell-like pheromone diffuses from food, and guides ants to the food. Thus our approach not only makes the ants reach the goals farther and more efficiently but also generates much shorter counter examples than those of the traditional approaches. In order to demonstrate the effectiveness of our approach, we have implemented our approach on a practical model checker, and conducted numerical experiments. The experimental results show that our approach is effective for improving execution efficiency and the length of counter examples.

Received on 17 January, 2016; accepted on 11 April, 2016; published on 21 April, 2016

Keywords: Ant Colony Optimization, Model Checking, State Explosion

doi:10.4108/eai.21-4-2016.151156

1. Introduction

Model Checking is a technique that checks whether the design of a software or hardware described as a state transition model satisfies the property specified by the user, which is called a specification [1]. Tools that automatically perform the model checking are called model checkers. General model checkers use a deterministic algorithm for the checking, which often consumes too much machine resources because they exhaustively check their search spaces. In order to mitigate this problem, techniques based on Ant Colony Optimization (ACO) [2] have been proposed [3–6]. They are nondeterministic algorithms. ACO is a swarm intelligence-based method inspired by the behaviors of ants' collecting food, and a multi-agent system that exploits artificial stigmergy (artificial pheromone) for the solution of combinatorial optimization problems. In other words, ACO is a kind of statistic algorithms that is categorized into meta-heuristics [7]. Since ACO only searches a part of entire search space based on the property in which pheromone attracts ants, the ACO based model checking can suppress the use of the computational resources.

One of the major features of the model checker is to present an error trace to its users (called a

*Corresponding author. Email: tkumazawa@acm.org

counter example), which helps the users understand why the model violates the property [8]. Thus, the shorter the presented counter example is, the more comprehensible it is for the users. As an optimization method, ACO based model checking enables model checkers to generate shorter counter examples.

We propose a new ACO-based model checking that consumes less computational resources than the traditional ACO approaches do. Most model checking techniques deal with properties roughly categorized into safety and liveness [9, 10]. Safety properties state that undesirable things never hold, while liveness properties assert that the desirable things finally hold. Our approach handles the safety and focuses on the fact that finding the violation of the safety is reduced to the reachability problem on directed graphs. In other words, the model checking for the safety just searches final (i.e. error) states starting with a specific start state. For safety checking, a path from the start state to a final state is a counter example.

In the ACO based model checking, ants move in the directed graph to look for a final state. Ants probabilistically choose the move direction according to the amounts of pheromone trails on the graph deposited by preceding ants. This behavior of ants helps ants to find relatively short paths to the final states, because the paths that have much pheromone tend be selected by many ants. Thus pheromone trails represent the quality of paths to the final state in terms of path length. In order to further assist ants to search final states, we introduce another special pheromone that attracts ants to the final states. The attraction of the pheromone efficiently guides ants, and localizes the search space, so that our approach can achieve the less consumption of the resources. We can summarize the contributions of our approach to the traditional ACO based approaches as follows:

1. Suppressing memory consumption,
2. Decreasing the time for completing checking, and
3. Shortening obtained counter examples.

The structure of the balance of this article is organized as follows. The next section presents preliminaries of our approach. In the third section, we give a brief explanation of a traditional ACO based approach for safety. After that, in the fourth section, we introduce the smell-like pheromone and the details of our ACO approach that takes advantages of the smell-like pheromone. In the fifth section, we demonstrate the feasibility of our approach, and conclude our discussion in the sixth section.

2. Model Checking

In this section, we give an overview of model checking, and explain the problems that we address.

2.1. Overview

Formal verification is one of the techniques that verify properties of software or hardware. The formal verification techniques can be categorized into two kinds in terms of their approaches. One is logical reasoning approach that represents properties of software as a mathematical theorem such as Hoare logic or predicate calculus. The logical reasoning verifies software using a tool such as theorem provers. It is, however, difficult for the provers to verify software in completely automatic manner, and therefore, they require some interactions with the users.

The other one is Model Checking, which is proposed by Clarke et al [1, 11] and known as one of the most successful research topics in Software Engineering. The model checking was initially applied for the verification of communication protocols [12] and hardware circuits. Model checking is currently used for verifying software without any support from users. Model checking consists of the following steps: 1) representing the model of a system as a state transition system such as an automaton, 2) describing a specification representing required property with temporal logic such as Linear Temporal Logic (LTL) [13] or Computation Tree Logic (CTL) [11], and 3) checking whether the state transition system satisfies the specification. The final step is automatically performed by a model checker. For example, SPIN [10, 14] is one of the most popular model checkers. It checks whether a model described in Promela satisfies the specification described in LTL. Promela is the description language specifically designed for SPIN. At this time, SPIN converts the model and the negation of the specification into Büchi automata, then builds their intersection to exhaustively search for its accepting paths on it. If some accepting paths are found, it means the model does not hold the specification, and the corresponding paths are shown as the counter examples. Otherwise the fact of satisfaction is notified.

In the case of safety checking, the Büchi automaton of the given specification has the property that every path from its start state to its accepting state is accepting. Thus, its accepting states are considered to be final error states. Owing to the property, the accepting states of the intersection are also final error states, to which paths show counter examples. In other words, in order to find counter examples, it is sufficient to search for the accepting states on the intersection.

2.2. Challenges and Related Work

In this article, we tackle with two problems of model checking: *state explosion problem* and *incomprehensive counter examples*.

