Exploration strategies for balancing efficiency and comprehensibility in model checking with ant colony optimization

Tsutomu Kumazawa, Munehiro Takimoto and Yasushi Kambayashi

Software Research Associates, Inc., Tokyo, Japan; Tokyo University of Science, Chiba, Japan; Nippon Institute of Technology, Saitama, Japan

ABSTRACT

Model checking is a formal and automated verification technique to show that a software system behaves in accordance with the given specification. Traditional model checking uses exhaustive search techniques for finding violative behaviours of the specification. The techniques, however, often do not work for huge systems because it demands a huge amount of computational resources. Search-Based Software Engineering is known to effectively solve many software engineering problems including model checking. It pursues the good balance between efficiency and qualities of solutions by using swarm intelligence and metaheuristic search methodologies. This article focuses on the state-of-the-art model checking with Ant Colony Optimization. Ant Colony Optimization is a metaheuristic, population-based and stochastic optimization algorithm. We propose two exploration strategies to further improve the balance in model checking based on Ant Colony Optimization. The proposed strategies introduce different kinds of randomized selection mechanisms to diversify solutions found by many agents. The strategies help the search algorithm extend the reachable regions effectively. Through numerical experiments, we confirmed that the proposed strategies require less computation time and memory as compared to the existing model checking with Ant Colony Optimization at the cost of finding slightly less qualified solutions.

1. Introduction

Search-Based Software Engineering (SBSE) (Harman et al., 2012) is a research field that aims to solve software engineering problems using search methodologies. The search methodologies contain Swarm Intelligence and metaheuristics, which are based on distributed, multi-agent and collectively intelligent approaches for finding better solutions efficiently. While SBSE is applied to many kinds of research topics, such as software design and refactoring, the most promising domain is formal verification.

CONTACT Tsutomu Kumazawa kumazawa@sra.co.jp
Formal verification is a mathematical approach to prove or refute correctness of software systems. This article focuses on model checking, which verifies software systems’ behavioural aspects automatically (Clarke & Emerson, 1982; Clarke et al., 2018). In model checking, the description of important behaviour of the target system is called a model. The desired property of the system is written as a specification. Model checking assumes that the model and the specification are given by the user. Then it decides whether the model satisfies the specification or not. The problem can be reduced to the exhaustive search problem over the state space constructed from the model and the specification. A model checking tool outputs a violative behaviour, i.e. a counterexample, if it concludes that the model does not satisfy the specification. There are two major research challenges of model checking: State Explosion Problem and the problem of Generating Comprehensible and Short Counterexamples. The state explosion is a problem about verification efficiency. State spaces tend to be huge. For example, the number of states in a concurrent and multiprocessing system often grows exponentially as the number of component processes increases linearly. The state explosion problem claims that huge state spaces make model checking infeasible in practice because the classical exhaustive algorithms deplete computational resources. On the other hand, generating comprehensible counterexamples is a usability problem. A counterexample provides diagnostic information that helps human users understand a specification’s violation easily. Therefore, it is more favourable to obtain short counterexamples than to obtain lengthy ones. These problems are in a trade-off relationship. When we want a short counterexample, the exhaustive search is necessary but it suffers from the state explosion. Therefore, we need to pursue a search algorithm such that it achieves the best balance between efficiency and shortness of counterexamples.

In order to mitigate those problems and to attain an appropriate balance, researchers have actively developed non-exhaustive verification techniques using swarm intelligence and metaheuristics (Kumazawa et al., 2020). One of such algorithms is Ant Colony Optimization (ACO) (Dorigo & Stützle, 2004). ACO solves the shortest path search problem effectively. ACOhg (Alba & Chicano, 2007a) is a variant of ACO and is suitable for model checking with huge state spaces (Alba & Chicano, 2007b; Chicano & Alba, 2008a). Chicano and Alba (2008b) proposed ACOhg-live, which is a state-of-the-art model checking algorithm by running ACOhg at two phases.

In this article, we propose two novel exploration strategies with the aim of enhancing the performance of ACOhg-live. We call the two strategies the skip strategy and the replacement strategy, respectively. In general, many states should be explored to arrive at goal states for large state spaces. Based on the idea, the proposed strategies adopt simple randomized approaches to realize the effective and efficient exploration of many states. By conducting comparative experiments, we confirmed that the proposed strategies improve the balance between efficiency and shortness of counterexamples as compared to ACOhg-live.

The rest of this article is organized as follows. Section 2 explains the outlines of model checking, namely ACOhg, and ACOhg-live. Section 3 presents the review on previous researches. Section 4 proposes our exploration strategies. Section 5 reports the results of our comparative experiments between ACOhg-live and the proposed strategies. We conclude our discussion in Section 6.
This article is an extension of the work originally presented in the Thirteenth International Conference on Computational Collective Intelligence (Kumazawa et al., 2021). The main extensions in this article are as follows.

- We add the state-of-the-art search techniques to the literature review (Section 3).
- In Section 4.1, we newly introduce two variants of the skip strategy. The proposed variants employ the different stochastic path selection methods that are different from the selection method in the original skip strategy proposed in our previous work (Kumazawa et al., 2021). The performance evaluation of the proposed variants is added in Section 5.
- We show the whole pseudocodes of the skip strategy (Algorithm 3) and the replacement strategy (Algorithm 4).
- We add another experimental result using a benchmark of random graphs (Section 5.2.2).
- We discuss our experiments in detail (Section 5.3). The newly added topics contain the analysis of the drawback found in our proposed strategies and the internal validity of our experiments.

2. Background

This section briefly introduces model checking and its automata-theoretic approach. We then present a model checking method based on Ant Colony Optimization.

2.1. Automata-theoretic model checking

Model checking is an automated approach to verify software systems in a formal fashion. It checks the behaviour of the target software system and determines whether it conforms to the predefined specifications. This article focuses on an automata-theoretic approach for model checking (Vardi & Wolper, 1986). This approach assumes that the state spaces are represented as finite automata. The model checking problem is reduced to the emptiness checking of the automata. Thus, the automata that are used in model checking are called Büchi automata. Just like a classical Nondeterministic Finite Automaton (NFA), a Büchi automaton consists of a set of states and a set of transition edges, a set of accepting states, and at least one initial state. A Büchi automaton can be represented as a directed graph. While an NFA accepts the finite words that reach some accepting state, a Büchi automaton recognizes the words of infinite length that visit some accepting state infinitely many times. In other words, the Büchi automaton accepts a word whose path, i.e. a sequence of states, is comprised of a cycle having some accepting states and a path from an initial state to the accepting states (Figure 1). We indicate an initial state by an incoming edge without source state. A doubly concentric circle represents an accepting state. For example, consider a Büchi automaton in Figure 2. It has two states 0 and 1, where 0 is the initial state and 1 is the accepting state, respectively. It accepts words passing state 1 infinitely many times, e.g. 0, 1, 1, 1, ... .

