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

Tsutomu Kumazawa[*]
Software Research Associates, Inc.
2-32-8 Minami-Ikebukuro, Toshima-ku
Tokyo 171-8513, Japan
tkumazawa@acm.org

Chihiro Yokoyama
Tokyo University of Science
2641 Yamazaki, Noda-shi
Chiba 278-8510, Japan
c-yokoyama@cs.is.noda.tus.ac.jp

Munehiro Takimoto
Tokyo University of Science
2641 Yamazaki, Noda-shi
Chiba 278-8510, Japan
mune@cs.is.noda.tus.ac.jp

Yasushi Kambayashi
Nippon Institute of Technology
4-1 Gakuendai, Miyashiro-machi,
Minamisaitama-gun
Saitama 345-8501, Japan
yasushi@nit.ac.jp

ABSTRACT

Model Checking is a technique for automatically checking whether the model representing software or hardware satisfies the corresponding specification. Traditionally, the model checking uses deterministic algorithms, but the deterministic algorithms have a fatal problem of consuming too many computer resources. In order to mitigate the problem, the approach based on 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. 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 paper 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. Essentially, ACO detects the semi-shortest path to food by putting pheromones on the trails of ants, but the smell-like pheromone diffuses like smell of food. In our approach, the diffusing smell-like pheromone attracts ants to the food, so that our approach not only makes the ants reach the goals farther and more efficiently but also generate 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 remarkably effective for improving execution efficiency and the length of counter examples.

[*]Corresponding author - tkumazawa@acm.org

.

General Terms

Static Analysis, Swarm Intelligence, Software Engineering

Keywords

Model Checking, Ant Colony Optimization, State Explosion

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 [8]. The tool that automatically performs the model checking is called a model checker. 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 the problem, techniques based on Ant Colony Optimization (ACO) [10] that is non-deterministic algorithm have been proposed [1, 4, 5]. 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. ACO is a kind of statistic algorithms that is categorized into meta-heuristics. Since ACO only searches a part of the whole 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 model checkers is to present an error trace to its users (called a counter example), which helps the users understand why the model violates the property [6]. 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 use of resources than the traditional ACO approaches. Most model checking deals with properties roughly

BICT 2015, December 03-05, New York City, United States
Copyright © 2016 ICST
DOI 10.4108/eai.3-12-2015.2262408

categorized into safety and liveness [17, 14]. 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 order to assist ants to search final states, we introduce another kind of 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 achieves 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 rest of the paper 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. In the fifth section, we demonstrate the feasibility of our approach, and conclude our discussion in the sixth section.

2. BACKGROUNDS

Formal verification is one of techniques that verify properties of software or hardware. The formal verification techniques can be categorized into two kinds of approaches. One is logical reasoning 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 difficult for the provers to verify software in completely automatic, and therefore, they require some interactions with the users.

The other one is Model Checking, which is proposed by Clarke et al [7, 8]. 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) [18] or Computation Tree Logic (CTL) [7], and 3) checking whether the state transition system satisfies the specification. The final step is automatically performed by a model checker. For example, SPIN [15, 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 a 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 counter examples are shown as the corresponding paths. Otherwise the fact of satisfaction is notified. In case of checking safety, because the final states of the intersection are error states, paths to such states correspond to the accepting paths.

In Model Checking, the automata tend to be very large. Actually the size of the automaton increases exponentially as the size of the corresponding model becomes extremely large. The exponential increase is called *state explosion*, which may cause insufficient 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 [16], and Bitstate Hashing that reduces the memory area occupied by each state [14] have been proposed. Also, Symbolic Model Checking is known that it does not consume too much memory [3]. However, they are symptomatic treatments, and therefore, do not essentially solve the problem.

Apart from the improvements for traditional deterministic approaches, some heuristics algorithms have been proposed. They are effective in the 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, 11], 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. Alba et al. showed the applications of Genetic Algorithm (GA) for detection of deadlock, and unnecessary states and transitions [2]. The GA based approach was the first application of meta-heuristics for a model checking. Alba et al. also showed the effectiveness of Ant Colony Optimization (ACO) for the model checking [1]. 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 [4] and that of liveness [5]. More recently, Staunton et al. proposed the heuristic search algorithms for model checking based on Estimation of Distribution Algorithm [19, 20].