In model checking, the automata tend to be very large. Actually, the size of an automaton increases exponentially as the size of the corresponding model becomes large. The exponential increase is called state explosion, which may cause exhaustion of resources and may lead to system failure. In order to mitigate the state explosion, techniques such as Partial Order Reduction that decreases the number of states [15], and Bitstate Hashing that reduces the memory area occupied by each state [10] have been proposed. Also, Symbolic Model Checking is known that it does not consume too much memory [16]. They are, however, are symptomatic treatments, and therefore, do not essentially solve the problem.

The other problem is related to the length of counter examples. Because a counter example is presented to the user as a diagnosis of specification violation, it is highly desirable to make it comprehensible for the human user. As mentioned above, a counter example is represented as the path over the directed graph. It means that the counter example is more understandable if it is small enough in lengthwise. That is to say, we need to aim at the model checking methods that generate as short counter examples as possible.

Traditionally, the search techniques adopted by most model checkers are based on Depth First Search (DFS), such as Nested Depth First Search [10]. However, DFS may find long counter examples because there is no guarantee of the shortness of paths that DFS searches.

Recently, parallel algorithms have been studied as promising techniques to improve the performance of model checkers. Holzmann et al. parallelizes Breadth First Search (BFS) to verify safety properties and implements the proposed algorithms on SPIN [17, 18]. Their technique can be used for the detection of the shortest counter examples, because BFS is guaranteed to find shortest counter examples unlike DFS. They also extend the proposed algorithm to handle the subset of liveness properties.

Apart from the improvements for traditional deterministic approaches, some heuristics algorithms have been proposed. They are effective in cases where counter examples are required, although it is difficult for the algorithms to prove that the given model holds the specification because heuristics cannot give exact solutions but quasi-optimal solutions. Edelkamp et al. developed HSF-SPIN [12, 19], which is an extension of SPIN with heuristic search algorithms such as A*, Weighted A*, Iterative A* or Best First Search. The HSF-SPIN gives shorter counter examples and consumes less memory than the deterministic approaches. Groce and Visser proposed some heuristics for model checking of Java programs, based on the specifications to be checked, the structures of the programs, and users' definition [20]. Their evaluation was conducted with DFS, Best First Search, A* and Beam Search.

Recent researches have shown that various meta-heuristic approaches provide more favorable results than traditional deterministic algorithms do in terms of efficiency. Alba et al. showed the applications of Genetic Algorithm (GA) for detection of deadlock, and unnecessary states and transitions [21]. Godefroid et al. proposed a GA based model checking technique [22]. The GA based approaches were the first applications of meta-heuristics for model checking. Alba et al. later showed the effectiveness of Ant Colony Optimization (ACO) for the model checking [3]. We describe the details of the ACO approaches in the next section. Their approach is extended to the checking of safety with Partial Order Reduction [5] and that of liveness [6]. As another work, Francesca et al. proposed a deadlock detection technique using ACO [23]. They utilize the heuristics based on the structures of models to tell ants estimated directions to food. In the paper [24], Ferreira et al. presented a novel technique based on Particle Swarm Optimization (PSO) that finds safety violations. Chicano et al. compared several deterministic and nondeterministic software model checking algorithms using Java PathFinder [25]. Their target algorithms include DFS, BFS, A*, GA, ACO, PSO, Simulated Annealing, Random Search, and Beam Search. They reported that meta-heuristic techniques effectively found safety violations and short counter examples. Note that the work was the first one that applies Simulated Annealing to software model checking.

In addition to those already mentioned, there are some heuristic or meta-heuristic extensions. Staunton et al. proposed the heuristic search algorithms for model checking based on Estimation of Distribution Algorithm [26, 27]. Yousefian et al. proposed a safety checking technique for graph transformation systems based on GA to deal with the state explosion problem. [28]. Rafe et al. proposed a hybrid algorithm combining PSO with Gravitational Search Algorithm in order to find deadlocks in graph transformation systems [29]. Poulding and Feldt extended a variation of Monte-Carlo Tree Search algorithm for heuristic model checking [30].

It is worth noting that meta-heuristic approaches draw a lot of interest from the viewpoint of Search-Based Software Engineering (SBSE) [31]. SBSE refers to a research area of Software Engineering that uses meta-heuristic optimization methodologies. For example, Shousha et al. used GA for the detection of concurrency problems (i.e., starvation and deadlocks) in design models described in Unified Model Language (UML) [32]. Their method does not adopt model

```
procedure ACO
    Initialization
    while terminationCondition do
          ConstructAntSolutions
          UpdatePheromones
    od
end procedure
```

Figure 1. Pseudo code of ACO

checking and aims at presenting a technique that is suitable to UML models. Our approach can be positioned as part of the verification techniques in SBSE.

3. ACO Based Approach

In this section, we give the basics of ACO, and then, describe ACOhg that is a variant of ACO proposed by Alba et al. [3–5] for model checking with large state spaces.

3.1. ACO

ACO is one of the meta-heuristic search algorithms inspired by the behaviors of ants that discover paths from their nest to food. It is known that ACO is effective for some optimization problems such as routing, assigning and scheduling problems. When ACO is applied to model checking problems, a problem is a model expressed in as a directed graph, where the nest and food are the start and error states respectively. The paths between these nodes represent candidate solutions of the model checking problem, i.e. counter examples.

In ACO, agents corresponding to ants cooperatively search the shortest paths through indirect communications using pheromone. The optimal paths found by ants correspond to the optimal solution in optimization problems. The effect of the pheromones is represented as weight on edges, which are assigned to the paths that ants have visited. The pheromones play a role as guides that induce ants to select shorter paths to error states.

The basic model of ACO was given by Ant System (AS) proposed by Dorigo et al. [33]. In most cases, however, AS was not as powerful as other heuristic methods, and needs extensions for practical use. One of the most powerful extensions is Max-Min Ant System (MMAS) [34]. MMAS, which is a kind of iterative algorithms, not only updates pheromones at each iteration step based on the best solution over the previous iterations, but also gives the upper and lower limits to pheromone value. The extension prevents MMAS going into local minimums.