Model checking is conducted in the following manner (Clarke et al., 2018). First, we construct a Büchi automaton that accepts the words conforming to the system model
but contrary to the specification. When a specification is formulated with Linear Temporal Logic (Manna & Pnueli, 1992), its negation is transformed to the corresponding Büchi automaton (Clarke et al., 2018). The automaton constructed in this step is the intersection between the model and the automaton of the negated specification. Second, the emptiness of the Büchi automaton is examined with graph search algorithms. Classical model checking adopts exhaustive algorithms to find strongly connected components on directed graphs (Tarjan, 1972). If the Büchi automaton has an accepted word, the word is reported to the user as a counterexample. Otherwise, it concludes that the model satisfies the specification.

2.2. Model checking based on ant colony optimization

We introduce an automata-theoretic model checking technique using a variant of Ant Colony Optimization (ACO) (Dorigo & Stützle, 2004), called ACOhg (Alba & Chicano, 2007a, 2007b). ACO is motivated by foraging behaviour of worker ants. In ACO, many mobile agents (ants) search for paths over the given graph from an initial state (nest) to some goal state (food). They communicate with each other by means of pheromone deposited by the preceding agents. When each agent transits from a state to an adjacency, it prefers a state with denser pheromone trail to the ones with other pheromone trails. After the exploration, the agent puts the pheromone trail on its path. ACO converges to the shortest path since the denser pheromone trails a path has, the more likely it is selected by the succeeding agents. Pheromone trails evaporate as time elapses with the aim of avoiding local optima. ACO often suffers from state explosion for huge graphs because agents have to visit too many states. ACOhg is an extension of ACO specific to exploring huge graphs. In order to avoid depleting computational resources, ACOhg introduces the upper bound $\lambda_{ant}$ to the length of the path that an agent traverses. Agents gradually go to distant states by updating their initial states to the ones that the preceding agents have reached on a regular basis.
Algorithm 1 ACOhg (cited from (Chicano & Alba, 2008b))

1: init ← (initial_node);
2: next_init ← ∅;
3: τ ← initialize_pheromone();
4: step ← 1;
5: stage ← 1;
6: while step ≤ msteps do
7:    for k ← 1 to colsize do
8:        πk ← ∅;
9:        while |πk| ≤ λant ∧ T(πk−1) − πk + ∅ ∧ πk−1 ∈ F do
10:           node ← select_successor (πk−1, T(πk−1), τ, η);
11:           πk ← πk + node;
12:           τ ← local_pheromone_update (τ, ξ, node);
13:    end while
14:    next_init ← select_best_paths(init, next_init, πk);
15:    if f(πk) < f(πbest) then
16:        πbest ← πk;
17:    end if
18: end for
19:    τ ← pheromone_evaporation (τ, ρ);
20:    τ ← pheromone_update (τ, πbest);
21:    if step = 0 mod s then
22:       init ← next_init;
23:       next_init ← ∅;
24:       τ ← pheromone_reset()
25:    end if
26:    step ← step + 1;
27: end while

Algorithm 1 is the pseudocode of ACOhg. In lines 1–3, an initial state and pheromone trails are initialized to random densities. Here, the term initial_node indicates the initial state of the given Büchi automaton, i.e. the search space. After the initialization, the search process is repeated msteps times (line 6). The number of agents, i.e. the colony size, is denoted by colsize (line 7). In lines 9–13, each agent in the colony starts an initial state and keeps on moving over the state space until either the length of its path reaches λant or it encounters a goal state. The terms πk, |πk| and πk−1 represent a path traversed by agent k, its length and its last state, respectively. T(i) is the set of successors of state i and F denotes a set of the goal states. At line 10, a successor of the state where agent k currently stays is selected. The probability p^k_j where agent k in state i selects the adjacent state j is computed as follows:

\[ p^k_j = \frac{[\tau_j]^\alpha [\eta_j]^\beta}{\sum_{j \in T(i)} [\tau_j]^\alpha [\eta_j]^\beta}, \quad j \in T(i), \]  

where \( \tau_j \) is the density of pheromone deposited on state j, \( \eta_j \) is an appropriate heuristics, and \( \alpha \) and \( \beta \) are the scaling parameters. The density of pheromone trails on the visited states are weakened at a constant rate \( \xi \), i.e. \( \tau_j = (1 - \xi) \tau_j \) (line 12). After the traversal, the highly qualified paths are selected with the objective function f so that the shorter a path is, the better the value of f is (line 14). The function f for a path \( \pi^k \) of agent k is defined as follows:

\[ f(\pi^k) = \begin{cases} 
|\pi^k_{full}|, & \text{if } \pi^k_{-1} \in F, \\
|\pi^k_{full}| + h(\pi^k_{-1}) + p_p + p_c \frac{\lambda_{ant} - |\pi^k|}{\lambda_{ant} - 1}, & \text{if } \pi^k_{-1} \in F,
\end{cases} \]
where $\pi_{\text{full}}^k$ is a path from the initial state of ACOhg to $\pi_{-1}^k$, and $h$ is a heuristic function. The parameters $p_p$ and $p_c$ are the scales of the penalties for paths that do not reach any goal states and paths having cycles, respectively. In lines 15–17, the best path having found thus far is selected. Pheromone trails evaporate at a constant rate $\rho$, i.e. $\tau_i = (1 - \rho)\tau_i$ for each state $i$ (line 19). The lower bound of the trails is controlled by the parameter $a$. The pheromone trails deposited on the best-so-far path are then intensified using Equation (2) (line 20). From line 21 to 25, the initial states are updated to the last states of the best paths selected at line 14. The number of the updated initial states is up to the value of the predefined parameter $\iota$ (Chicano & Alba, 2008a). ACOhg conducts this update procedure once every $\sigma_s$ times, where $\sigma_s$ is a parameter.