3. ACO BASED APPROACH

In this section, we give the basis of ACO, and then, describe ACOhg that is a variant of ACO proposed by E. Alba et al. for model checking.

3.1 ACO

ACO is one of the meta-heuristic search algorithms inspired by the behaviors of ants that search paths to food. It is known that ACO is effective for some optimization problems such as routing, assigning and scheduling problems. In ACO, agents corresponding to ants cooperatively search optimal paths through indirect communications using pheromone. The effect of the pheromones is represented as weight on edges, which are assigned to the paths that ants have visited.

The basic model of ACO was given by Ant System (AS) proposed by Dorigo et al [9]. 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) [21]. 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

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

Figure 1: Pseudo code of ACO

into local minimums.

Figure 1 shows a typical ACO algorithm. Basically, ACO is the repetition of *ConstructAntSolutions* for searching paths, and *UpdatePheromones* for updating the distribution of pheromone. Also, the termination condition is satisfied when some solutions are detected or the number of the repetition exceeds a fixed number. In the initialization step, while regarding the start state and final states as the nest and food respectively, pheromone with random strength is randomly located between these states. After that, each ant probabilistically transits states starting from the start state in *ConstructAntSolutions* step. At this time, a transition to the next state is selected using the following equation:

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

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 the 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 as described in LTL. Also, α and β are empirically decided value in order to adjust the effects of the heuristic value and pheromone value, respectively.

Once ants move edges in fixed times, or some candidates of solutions are found, *UpdatePheromones* step follows to update pheromone value on each edge. In the step, 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 an ant selects the edges again.

3.2 ACOhg

It is difficult for traditional ACOs to handle the state transition system for model checking that has thousands of nodes. Because there can be billions of edges in the system, and they require a number of megabytes for 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 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

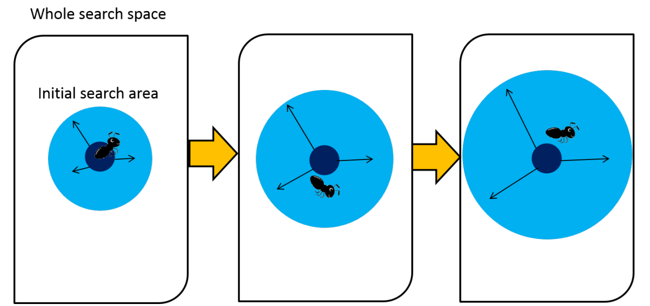


Figure 2: Expansion Technique

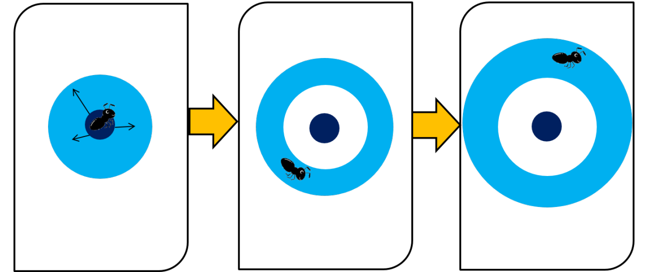


Figure 3: Missionary Technique

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. Also, as a more fatal problem, the traditional ACO has to initially set pheromone values to all the transitions. This may consume too much memory too.

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 one [1]. 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 final states. That is, λ_{ant} has to be decided so as to find the final states. ACOhg gives two kinds of techniques for deciding 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, but therefore, 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

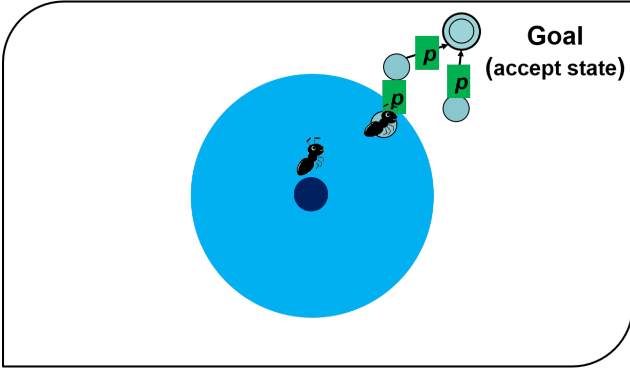


Figure 4: Diffusing of smell

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 *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 decided, 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 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 new pheromone 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.