Figure 1 shows how a typical ACO metaheuristic algorithm works. Basically, ACO is the repetition of *ConstructAntSolutions* step for searching paths, and *UpdatePheromones* step for updating the distribution of pheromone. Also, *terminationCondition* is satisfied when some solutions are found or the number of the repetition exceeds a fixed number. In the following, we discuss each step in more detail.

Initialization step initializes pheromones on edges for preparation of searching by ants. While regarding the start state and final states as the nest and food respectively, pheromones with random strength are randomly located between these states.

ConstructAntSolutions step makes each ant probabilistically transit states starting from the start state to find candidate solutions. At this time, a transition to the next state is selected using the following equation:

$$p_{ij}^{k} = \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\Sigma_{l \in N_i}[\tau_{il}]^{\alpha}[\eta_{il}]^{\beta}}, \quad if \ j \in N_i \tag{1}$$

In the equation, N_i is the set of states to which an ant can move from state i; j is the destination state of one step movement of the ant; τ_{ij} is the value of pheromone on the edge (i, j); and η is heuristic value representing the number of transitions to a final state, which is set to a smaller value than actual one. The heuristic value is estimated based on the locations of final states, or the length and property of a specification that are described in LTL. Also, α and β are empirically determined values in order to adjust the effects of the heuristic value and pheromone value, respectively. When ants have moved on edges in the fixed number of times, or some candidates of solutions are found, *UpdatePheromones* step follows.

UpdatePheromones step updates the pheromone value on each edge. The pheromone strengths on selected edges are increased according to the appropriateness of the edges. At the same time, the pheromones on the other edges are updated as follows:

$$\tau_{ij} \leftarrow (1 - x_i)\tau_{ij} \tag{2}$$

In the equation, x_i is set to a value between 0 and 1. This is the value representing the degree of evaporation of pheromones. Through the update process, the priorities of previously selected edges are decreased step by step until some ants select the edges again.

3.2. ACOhg

It is difficult for traditional ACOs to handle state transition systems for model checking that have thousands of nodes. Because there can be billions of edges in such systems, they require a number of megabytes in a memory to record pheromone values. Especially, the size of a model of concurrent system is known to be huge. For example, the size of the model of Dining Philosophers with n philosophers is 3^n, which