ACOhg-live (Chicano & Alba, 2008b) is a model checking algorithm that finds short counterexamples. It uses ACOhg at two phases: one for detecting a path from an initial state to some accepting states, and the other for detecting a cycle containing the accepting states. Algorithm 2 summarizes ACOhg-live. If ACOhg-live finds some accepting states at the first phase (line 2), it starts the second phase to search for the path from the states to themselves (lines 3–5). Once the second phase finds such a path, ACOhg-live stops and outputs the counterexample. Otherwise, the first phase is conducted again until it fails to find any accepting states (line 6).

### Algorithm 2 ACOhg-live (Chicano & Alba, 2008b)

```plaintext
1: repeat
2:   Find accepting states with ACOhg; {First Phase}
3:   for all accepting states do
4:     Find a cycle with ACOhg; {Second Phase}
5:   end for
6: until no accepting state is found
```

### 3. Related work

Classical automata-theoretic model checking adopts exhaustive search algorithms based on Depth-First search (DFS), e.g. Nested DFS (Clarke et al., 2018) and Tarjan’s algorithm (Tarjan, 1972). As discussed before in this article, it is difficult for such classical methods to generate short counterexamples due to the state explosion problem. Recent work attempts to fully utilize the power of multi-core processors (Zakharov, 2018) in order to enhance efficiency. Schuppan and Biere (2005) investigated short counterexamples generation from the viewpoint of automata construction at the cost of enlarging state spaces.

Another research direction of model checking is the use of heuristic searches with the aim of enhancing check efficiency. Directed Model Checking (Edelkamp et al., 2008) has investigated heuristics for model checking. Randomized algorithms are also proposed such as DFS with randomized backtracking (Parízek & Lhoták, 2019) and Vagabond algorithm (Daszczuk, 2021). Swarm verification parallelizes multiple search strategies including randomization (DeFrancisco et al., 2020; Holzmann et al., 2011).

Metaheuristic approaches for formal verification have been studied since 1990s. Genetic Algorithm (GA) is a pioneering method that has been studied intensively (Alba & Troya, 1996; Godefroid & Khurshid, 2004; Yousefian et al., 2014). Besides ACOhg, several techniques based on ACO also have been proposed (Francesca et al., 2011; Rafe
et al., 2019; Rebiha & Ciampaglia, 2007). Other researches employed Particle Swarm Optimization (PSO) (Ferreira et al., 2008), Estimation of Distribution Algorithm (Staunton & Clark, 2010, 2011), Bayesian Optimization Algorithm (Pira et al., 2017) and Monte Carlo Tree Search (Milewicz & Poulding, 2018; Poulding & Feldt, 2015). Chicano et al. (2011) employed Simulated Annealing (SA) and reported that metaheuristic methods outperform classical deterministic methods through their comparative experiments. In order to make model checking more efficient, several researches employed hybrid approaches, which combine more than one search method. Researchers have hybridized Gravitational Search and PSO (Rafe et al., 2015), Artificial Bee Colony algorithm and SA (Rezaee & Momeni, 2020), and ACOhg and a randomized backward search (Kumazawa et al., 2019, 2016). The researches discussed above present ample evidence that heuristic and metaheuristic approaches are effective in model checking. This article focuses on ACOhg with the aim of improving the balance between efficiency and comprehensibility in model checking.

4. Proposed exploration strategies

This section proposes two novel strategies to enhance the exploration capability of ACOhg-live. We call them the skip strategy and the replacement strategy, respectively. Note that the proposed strategies have no additional user-defined parameters.

4.1. Skip strategy

The first phase of ACOhg-live finds an accepting state in a Büchi automaton. However, when the automaton has several accepting states and some of them are on the same path, the first phase ACOhg can retard agents’ exploring due to the intensive search within the small region. Since a Büchi automaton has more than one accepting state in general, this problem deteriorates the performance of ACOhg-live strikingly. Consider applying ACOhg-live to a Büchi automaton shown in Figure 3. Although the first phase finds a path from the initial state 0 to an accepting state 1, the consecutive second phase fails because state 1 is not a part of a strongly connected component. Thus, the first phase has to start from state 0 again to find the other accepting state 2. If ACOhg arrived at state 2 by further exploration, ACOhg-live would succeed in finding a cycle without running the first phase repetitively. The aforementioned problem becomes serious when ACOhg-live is applied to practical and industrial software verification problems. Models of such software systems tend to be very large and also have many accepting states. As a consequence, the first phase that has to explore the large state space may

Figure 3. Comparison between ACOhg and skip strategy.
be repeated many times. This phenomenon has a negative impact on both search efficiency and generation of short counterexamples.

We propose the skip strategy to address this problem. The idea of the strategy is that each agent keeps on moving until the length of its path reaches the upper bound $\lambda_{ant}$ even when it encounters some goal state halfway. After it stops exploring, the strategy extracts one of the subpaths that ends at goal states randomly. Back to the example in Figure 3, when $\lambda_{ant} > 2$, the skip strategy enables the first phase ACOhg to reach both state 1 and 2.

Algorithm 3 shows the modified ACOhg with the skip strategy. At line 6, we add another termination condition such that the algorithm ends if there are one or more agents that arrive at one of the goal states. We call this termination condition Stop-on-Arrival condition in the following. Stop-on-Arrival condition is equivalent to the one adopted in (Chicano & Alba, 2008a) and is necessary for the efficiency of the skip strategy. We then change the manner of exploration by each agent, as shown in lines 10–20. From line 10 to 17, an agent repeatedly transits states up to visiting $\lambda_{ant}$ states. Whenever it comes across accepting states on its way, it ignores the states and keep on moving. We store a set of its subpaths whose last states are goal states. After the agent builds a path, a random selection is conducted to extract one of the stored subpaths (lines 18–20). The selection is performed in accordance with Uniform