In our model, we regard the smell as another kind of pheromones. We call this *goal pheromone*. We made the goal pheromone be stronger than the normal pheromone. Once the goal pheromone is 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 final states more quickly and generating shorter counter examples, although memory consumption may increase a little to hold the goal pheromones.

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

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 a 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 consisting of movement of ants and update of pheromones, which is repeated until the number of steps 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 fitness function f in the stage is always held in a^{best}.

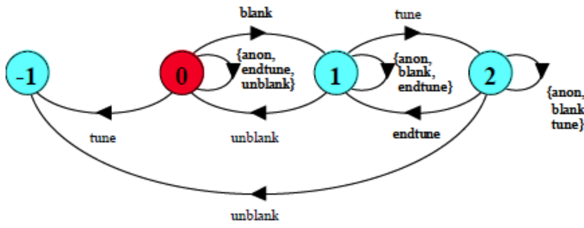
Once the operation in the current stage is completed, pheromone values are globally updated for the effect of evaporation and enhancing pheromones on a^{best} through functions *pheromone_evaporation* and *pheromone_update* respectively in lines 15 and 16. Furthermore, in our algorithm, goal pheromone is diffused through function *scatter_goal_pheromone*. Finally, in order to search the 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 a goal pheromone shows that there are

```
LTSA - DatabaseRing.lts
File Edit Check Build Window Help Options
DATABASE_RING
Edit Output Draw

*** Channels are used to prevent DEADLOCK when all simultaneously perform
    local updates.
*** When all nodes are locally quiescent (quiet), the databases should be consistent
*/

const N      = 3     // number of nodes
range Nodes  = 1..N
set Value    = {red, green, blue}

set S        = {[Nodes][Value]}
PIPE         = (put[x:S] -> get[x] -> PIPE).

const False = 0
const True  = 1
range Bool  = False..True

minimal
NODE(I=0)
  = NODE['null][False],
NODE[v:{null,Value}][update:Bool]
  = (when (!update) local[u:Value]     //local update
```

(a) Input model



(b) Generated model

Figure 6: Models of LTSA

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 shorter paths to the final states, which leads to shorter 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. Also, the normal pheromone evaporates, while the goal pheromone has a constant value. It is derived from the fact that the smell is always supplied by the food that is the source of it. 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*.

The goal pheromone is assigned on in-edges of the current states by *scatter_goal_pheromone* in order to make them defuse. At this time, 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 practical model checker LTSA (Labeled Transition System Analyzer) [17, 13]. LTSA is one of the model checkers that can be customized easily, and can handle models that are suitable for our purpose. 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 error state corresponding to the final states for checking safety, as shown in Figure 6 (b). The single final state property of the models makes handling of models easy. LTSA generates the models as directed graphs in Aldebaran form. We have extended the checking phase for the Aldebaran form. In our implementation, we have chosen the value of goal pheromones randomly in accordance with the description Section 4.2.

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 used values as shown in Table 1 for the coefficients and constants that appear in equations and algorithms. These are the best settings we decided through preliminary experiments.

We conducted three experiments in terms of the length of counter parts, execution time, 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 8, our approach is more efficient than ACOhg. It shows that goal pheromones guide ants to the final states, suppressing going out their ways. Furthermore,

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.