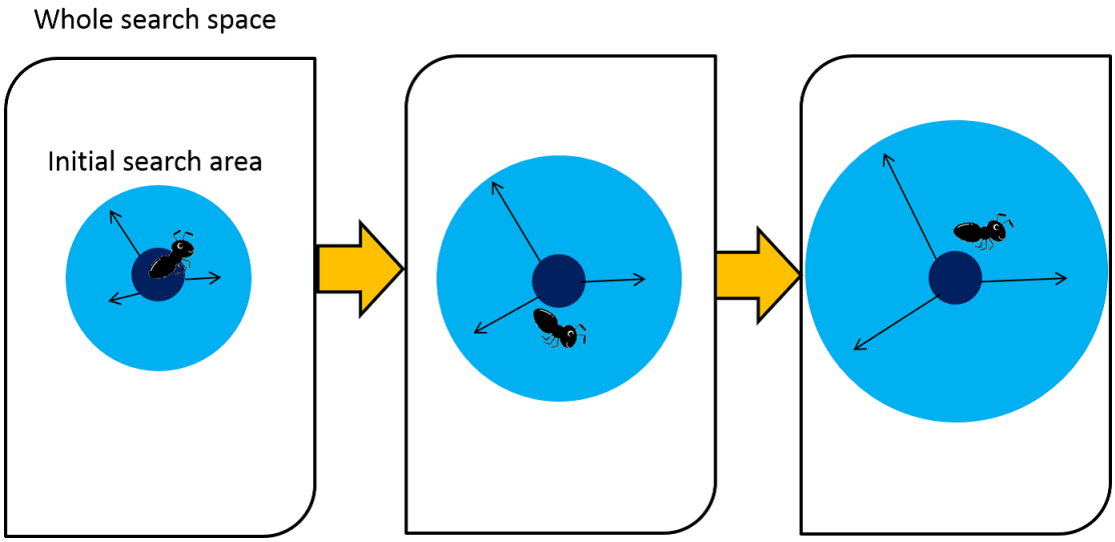


Figure 2. Expansion Technique

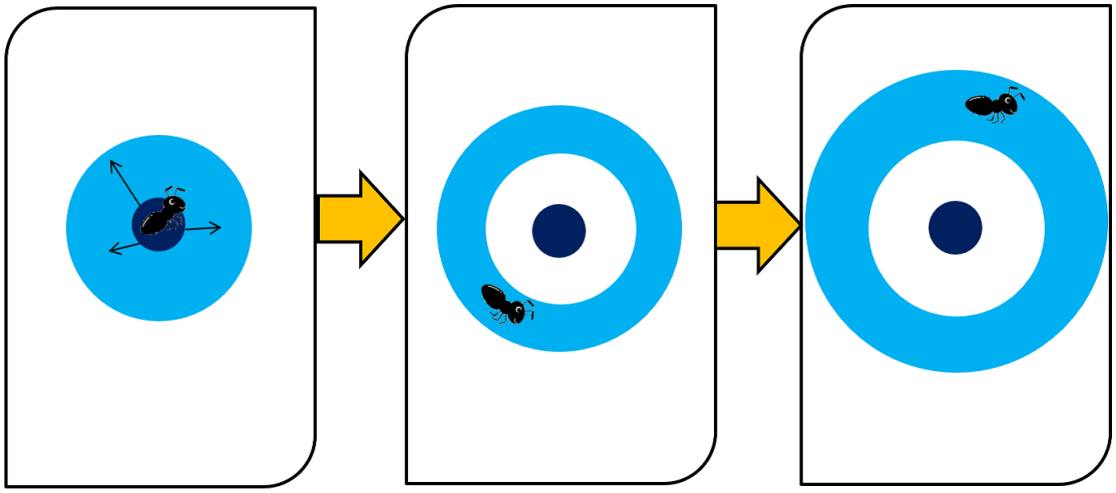


Figure 3. Missionary Technique

increases exponentially. The simple solutions such as prohibiting revisits to the same states are not effective, because some ants may run into brick walls, or may wander from state to state for a long time even if they finally reach the final states. Thus, the behaviors of the traditional ACO in model checking may result in the state explosion without finding any candidates of solutions. In addition, as a more fatal problem, the traditional ACO has to initially set pheromone values to all the transitions. This may also consume too much memory.

In order to mitigate the problems of the traditional ACO, Alba et al. proposed *ACO for huge graphs* (ACOhg), which can perform searching with less memory than the traditional ones [3, 4]. The basic idea of ACOhg is to give the upper limit λ_{ant} to the number of move steps of ants at one stage. This search manner suppresses the time and memory consumption, but limiting of move steps may prevent ants from reaching the final states. That is, λ_{ant} has to be decided so as to find the final states. ACOhg gives two kinds of techniques for determining λ_{ant}; they are *expansion technique* and *missionary technique*.

In the expansion technique, once the system cannot find a final state in the search for the current λ_{ant}, λ_{ant} is increased by the value given as a parameter, and then the system searches the wider area defined by the new λ_{ant} again as shown in Figure 2. The process starts with small λ_{ant} and repeats until it finds some final states. The expansion technique increases λ_{ant} just enough, so that the memory consumption can be suppressed. Also, it is easy to implement because it is a simple extension of the traditional ACO. On the other hand,

its behavior becomes closer to the traditional ACO's as λ_{ant} increases.

The missionary technique is similar to the expansion technique, but searches are performed from not the start state but some states on the edge of the previously searched area, i.e. ignoring old pheromone as shown in Figure 3. The new start state on the edge is selected from the states that ants reach at the previous stage. This search manner enables ACO to gradually extend the search area without changing λ_{ant}, so that it only requires constant time and memory consumption at each stage.

Both approaches decide the strength of pheromones to be assigned based on a *fitness function*. The fitness function returns the degree of penalty for the trail of each ant, which becomes very large in the case where the trail includes some cycles, or no final state. Based on the fitness function f, a^{best} with the lowest penalty $f(a^{best})$ is determined, and then, the pheromone of its trace is strengthened as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + \frac{1}{f(a^{best})}, \forall (i,j) \in a^{best} \tag{3}$$

ACOhg, which is based on MMAS, also uses the fitness function for calculating the limit values τ_{max} and τ_{min} of the pheromone value for each trail as follows:

$$\tau_{max} = \frac{1}{\rho f(a^{best})}, \tag{4}$$

$$\tau_{min} = \frac{\tau_{max}}{a} \tag{5}$$

In the equation, ρ and a are constants that control the range of the pheromone values. Also, the missionary approach uses the fitness function to decide the next start states at each stage.

4. Extended ACOhg

We extend ACOhg by introducing a new kind of pheromones in order to suppress futile movement of ants. We call the extended ACOhg *EACOhg*. This section presents the outline of EACOhg, and then describes the details of the new pheromone guiding ants to the final states.

4.1. Outline of EACOhg

ACOhg can suppress the analysis time and memory consumption, but it searches from the start state in any directions like the traditional ACO. If there is information about the direction, the search area can be localized. We extend ACOhg so as to use the direction information to find the final states. In the real world, the direction information corresponds to smell diffused from food. Indeed, ants can recognize not only pheromones, but also the smell, which is important information to reach unfound food in the early stage.

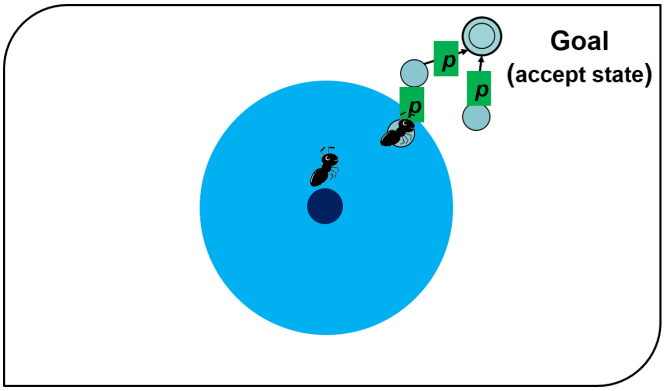


Figure 4. Diffusing of smell

1: $\tau \leftarrow initialize\_pheromone()$;
2: $\gamma \leftarrow \{ node \mid node \in F \}$;
3: **while** $step \leq msteps \wedge \nexists i \in [1..colsize] a^i_* \in F$ **do**
4: **for** $k = 1$ to $colsize$ **do**
5: $a^k \leftarrow \emptyset$;
6: **while** $|a^k| \leq \lambda_{ant} \wedge T(a^k_*) - a^k \neq \emptyset \wedge a^k_* \notin F$ **do**
7: $node \leftarrow select\_successor(a^k_*, T(a^k_*), \tau, \gamma, \eta)$;
8: $a^k \leftarrow a^k + node$;
9: $\tau \leftarrow local\_pheromone\_update(\tau, \xi, (a^k_*, node))$;
10: **end while**
11: **if** $f(a^k) < f(a^{best})$ **then**
12: $a^{best} \leftarrow a^k$;
13: **end if**
14: **end for**
15: $\tau \leftarrow pheromone\_evaporation(\tau, \rho)$;
16: $\tau \leftarrow pheromone\_update(\tau, a^{best})$;
17: $\gamma \leftarrow scatter\_goal\_pheromone(\gamma)$;
18: $\lambda_{ant} \leftarrow \lambda_{ant} + \sigma$;
19: **end while**