Algorithm 3 Skip Strategy

1: init $\leftarrow (initial\_node)$;
2: next_init $\leftarrow \emptyset$;
3: $\tau \leftarrow initialize\_pheromone()$;
4: step $\leftarrow 11$;
5: stage $\leftarrow 11$;
6: while step $\leq msteps \land [1\ldots coli\_size]: \pi_{-1}^k \in F$ do
7:     for $k \leftarrow 1$ to coli\_size do
8:         $\pi^k \leftarrow \emptyset$;
9:         subpaths $\leftarrow \emptyset$;
10:        while $|\pi^k| \leq \lambda_{ant} \land T(\pi_{-1}^k) - \pi^k \neq \emptyset$ do
11:            if $\pi_{-1}^k \in F$ then
12:                subpaths $\leftarrow$ subpaths $+$ $\pi^k$;
13:            end if
14:            node $\leftarrow select\_successor(\pi_{-1}^k, T(\pi_{-1}^k), \tau, \eta)$;
15:            $\pi^k \leftarrow \pi^k +$ node;
16:            $\tau \leftarrow local\_pheromone\_update(\tau, \xi, node)$;
17:        end while
18:        if subpaths $\neq \emptyset$ then
19:            $\pi^k \leftarrow random\_selection(subpaths)$;
20:        end if
21:        next_init $\leftarrow select\_best\_paths(init, next\_init, \pi^k)$;
22:        if $f(\pi^k) < f(\pi^{best})$ then
23:            $\pi^{best} \leftarrow \pi^k$;
24:        end if
25:    end for
26:    $\tau \leftarrow pheromone\_evaporation(\tau, \rho)$;
27:    $\tau \leftarrow pheromone\_update(\tau, \pi^{\text{best}})$;
28:    if step $= 0$ mod $s$ then
29:        init $\leftarrow$ next_init;
30:        next_init $\leftarrow \emptyset$;
31:    end if
32:    step $\leftarrow$ step $+$ 1;
33: end while
Distribution. We enumerate all of the nearby accepting states as candidates if the size of the colony is sufficiently large. The procedure below line 21 of Algorithm 3 is the same as that below line 14 of Algorithm 1.

We can consider a variety of the skip strategy by changing the probability distribution used in random selection at line 19. In addition to the random path selection according to Uniform Distribution, we introduce the following weighted probability for the different path selection methods. In random selection, we first index the subpaths that are considered. Then, we select the \( i \)-th path from the subpaths according to the following probability.

\[
q_i = \frac{i}{\sum_{j=1}^{N} j}
\]  

(3)

where \( N \) is the size of the subpaths and \( i = 1, \ldots, N \). Using Equation (3), we propose two variants of the skip strategy whose manners of indexing the subpaths are as follows.

- **Forward Indexing**: We give the subpaths indices by the ascending order from the first detected subpath to the last.
- **Backward Indexing**: We give the subpaths indices by the descending order.

From Equation (3), the forward indexing prioritizes selecting the subpath that is discovered later. Thus, this indexing encourages the agents to explore deeply. In contrast, the backward indexing selects the subpath found earlier with high probability. This indexing makes the agents behave in the manner similar to ACOhg.

### 4.2. Replacement strategy

Agents in ACOhg need to extend their exploration and their paths to goal states when their preceding agents do not reach goal states. However, if the paths found by the preceding agents are evaluated well as compared to the ones found by the followers, it is likely that the already-known paths are selected many times at line 14 of Algorithm 1. This stagnation is unfavourable because the agents tend to explore some small region of the huge state space intensively and cannot extend their exploration. This situation can be in danger of falling into local optima.

In order to address this problem, we introduce the other exploration strategy called the replacement strategy. The idea of this strategy is to jump into relatively inferior solutions stochastically in case of stagnation. When agents fail to find superior paths, the proposed strategy selects some of the superior paths randomly and replaces them with inferior ones (Figure 4). After the exploration step at lines 7–18 of Algorithm 1, we identify the stagnation by checking whether or not the selected paths at line 14 are the same as the one before the selection. The replacement strategy then removes each of the paths at probability 0.5. The highly ranked paths found as the result of the procedure from line 9 to 13 are added to the remaining best-so-far paths so that the number of the selected paths can be up to the parameter \( \iota \) to conduct line 14 and its subsequent lines.
Algorithm 4 Replacement Strategy

1: init ← \{initial_node\};
2: next_init ← ∅;
3: τ ← initialize_pheromone();
4: step ← 1;
5: stage ← 1;
6: while step ≤ msteps do
7:   paths ← ∅;
8:   for k ← 1 to colsize do
9:     p_k ← ∅;
10:    while |p_k| ≤ lant \(\) + T(p_k−1) − π_k−1 \(\) ≠ ∅ \(\) ∧ \(\) π_k−1 \(\) ∈ F do
11:       node ← select_successor (p_k−1, T(p_k−1), τ, η);
12:       π_k ← π_k + node;
13:       stage ← local_pheromone_update (τ, η, node);
14:   end while
15:   paths ← paths + p_k
16:   next_init_temp ← select_best_paths(init, next_init, \(\) π_k\(\));
17:   if \(f(\pi_k) < f(p_{best})\) then
18:      p_{best} ← p_k;
19:   end if
20: end for
21: if next_init = next_init_temp then
22:   n ← 0;
23: for all path ∈ next_init do
24:   if rand() < 0.5 then
25:      next_init ← next_init - path; {Remove path}
26:      n ← n + 1;
27:   end if
28: end for
29: m ← 0;
30: for all path ∈ paths − next_init do
31:   if m ≤ n and || ≤ ε then
32:      next_init ← next_init + path; {Add inferior path}
33:      m ← m + 1;
34:   else
35:      break;
36: end if
37: end for
38: else
39:   next_init ← next_init_temp; {Stagnation not occurred}
40: end if
41: Run Line 19 to 26 in Algorithm 1
42: end while

Figure 4. Replacement strategy.

Based on the above idea, Algorithm 4 presents the whole algorithm of ACOhg with the replacement strategy. Its description from line 1 to 20 is the same as that of Algorithm 1 from line 1 to 18, except that the paths found by the agents are stored into temporary
variables at lines 15 and 16. Also, we conduct the procedure from line 19 to 26 of Algorithm 1 without modification as described at line 41 in Algorithm 4. The main body of the replacement strategy is the procedure from line 21 to 40. Line 21 checks whether the stagnation arises or not. If we detect the stagnation, we first remove some best-so-far paths at random from line 23 to 28. Then, the inferior paths found by the agents are added to the remaining best-so-far paths from line 30 to 37. If we do not find the stagnation, the best-so-far paths are passed to the next iteration without replacement (line 39).