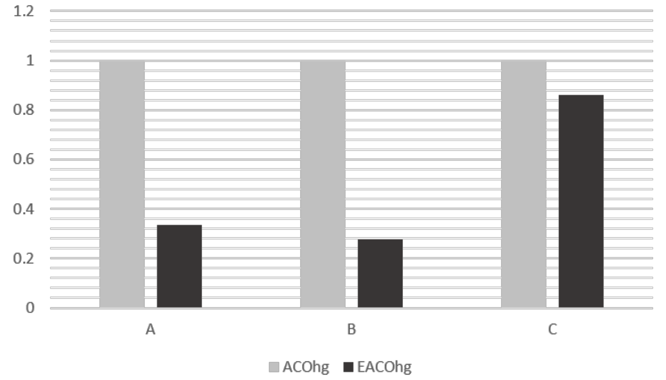


Figure 7: The length of counter parts

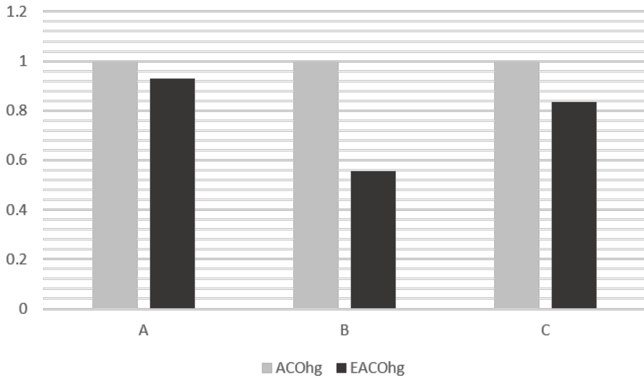


Figure 8: Execution time

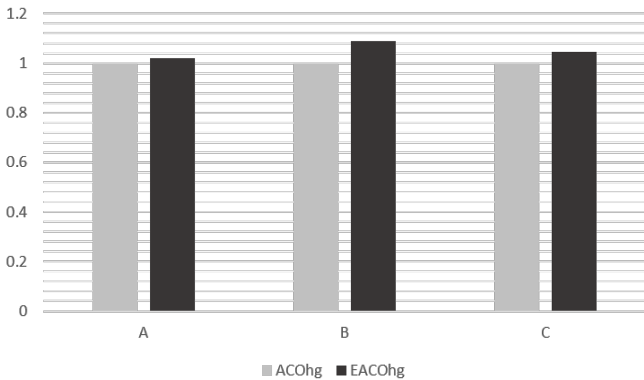


Figure 9: Memory consumption

Figure 7 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 9. The increase of the memory consumption fell within 10% at most, and is 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.

6. CONCLUSIONS AND FUTURE DIRECTIONS

We have proposed a new ACO based model checking 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, guiding ants to them, so that our approach can not only reach the goals further more efficiently but also generate much shorter counter examples.

In order to show the effectiveness of our approach, we implemented the EACOhg on a practical model checker, and conducted experiments. The results of experiments show that our approach is remarkably effective for improving execution efficiency and the length of counter examples.

We have observed that our approach does not achieve effectiveness in the very large model (model C) as shown in Figures 7 and 8. The reason may be that the effect of smell-like pheromone is compromised in a large model. In order to overcome this problem, we plan to make directionality in diffusing the smell-like pheromone. In addition, our extended ACO, EACOhg can be applicable to other optimization methods such as Partial Order Reduction (POR) [4], and checking of liveness [5]. 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.

Acknowledgments

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.

7. REFERENCES

[1] E. Alba and F. Chicano. Finding safety errors with aco. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1066–1073, 2007.

[2] E. Alba and J. M. Troya. Genetic algorithms for protocol validation. In *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, PPSN IV, pages 870–879, 1996.

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

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

[5] F. Chicano and E. Alba. Finding liveness errors with ACO. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC*, pages 2997–3004, 2008.

[6] E. M. Clarke. The birth of model checking. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, pages 1–26. 2008.

[7] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, 1982.

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

[9] M. Dorigo, V. Maniezzo, and A. Colorni. 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, 1996.

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

[11] S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with hsf-spin. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, pages 57–79, 2001.

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

[13] D. Giannakopoulou and J. Magee. 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, pages 257–266, 2003.

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

[15] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[16] A. Lluch-Lafuente, S. Edelkamp, and S. Leue. Partial order reduction in directed model checking. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 112–127. Springer-Verlag, 2002.

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

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

[19] J. Staunton and J. A. Clark. 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, pages 212–221, 2010.

[20] J. Staunton and J. A. Clark. 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, pages 1923–1930, 2011.

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