Figure 5. Algorithm of EACOhg

In our model, we regard the smell as another kind of pheromones. We call this *goal pheromone*. We made the goal pheromone stronger than the normal pheromone. Once the goal pheromones are put on transition edges, the edges are selected in preference to the edges with the normal pheromone as shown in Figure 4. The search manner localizes the search area further, contributing to finding the final states more quickly and generating shorter counter examples, although memory consumption may increase a little to hold the goal pheromones.

Figure 5 shows the pseudo code of EACOhg, where the kth ant is represented by a^k, and a path where a^k traversed is represented by $|a^k|$. Also, a^k_j and a^k_* represent the jth node and the last node on the path

respectively. $T(a_*^k)$ represents the set of nodes to which a^k can transit from a_*^k.

The EACOhg starts with randomly assigned pheromone value in $[0-1.0]$ to all the edges as shown in line 1. After that, the search step, which consists of movement of ants and update of pheromones, is repeated until the number of repetitions exceeds upper limit *msteps* or some ants reach the final states F in the loop body in lines 3–19.

Each stage performs a movement within λ_{ant} and the movement causes pheromone update. The destination *node* to which to move is probabilistically selected based on the strength of the pheromone value τ through function *select_successor* in line 7. This node is appended to a^k. At this time, the pheromone on the selected edge is enhanced through function *local_pheromone_update* in line 9. Thus, the best path based on the fitness function f in the stage is always held in a^{best}.

Once the operation in the current stage is completed, pheromone values are globally updated for the effect of evaporation and enhancing pheromones on a^{best} through the functions *pheromone_evaporation* and *pheromone_update* respectively in lines 15 and 16. Furthermore, in our algorithm, the goal pheromone is diffused through the function *scatter_goal_pheromone*. Finally, in order to search a wider area in the case where no ant reaches the final states. λ_{ant} is increased in line 18.

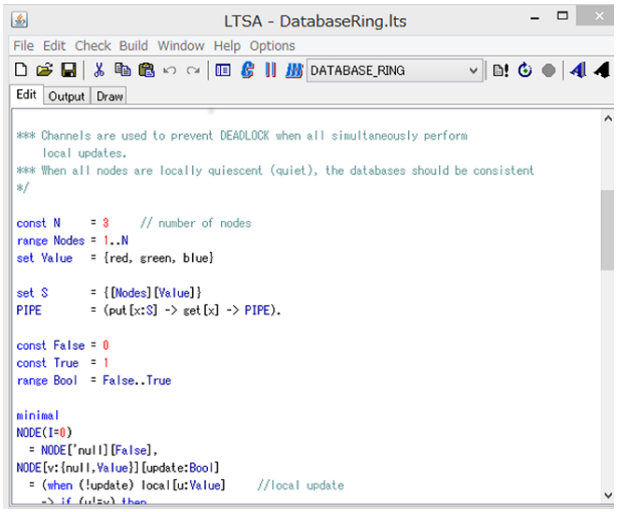
4.2. Goal Pheromone

The existence of the goal pheromone shows that there are some paths from the current state to the final states. Therefore, it is enhanced in order to attract ants more strongly through the following properties:

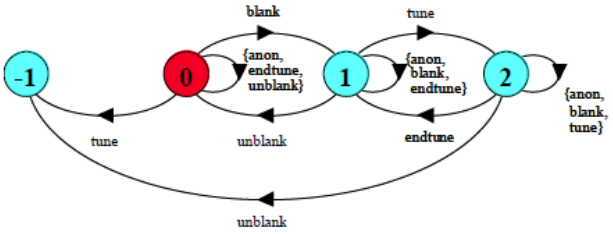
1. Attracting ants more strongly than normal pheromone,
2. Non-volatility, and
3. Defusing from the final states to their peripheries.

Note that the above properties induce ants to select relatively short paths to the final states, which leads to short counter examples.

The normal pheromones behave along with MMAS and hence, have the value less than or equal to τ_{max}. The goal pheromone is set to the value more than τ_{max} in order to attract ants more strongly than normal pheromones. Moreover, while the normal pheromone evaporates, the goal pheromone does not evaporate and has a constant value. It is derived from the fact that the smell is always supplied by the source, i.e. food. Thus, once an ant finds some goal pheromones, it moves to the edges with them with high probability. The behavior of an ant is implemented by *select_successor*.



(a) Input model



(b) Generated model

Figure 6. Models of LTSA

The goal pheromone is assigned on in-edges of the current states by *scatter_goal_pheromone* in order to make them defuse. At that time, the pheromone is scattered on only some edges that are randomly selected. The scatter manner contributes to suppressing time and memory consumption.