5. Experiments

We conducted performance comparison experiments between the original ACOhg-live and the modified ACOhg-live with the proposed strategies using two benchmarks.

5.1. Experimental settings

We implemented the prototype tool that runs ACOhg-live and the proposed strategies from scratch with Python 3.6. The models and the specifications are inputted as Labeled Transition Systems (LTSS) (Magee & Kramer, 2006). LTSSs are suitable for describing concurrent state transition systems firing event sequences. In order to make the prototype efficient, we implemented on-the-fly checking (Clarke et al., 2018), which creates the state space as needed. The tool outputs either a counterexample as an event sequence if a violation is found, or None, i.e. a special value built in Python, otherwise.

We prepared two benchmarks. The first benchmark is a database ring system, whose model is based on (Magee & Kramer, 2006). This benchmark describes a distributed database that consists of several communication nodes organizing a ring-shaped network. Each node receives updated information from one of its adjacent nodes, and changes its internally stored data. The node notifies the other adjacent node of the information to keep the consistency of the network. We prepared three specifications, called Spec. A, B and C, which are related to the requirements that the all nodes update their internal states successfully.

The other benchmark is a model that consists of random graphs. We generated five Erdős-Rényi graphs (Erdős & Rényi, 1959), where the number of nodes is 20 and the probability of creating an edge is 0.1. The graphs are then transformed to LTSSs. We regard the LTSSs as component processes. We followed this procedure to generate another Büchi automaton of a specification. This time we randomly selected some states as accepting states.

We set the following search conditions and ran the prototype thirty times for each specification and each experimental condition:

- **OR**: running the original ACOhg-live without our proposed strategies,
- **ORT**: running the original ACOhg-live by corresponding the terminal condition of its first phase with Stop-on-Arrival condition, with the aim of evaluating our strategies under the fair terminal condition,
- **SK**: applying the skip strategy with Uniform Distribution in the first phase,
- **SF**: applying the skip strategy with the weighted probability defined by Equation (3) and the forward indexing in the first phase,
- **SB**: applying the skip strategy with the weighted probability defined by Equation (3) and the backward indexing in the first phase,
- **RE**: applying the replacement strategy in the second phase,
- **RET**: running the first stage of the original ACOhg-live by corresponding the terminal condition with Stop-on-Arrival condition and applying the replacement strategy in the second phase,
- **HY**: applying the skip strategy with Uniform Distribution and the replacement strategy in the first and the second phase, respectively,
- **HF**: applying the skip strategy with the weighted probability defined by Equation (3) and the forward indexing, and the replacement strategy in the first and the second phase, respectively, and
- **HB**: applying the skip strategy with the weighted probability defined by Equation (3) and the backward indexing, and the replacement strategy in the first and the second phase, respectively.

For each experimental condition, we measured the length of the detected counterexample, the running time, and the memory consumption. The length of a counterexample evaluates the comprehensibility problem. Running time and memory consumption evaluate how efficiently the state explosion problem is handled. We used the settings of the parameters for ACOhg-live shown in **Table 1**. We selected this configuration tuned the previously established study (Chicano & Alba, 2008b). ACOhg is assumed to combine heuristics (Section 2.2). In our experiments, we did not use any heuristics for the first phase since accepting states are not known in advance. For the second phase, we employed the hamming distance of binary representations of states (Chicano & Alba, 2008b). This heuristics estimates the distances between the current state and the accepting state to be searched. We used HP ProDesk 600 G4 SFF with Intel Core i7-8700, CPU 3.20GHz, RAM 16.0 GB, and Windows 10 Pro for the experiments. We turned off the garbage collection built in Python when running the prototype in order to avoid its influence on the performance.

### 5.2. Results

#### 5.2.1. Database ring system

We show the results of our experiments in **Figure 5** and **Table 2**. In **Table 2**, the lengths of counterexamples, running time and memory consumption are denoted by ‘Length’,

| Parameter | First phase ACOhg | Second phase ACOhg |
|-----------|-------------------|-------------------|
| msteps    | 10                | 10                |
| csize     | 10                | 10                |
| λ_{act}   | 20                | 10                |
| σ_{g}     | 2                 | 2                 |
| ι         | 10                | 10                |
| ξ         | 0.7               | 0.5               |
| a         | 5                 | 5                 |
| p          | 0.2               | 0.2               |
| a         | 1.0               | 1.0               |
| β         | 2.0               | 2.0               |
| p_{r}     | 1000              | 1000              |
| p_{c}     | 1000              | 1000              |
‘Time’ and ‘Memory’, respectively. The terms ‘Sp.’, ‘Avg.’, ‘SD’ and ‘MD’ are abbreviations of the target specification, average, standard deviation, and median, respectively.

For running time and the memory consumption, the skip strategies (SK, SF and SB) improved the original ACOhg-live (OR) and ACOhg-live with Stop-on-Arrival condition (ORT and RET) for each specification. The results suggest that the skip strategy promotes effective exploration of the state space. SK, SF and SB tend to generate a bit longer counterexamples as compared to OR, ORT and RE on average for Spec. B and C. SK, SF and SB make many agents extend on their exploration preferentially and weakens the feature of ACOhg-live to generate short counterexamples. The replacement strategy (RE) does not present significant differences about the performance as compared to OR. The observation indicates that it is difficult to improve the performance of ACOhg-live with the replacement strategy alone.

We discuss the effectiveness of combining both the skip and replacement strategies (HY, HF and HB) next. For the running time in Spec. A, the average and the median of HY, HF and HB are almost comparable to those of the corresponding skip strategies,

Figure 5. Box plots for database ring system. For each case (a), (b) and (c), the plots are placed in order of Lengths of Counterexamples, Running Time and Memory Consumption from left to right.
Table 2. Summary for database ring system.

| Sp. | Item | Length | Avg. | SD | MD | Time (s) | Avg. | SD | MD | Memory (kb) | Avg. | SD | MD |
|-----|------|--------|------|----|----|---------|------|----|----|-------------|------|----|----|
| A   | OR   | 15.23  | 3.48 | 15 | 3.45 | 0.70   | 3.49 | 267.35 | 35.38 | 262.51 |
|     | ORT  | 21.67  | 10.44 | 16 | 1.87 | 0.24   | 1.81 | 184.18 | 15.72 | 187.37 |
|     | SK   | 17.50  | 7.42  | 16 | 1.41 | 0.39   | 1.25 | 154.59 | 23.89 | 148.71 |
|     | SF   | 16.43  | 4.59  | 16 | 1.49 | 0.46   | 1.52 | 159.73 | 27.04 | 159.73 |
|     | SB   | 17.90  | 7.32  | 16 | 1.41 | 0.51   | 1.25 | 158.49 | 31.61 | 148.16 |
|     | RE   | 15.17  | 3.33  | 15 | 3.40 | 0.68   | 3.29 | 266.97 | 41.48 | 267.49 |
|     | RET  | 17.87  | 7.51  | 15 | 1.94 | 0.20   | 2.02 | 176.98 | 11.96 | 174.33 |
|     | HY   | 16.90  | 6.33  | 16 | 1.40 | 0.47   | 1.26 | 158.70 | 19.72 | 156.24 |
|     | HF   | 18.57  | 8.29  | 15  | 1.34 | 0.27   | 1.28 | 166.13 | 20.40 | 161.46 |
|     | HB   | 16.23  | 4.72  | 16 | 1.47 | 0.49   | 1.27 | 153.94 | 25.72 | 146.33 |
| B   | OR   | 27.47  | 1.93  | 26 | 11.53 | 0.68  | 11.53 | 785.81 | 66.24 | 773.68 |
|     | ORT  | 27.57  | 1.94  | 26 | 9.66 | 1.18   | 9.40 | 622.97 | 70.35 | 628.91 |
|     | SK   | 29.40  | 2.74  | 28 | 1.16 | 0.37   | 1.09 | 175.72 | 35.88 | 174.52 |
|     | SF   | 29.90  | 2.86  | 29 | 0.92 | 0.40   | 0.88 | 143.23 | 26.08 | 154.38 |
|     | SB   | 29.80  | 2.06  | 30 | 1.91 | 0.45   | 1.87 | 191.29 | 29.98 | 188.21 |
|     | RE   | 27.60  | 1.96  | 26 | 11.83 | 0.73  | 11.84 | 765.14 | 42.26 | 774.07 |
|     | RET  | 28.03  | 1.97  | 28.5 | 9.44 | 1.07  | 9.15 | 615.33 | 65.18 | 603.93 |
|     | HY   | 30.17  | 2.31  | 31 | 1.16 | 0.32   | 1.04 | 164.95 | 28.88 | 160.18 |
|     | HF   | 31.77  | 2.39  | 32 | 0.75 | 0.28   | 0.73 | 127.05 | 19.65 | 131.81 |
|     | HB   | 29.13  | 2.65  | 29 | 1.66 | 0.44   | 1.58 | 195.47 | 19.55 | 193.97 |
| C   | OR   | 28.93  | 1.77  | 30 | 12.04 | 1.35  | 11.71 | 770.61 | 129.75 | 741.25 |
|     | ORT  | 28.00  | 1.83  | 27.5 | 9.56 | 4.36  | 9.16 | 483.03 | 193.30 | 454.47 |
|     | SK   | 29.70  | 2.84  | 29 | 1.49 | 0.74   | 1.41 | 176.92 | 44.22 | 175.67 |
|     | SF   | 28.37  | 1.80  | 28 | 1.16 | 0.66   | 1.00 | 138.95 | 30.88 | 142.72 |
|     | SB   | 28.33  | 2.49  | 28.5 | 2.03 | 0.92   | 2.02 | 196.40 | 39.43 | 197.78 |
|     | RE   | 29.07  | 1.69  | 30 | 11.78 | 1.35  | 11.42 | 726.17 | 52.75 | 723.51 |
|     | RET  | 29.77  | 2.33  | 30.5 | 4.83 | 2.34  | 4.55 | 349.29 | 124.68 | 340.87 |
|     | HY   | 31.30  | 3.62  | 31.5 | 1.12 | 0.39   | 1.03 | 142.43 | 21.14 | 139.15 |
|     | HF   | 30.80  | 3.03  | 31 | 0.66 | 0.19   | 0.67 | 124.15 | 21.14 | 122.83 |
|     | HB   | 30.30  | 3.51  | 30 | 1.47 | 0.42   | 1.52 | 159.88 | 24.38 | 159.70 |

i.e. SK, SF and SB, respectively, and are small as compared to those of OR, ORT, RE and RET. According to the result in Spec. B, the average, standard deviation and median of the memory consumption of the hybrid methods HY, HF and HB are the smallest. The hybrid methods HY, HF and HB are the most efficient with respect to both the running time and memory consumption for Spec. C. Except the running time in Spec. A, HY, HF and HB tend to decrease the standard deviations of the running time and memory consumption as compared to the corresponding skip strategies SK, SF and SB respectively. Although the lengths of counterexamples found by the hybrid methods worsen especially for Spec. B and C as compared to the other conditions, we believe that the performance improvement with the hybridization of our strategies compensates for this disadvantage of detecting long counterexamples. Our strategies are expected to provide ACOh-g-live with the good balance between performance and comprehensibility of counterexamples.

5.2.2. Randomly generated model

The experimental results are shown in Figure 6 and Table 3. OR generates the shortest counterexamples on average of all experimental conditions. For running time and
memory consumption, Stop-on-Arrival condition (ORT and RET) and our strategies (SK, SF, SB, HY, HF and HB) show the good results as compared to OR. Whereas the memory consumption of our strategies is comparable to that of Stop-on-Arrival condition, the contribution of our strategies to running time is more considerable than that of Stop-on-Arrival condition. Considering these observations, the proposed strategies balance well in comparison with the original ACOhg-live in terms of both efficiency and comprehensibility of counterexamples as the case of the database ring system. We do not observe considerable differences between the results of three variants of the skip strategy.

5.3. Discussion

We discuss why the proposed strategies make ACOhg-live generate rather long counterexamples. Recalling that Stop-on-Arrival condition (ORT and RET) tends to generate long counterexamples as compared to the original ACOhg-live (OR) according to the experimental results, we conjecture that this condition can weaken ACOhg’s effect of shortening paths. In fact, the employment of Stop-on-Arrival condition implies that the process of shortening paths works well until the arrival at some accepting state. Unlike ordinal ACO, which searches and shortens paths from the initial state to the goal state, ACOhg with Stop-on-Arrival condition shortens only subpaths and sometimes get into local optima. Whereas Stop-on-Arrival condition has the drawback in terms of solution quality, it presents an advantage in terms of both time and space performance. A potential
improvement for the drawback is to use effective heuristics because they have great influences on performance and quality of solutions.