5. Experimental Results

In order to demonstrate the effectiveness of our approach, we have implemented our extended ACO, *EACOhg*, on a practical model checker called LTSA (Labeled Transition System Analyzer) [9, 35]. LTSA is one of the model checkers that can be customized easily, and can handle models that are suitable for our purpose. LTSA supports the checking of Fluent Linear Temporal Logic [35], which is a kind of LTL specialized for event-based systems described as Labeled Transition Systems. For example, we may give a model with several final states as an input of LTSA as shown in Figure 6(a), while LTSA generates models with a single final state, which is an error state corresponding to the final states for checking safety, as shown in Figure 6 (b). The property of single final state of the models makes handling of models easy. In Figure 6 (b), the red state labeled 0 is the initial state, and the state

Table 1. Settings of coefficients

coefficients	values
mstep	100
colsize	10
α	1.0
β	2.0
η	1.0
ρ	0.2
a	5
P_p	70
P_c	70

P_p and P_c are weights of penalties used by the fitness function for no final state and the path with some cycles.

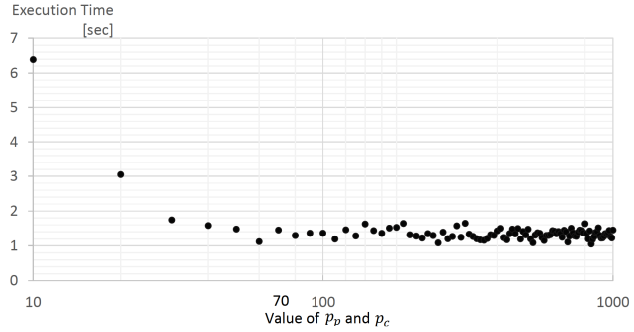


Figure 7. Preliminary execution time on small model

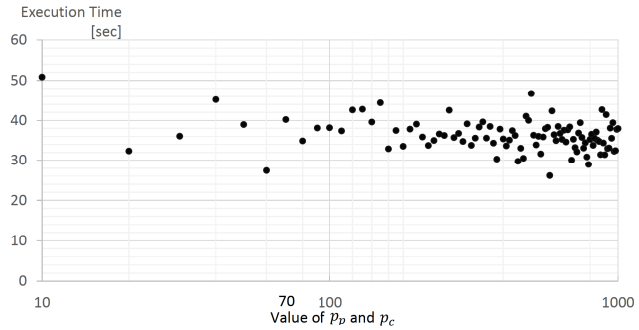


Figure 8. Preliminary execution time on middle model

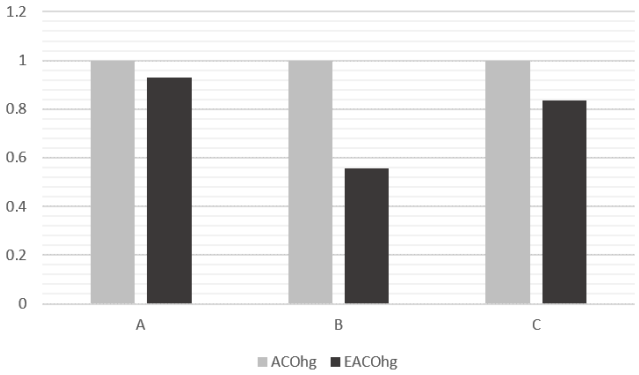


Figure 9. Execution time

−1 is the single final state. LTSA generates the models as directed graphs in Aldebaran form, for which we have extended the checking phase. Also, in the phase, we have implemented goal pheromones in accordance with the description Section 4.2, for which we have chosen random values in tenfold range i.e. [0.0-10.0] so that goal pheromones have much more influence than normal pheromones. Notice that the random setting of each goal pheromone contributes to mitigating local minimum problem as well as normal pheromones.

We used values as shown in Table 1 for the coefficients and constants that appear in equations and algorithms. Most of these values can be basically the same values as ACOhg because EACOhg is a simple extension of ACOhg. However, the values of P_p and P_c have to be empirically decided.

We decided the best settings for P_p and P_c through preliminary experiments. In the preliminary experiments, we generated two models with small and middle sizes based on dinning philosophers problem, checking them for dead lock safety while changing P_p and P_c from 10 to 1000. Figures 7 and 8 show execution time for each experiment, where the small and middle models have 213 and 7,773 states respectively. Note that the figures are semi-logarithmic plots. As shown in the figures, the execution time decreases at first as they increase, regardless of the size of models. Once P_p and P_c reach 70, the execution time does not decrease. Thus, we decided to adopt 70 as P_p and P_c in main experiments.

In the main experiments, we prepared three models: model A with 3843 states, model B with 31747 states, and model C with 266,218 states adjusting the parameters of two kinds of examples *Mutex_fluent* and *DatabaseRing*. We conducted three experiments in terms of execution time, the length of counter examples, and memory consumption. In each experiment, we compared our approach with ACOhg for models A, B and C, where we show the average of the results of one hundred times applications for each model.

As shown in Figure 9, our approach is more efficient than ACOhg. It shows that goal pheromones guide ants to the final states, suppressing the chances going out their ways. Furthermore, Figure 10 shows that our approach generates much shorter counter examples than ACOhg. The fact also substantiates the efficiency of our approach by the shorter trails of ants.

On the other hand, our approach consumes more memory than ACOhg as shown in Figure 11. The increase of the memory consumption fell within 10% at most, and was 5% in average, though. Considering that our approach have achieved 23% more efficient execution and 50% shorter counter examples in their averages, we can say that the increase is negligible.