In our experiments, we investigated three variants of the skip strategy. According to our results, although the weighted probability defined by Equation (3) with the forward indexing tends to perform the most favourably for the database ring system, we observe little difference among the proposed variants for the randomly generated model. These results suggest that we have to select the most appropriate variants depending on verification problems. We have to conduct further investigation to understand the relationships of the proposed variants with characteristics of verification problems.

We next discuss the internal issues of our experiments. First, our prototype implementation has much room for improvement especially in terms of performance enhancement. We are planning to incorporate various optimization techniques of model checking, e.g. partial order reduction (Chicano & Alba, 2008a). Since our implementation runs in a sequential manner, we can improve the performance of the proposed strategies and our search procedure by parallelizing them. Second, our parameter settings of ACOhg can affect its convergence efficiency. Although our settings are based on the one established in the previous study (Chicano & Alba, 2008b), there may exist more efficient settings than ours for each benchmark. However, such settings can be too specific to the problem domain and can arise difficulty with respect to generalization. Third, because ACO is a stochastic search algorithm, we have to draw many samples and observe the averaged performance. Unfortunately, it is very difficult to determine the appropriate sample size in general. In our experiments, we sampled thirty runs independently for each condition. We believe that our sample size is reasonable.

We also discuss the external conditions of our experimental results. The database ring benchmark is a typical distributed and concurrent system. Since one of the most prominent features of model checking is verifying concurrent systems (Clarke et al., 2018), we believe that our experiments are appropriate for evaluating the proposed strategies under a representative use case of model checking. The other benchmark is comprised of random graphs. We expect that the benchmark does not depend on characteristics that are specific to the database ring benchmark. Hence, the model provides us with evidence of the general effectiveness of our proposed strategies.

### 6. Conclusion

This article tackles with two main challenges in model checking: the state explosion problem and the problem of finding short counterexamples. Our approach is involved with Search-Based Software Engineering, which tries to overcome the problems with non-exhaustive search algorithms based on swarm intelligence. We propose two novel exploration strategies for making an existing model checking technique based on Ant Colony Optimization, ACOhg-live, more efficient. Our experimental results lead us to the conclusion that ACOhg-live with the proposed strategies outperforms the original one with respect to runtime performance. Although our strategies detect longer counterexamples than the original ACOhg-live does, we believe that the drawback is subtle.

As future work, we plan to combine appropriate heuristics with our strategies. We will select the heuristics that are suitable for the formalism of automata-theoretic model
checking in our implementation. We also have to compare the proposed strategies with the search methods other than ACOhg-live, e.g. the techniques discussed in Section 3.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Notes on contributors

Tsutomu Kumazawa is a researcher at Key Technology Laboratory in Software Research Associates, Inc., and a visiting fellow in Tokyo University of Science. His main research interests are formal verification and its industrial applications. He received his M.S. and Ph.D. from the University of Tokyo and B.A. in Engineering from Tokyo Metropolitan University.

Munehiro Takimoto is a professor in the Department of Information Sciences from Tokyo University of Science. His research interests include theory and practice of programming languages, and the various things derived from them, which include mobile agent systems and their applications. He received his Ph.D., M.S., and B.A in Engineering from Keio University. He is a member of ACM, IEEE Computer Society, IPSJ, JSSST, and IEICE.

Yasushi Kambayashi is an associate professor in the Department of Computer and Information Engineering from Nippon Institute of Technology. He previously worked at Mitsubishi Research Institute as a staff researcher. His research interests include theory of computation, multi-agent systems, and political science. He received his MSc and PhD in Computer Science from the University of Washington and the University of Toledo, respectively and his BA in Political Science from Keio University.

ORCID

Tsutomu Kumazawa http://orcid.org/0000-0002-1096-1097

References

Alba, E., & Chicano, F. (2007a). ACOhg: dealing with huge graphs. In 9th Annual Conference on Genetic and Evolutionary Computation (pp. 10–17) Association for Computing Machinery. https://doi.org/10.1145/1276958.1276961

Alba, E., & Chicano, F. (2007b). Finding safety errors with ACO. In 9th Annual Conference on Genetic and Evolutionary Computation. Association for Computing Machinery. (pp. 1066–1073). https://doi.org/10.1145/1276958.1277171

Alba, E., & Troya, J. M. (1996). Genetic algorithms for protocol validation. In 4th International Conference on Parallel Problem Solving from Nature. Springer (pp. 870–879). https://doi.org/10.1007/3-540-61723-X_1050

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

Chicano, F., & Alba, E. (2008b). Finding liveness errors with ACO. In IEEE congress on evolutionary computation (pp. 2997–3004). IEEE. https://doi.org/10.1109/CEC.2008.4631202

Chicano, F., Ferreira, M., & Alba, E. (2011). Comparing metaheuristic algorithms for error detection in java programs. In 3rd International Conference on Search Based Software Engineering (pp. 82–96). Springer. https://doi.org/10.1007/978-3-642-23716-4_11

Clarke, E. M., & Emerson, E. A. (1982). Design and synthesis of synchronization skeletons using branching-time temporal logic. In Logic of programs, workshop (pp. 52–71). https://doi.org/10.1007/BFb0025774
Clarke, E. M., Kroening, D., Peled, D., & Veith, H. (2018). Model checking (2nd ed.). MIT Press.

Daszczuk, W. B. (2021). Non-exhaustive verification in integrated model of distributed systems (IMDS) using vagabond algorithm. In Theory and engineering of dependable computer systems and networks (pp. 32–43). Springer.

DeFrancisco, R., Cho, S., Ferdman, M., & Smolka, S. A. (2020). Swarm model checking on the GPU. International Journal on Software Tools for Technology Transfer, 22(5), 583–599. https://doi.org/10.1007/s10009-020-00576-x

Dorigo, M., & Stützle, T. (2004). Ant colony optimization. Bradford Company, MIT Press.