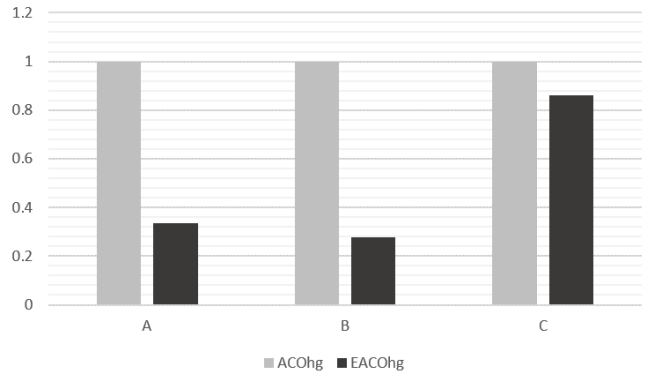


Figure 10. The length of counter examples

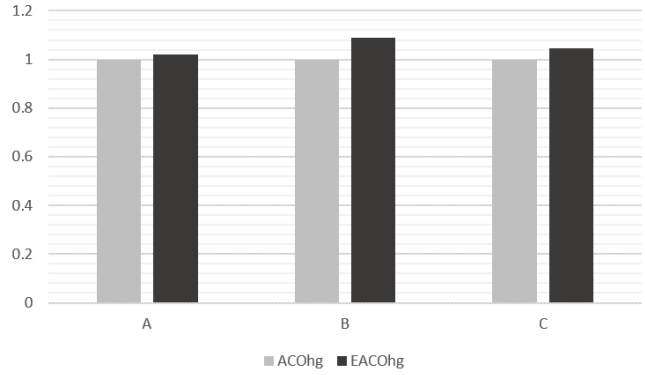


Figure 11. Memory consumption

6. Conclusions and Future Directions

We have proposed a novel ACO based model checking, i.e. EACOhg that further suppresses futile movements of ants while suppressing the resource consumption by introducing smell-like pheromone. The smell-like pheromone diffuses from goals and guides ants to the goals. Thus our approach not only makes the ants reach the goals further more efficiently but also generates much shorter counter examples than the existing best ACO based approach, ACOhg.

In order to show the effectiveness of our approach, we implemented the EACOhg on a practical model checker, and conducted experiments. The results of the experiments show that our approach is effective for improving execution efficiency and the length of counter examples. Future work involves the enhancement of EACOhg and its evaluation based on more realistic case studies.

We have observed that our approach does not achieve effectiveness in the very large model (model C) as shown in Figures 9 and 10. The reason may be that the effect of the smell-like pheromone is compromised in a large model. In order to overcome this problem, we plan to add directionality in diffusing the smell-like pheromone.

We are pursuing further improvements of EACOhg to make it more practical and complete. We envisage that the improvements should be achieved by the combination of EACOhg with heuristics and the optimization methods of classical model checking. Some researchers showed that appropriate heuristics can improve the efficiency of model checking in the wide range of problem domains [12, 19, 20, 23]. Most heuristics are problem-specific. They depend on the specifications to be checked and the structures of the target system models. Therefore we have to carefully investigate which heuristic methods contribute to the performance improvements of EACOhg.

Traditional model checking researches have proposed various kinds of optimization techniques as noted in Section 2. Although most of the methods aim at optimizing deterministic search, EACOhg can be applicable to some of them such as Partial Order Reduction (POR) [5]. In addition, we plan to extend EACOhg to the checking of liveness, because EACOhg only targets on the safety checking and does not support the check the properties such as Response and Existence [36]. In the case of liveness checking, counter examples may have infinite length and contains cycles. Thus, we have to investigate the effects of goal pheromones on finding paths with cycles. One approach is proposed by Chicano and Alba [6]. As the next phase, we plan to apply our method to safety checking with POR and liveness checking so as to demonstrate the applicability of our approach.

Finally and the most importantly, we have to apply EACOhg to practical models of software systems or communication protocols to evaluate its effectiveness empirically. For example, the recent work by Chicano et al. [25] evaluates several meta-heuristic model checking techniques using communication protocols. Similarly, Shousha et al. [32] uses models of a bank fund transfer problem and a cruise control problem. EACOhg is not built on the specific programming or modeling languages, hence it is possible to adapt to the real world systems. A candidate data set for our empirical research is the Software-artifact Infrastructure Repository [37], which is used for evaluation by Poulding and Feldt [30].

Acknowledgements. This work is supported in part by Japan Society for Promotion of Science (JSPS), with the basic research program (C) (No. 25330089 and 26350456), Grant-in-Aid for Scientific Research. We have received the support and suggestions from Mr. Keiichiro Takada in Department of Information Sciences, Tokyo University of Science.

References

[1] Clarke, Jr., E.M., Grumberg, O. and Peled, D.A. (1999) *Model Checking* (MIT Press).

[2] Dorigo, M. and Stützle, T. (2004) *Ant Colony Optimization* (Bradford Company, MIT Press).

[3] Alba, E. and Chicano, F. (2007) Finding safety errors with aco. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07: 1066–1073. doi:10.1145/1276958.1277171.

[4] Alba, E. and Chicano, F. (2007) Acohg: Dealing with huge graphs. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07: 10–17. doi:10.1145/1276958.1276961.

[5] Chicano, F. and Alba, E. (2008) Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models. *Information Processing Letters* **106**(6): 221–231. doi:10.1016/j.ipl.2007.11.015.

[6] Chicano, F. and Alba, E. (2008) Finding liveness errors with ACO. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC*: 2997–3004. doi:10.1109/CEC.2008.4631202.

[7] Boussaïd, I., Lepagnot, J. and Siarry, P. (2013) A survey on optimization metaheuristics. *Inf. Sci.* **237**: 82–117. doi:10.1016/j.ins.2013.02.041.

[8] Clarke, E.M. (2008) The birth of model checking. In Grumberg, O. and Veith, H. [eds.] *25 Years of Model Checking - History, Achievements, Perspectives*, 1–26. doi:10.1007/978-3-540-69850-0_1.

[9] Magee, J. and Kramer, J. (1999) *Concurrency: State Models & Java Programs* (John Wiley & Sons, Inc.).

[10] Holzmann, G. (2004) *The Spin Model Checker: Primer and Reference Manual* (Addison-Wesley).

[11] Clarke, E.M. and Emerson, E.A. (1982) Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*: 52–71. doi:10.1007/BFb0025774.

[12] Edelkamp, S., Leue, S. and Lluch-Lafuente, A. (2004) Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer* **5**(2-3): 247–267. doi:10.1007/s10009-002-0104-3.

[13] Manna, Z. and Pnueli, A. (1992) *The Temporal Logic of Reactive and Concurrent Systems: Specification* (Springer-Verlag New York, Inc.).

[14] Holzmann, G.J. (1997) The model checker spin. *IEEE Transactions on Software Engineering* **23**(5): 279–295. doi:10.1109/32.588521.

[15] Lluch-Lafuente, A., Edelkamp, S. and Leue, S. (2002) Partial order reduction in directed model checking. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software* (Springer-Verlag): 112–127. doi:10.1007/3-540-46017-9_10.

[16] Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L. and Dill, D.L. (1994) Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **13**(4): 401–424.

[17] Holzmann, G.J. (2012) Parallelizing the spin model checker. In *Proceedings of the 19th International Conference on Model Checking Software*, SPIN'12: 155–171. doi:10.1007/978-3-642-31759-0_12.

[18] Filippidis, I. and Holzmann, G.J. (2014) An improvement of the piggyback algorithm for parallel model checking. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014: 48–57. doi:10.1145/2632362.2632375.

[19] Edelkamp, S., Lafuente, A.L. and Leue, S. (2001) Directed explicit model checking with hsf-spin. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01: 57–79. doi:10.1007/3-540-45139-0_5.

[20] Groce, A. and Visser, W. (2004) Heuristics for model checking java programs. *International Journal on Software Tools for Technology Transfer* **6**(4): 260–276. doi:10.1007/s10009-003-0130-9.

[21] Alba, E. and Troya, J.M. (1996) Genetic algorithms for protocol validation. In *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, PPSN IV: 870–879. doi:10.1007/3-540-61723-X_1050.

[22] Godefroid, P. and Khurshid, S. (2004) Exploring very large state spaces using genetic algorithms. *Int. J. Softw. Tools Technol. Transf.* **6**(2): 117–127. doi:10.1007/s10009-004-0141-1.

[23] Francesca, G., Santone, A., Vaglini, G. and Villani, M.L. (2011) Ant colony optimization for deadlock detection in concurrent systems. In *Proceedings of IEEE 35th Annual Computer Software and Applications Conference* (IEEE Computer Society): 108–117. doi:10.1109/COMPSAC.2011.22.

[24] Ferreira, M., Chicano, F., Alba, E. and Gómez-Pulido, J. (2008) Detecting protocol errors using particle swarm optimization with java pathfinder. In *Proceedings of the High Performance Computing & Simulation Conference*, HPCS '08: 319–325.

[25] Chicano, F., Ferreira, M. and Alba, E. (2011) Comparing metaheuristic algorithms for error detection in java programs. In *Proceedings of the Third International Conference on Search Based Software Engineering*, SSBSE'11 (Springer-Verlag): 82–96. doi:10.1007/978-3-642-23716-4_11.

[26] Staunton, J. and Clark, J.A. (2010) Searching for safety violations using estimation of distribution algorithms. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10: 212–221. doi:10.1109/ICSTW.2010.24.

[27] Staunton, J. and Clark, J.A. (2011) Finding short counterexamples in promela models using estimation of distribution algorithms. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11: 1923–1930. doi:10.1145/2001576.2001834.

[28] Yousefian, R., Rafe, V. and Rahmani, M. (2014) A heuristic solution for model checking graph transformation systems. *Applied Soft Computing* **24**: 169–180. doi:10.1016/j.asoc.2014.06.055.

[29] Rafe, V., Moradi, M., Yousefian, R. and Nikanjam, A. (2015) A meta-heuristic solution for automated refutation of complex software systems specified through graph transformations. *Applied Soft Computing* **33**(C): 136–149. doi:10.1016/j.asoc.2015.04.032.

[30] Poulding, S. and Feldt, R. (2015) Heuristic model checking using a monte-carlo tree search algorithm. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15: 1359–1366. doi:10.1145/2739480.2754767.

[31] Harman, M., Mansouri, S.A. and Zhang, Y. (2012) Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.* **45**(1): 11:1–11:61. doi:10.1145/2379776.2379787.

[32] Shousha, M., Briand, L. and Labiche, Y. (2012) A uml/marte model analysis method for uncovering scenarios leading to starvation and deadlocks in concurrent systems. *IEEE Trans. Softw. Eng.* **38**(2): 354–374. doi:10.1109/TSE.2010.107.

[33] Dorigo, M., Maniezzo, V. and Colorni, A. (1996) The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on SYSTEMS,Systems, Man, and Cybernetics, Part B: Cybernetics* **26**(1): 29–41. doi:10.1109/3477.484436.

[34] Stützle, T. and Hoos, H.H. (2000) Max-min ant system. *Future Generation Computer System* **16**(9): 889–914.

[35] Giannakopoulou, D. and Magee, J. (2003) Fluent model checking for event-based systems. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11: 257–266. doi:10.1145/940071.940106.

[36] Dwyer, M.B., Avrunin, G.S. and Corbett, J.C. (1999) Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99 (ACM): 411–420. doi:10.1145/302405.302672.

[37] Do, H., Elbaum, S. and Rothermel, G. (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.* **10**(4): 405–435. doi:10.1007/s10664-005-3861-2.