DeFrancisco, R., Cho, S., Ferdman, M., & Smolka, S. A. (2020). Swarm model checking on the GPU. International Journal on Software Tools for Technology Transfer, 22(5), 583–599. https://doi.org/10.1007/s10009-020-00576-x

Edelkamp, S., Schuppan, V., Bosnacki, D., Wijs, A., Fehnker, A., & Aljazzar, H. (2008). Survey on directed model checking. In 5th international workshop on model checking and artificial intelligence (Vol. 5348, pp. 65–89). Springer. https://doi.org/10.1007/978-3-642-00431-5_5

Erdős, P., & Rényi, A. (1959). On random graphs I. Publicationes Mathematicae Debrecen, 6, 290–297.

Ferreira, M., Chicano, F., Alba, E., & Gómez-Pulido, J. A. (2008). Detecting protocol errors using particle swarm optimization with java pathfinder. In High Performance Computing & Simulation Conference. ECMS. (pp. 319–325).

Francesca, G., Santone, A., Vaglini, G., & Villani, M. L. (2011). Ant colony optimization for deadlock detection in concurrent systems. In IEEE 35th Annual Computer Software and Applications Conference (pp. 108–117). https://doi.org/10.1109(COMPSAC.2011.22

Godefroid, P., & Khurshid, S. (2004). Exploring very large state spaces using genetic algorithms. International Journal on Software Tools for Technology Transfer, 6(2), 117–127. https://doi.org/10.1007/s10009-004-0141-1

Harman, M., Mansouri, S. A., & Zhang, Y. (2012). Search-based software engineering: trends, techniques and applications. ACM Computing Surveys, 45(1), 11:1–11:61. https://doi.org/10.1145/2379776.2379787

Holzmann, G. J., Joshi, R., & Groce, A. (2011). Swarm verification techniques. IEEE Transactions on Software Engineering, 37(6), 845–857. https://doi.org/10.1109/TSE.2010.110

Kumazawa, T., Takada, K., Takimoto, M., & Kambayashi, Y. (January, 2019). Ant colony optimization based model checking extended by smell-like pheromone with hop counts. Swarm and Evolutionary Computation, 44, 511–521. https://doi.org/10.1016/j.swevo.2018.06.002

Kumazawa, T., Takimoto, M., & Kambayashi, Y. (2020). A survey on the applications of swarm intelligence to software verification. In Handbook of research on fireworks algorithms and swarm intelligence (pp. 376–398). https://doi.org/10.4018/978-1-7998-1659-1.ch017

Kumazawa, T., Takimoto, M., & Kambayashi, Y. (2021). Exploration strategies for model checking with ant colony optimization. In Computational collective intelligence (pp. 264–276). Springer. https://doi.org/10.1007/978-3-030-88081-1-20

Kumazawa, T., Yokoyama, C., Takimoto, M., & Kambayashi, Y. (2016). Ant colony optimization based model checking extended by smell-like pheromone. EAI Endorsed Transactions on Industrial Networks and Intelligent Systems, 16(7). https://doi.org/10.4108/eai.21-4.2016.151156

Magee, J., & Kramer, J. (2006). Concurrency: state models & java programming (2nd ed.). John Wiley & Sons.

Manna, Z., & Pnueli, A. (1992). The temporal logic of reactive and concurrent systems: specification. Springer.

Milewicz, R. M., & Poulding, S. (2018). Scalable parallel model checking via Monte-Carlo tree search. ACM SIGSOFT Software Engineering Notes, 42(4), 1–5. https://doi.org/10.1145/3149485.3149495

Parízek, P., & Lhoták, O. (2019). Fast detection of concurrency errors by state space traversal with randomization and early backtracking. International Journal on Software Tools for Technology Transfer, 21(4), 365–400. https://doi.org/10.1007/s10009-018-0484-7

Pira, E., Rafe, V., & Nikanjam, A. (2017). Deadlock detection in complex software systems specified through graph transformation using Bayesian optimization algorithm. Journal of Systems and Software, 131, 181–200. https://doi.org/10.1016/j.jss.2017.05.128

Poulding, S., & Feldt, R. (2015). Heuristic model checking using a Monte-Carlo tree search algorithm. In 2015 Annual Conference on Genetic and Evolutionary Computation (pp. 1359–1366). https://doi.org/10.1145/2739480.2754767
Rafe, V., Darghayedi, M., & Pira, E. (2019). MS-ACO: a multi-stage ant colony optimization to refute complex software systems specified through graph transformation. *Soft Computing, 23*(12), 4531–4556. https://doi.org/10.1007/s00500-018-3444-y

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

Rebiha, R., & Ciampaglia, G. L. (2007). An ant colony verification algorithm. In *7th International Conference on Intelligent Systems Design and Applications* (pp. 901–906). https://doi.org/10.1109/ISDA.2007.63

Rezaee, N., & Momeni, H. (2020). A hybrid meta-heuristic approach to cope with state space explosion in model checking technique for deadlock freeness. *Journal of AI and Data Mining, 8*(2), 189–199. https://doi.org/10.22044/jadm.2019.7564.1900

Schuppan, V., & Biere, A. (2005). Shortest counterexamples for symbolic model checking of LTL with past. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 493–509). Springer. https://doi.org/10.1007/978-3-540-31980-1_32

Staunton, J., & Clark, J. A. (2010). Finding short counterexamples in promela models using estimation of distribution algorithms. In *2010 3rd International Conference on Software Testing, Verification, and Validation Workshops* (pp. 212–221). https://doi.org/10.1109/ICSTW.2010.24

Staunton, J., & Clark, J. A. (2011). Searching for safety violations using estimation of distribution algorithms. In *13th Annual Conference on Genetic and Evolutionary Computation* (pp. 1923–1930). https://doi.org/10.1145/2001576.2001834

Tarjan, R. (1972). Depth first search and linear graph algorithms. *SIAM Journal on Computing, 1*(2), 146–160. https://doi.org/10.1137/0201010

Vardi, M. Y., & Wolper, P. (1986). An automata-theoretic approach to automatic program verification. In *First symposium on logic in computer science*. IEEE Computer Society Press. (pp. 322–331).

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

Zakharov, I. (2018). A survey of high-performance computing for software verification. In *Tools and methods of program analysis* (pp. 196–208). Springer.