An Ant Colony Verification Algorithm

Rachid Rebiha Giovanni L. Ciampaglia Università della Svizzera Italiana, Faculty of Informatics. {rachid.rebiha, ciampagg}@lu.unisi.ch

Abstract

Büchi automata are widely used as a modeling formalism in formal verification. The emptiness check procedure is used to carry on the model checking of a model M of a system, against an LTL formula ϕ , that expresses the desidered properties the system should satisfy. Algorithms for the emptiness check on Büchi automata are able to find a faulty computation, if the language accepted by the automaton of the synchronized product of \mathcal{M} and ϕ is non empty. Nonetheless, these algorithms don't perform any optimization on such solution. In this paper we propose an ant colony optimization algorithm for the emptiness check of a subclass of Büchi automata, that runs on-the-fly, that is without storing the whole digraph of the automaton in main memory, thus avoiding the problem of exponential growth of the state space. Our approach features a non standard search strategy, with the capability for artificial ants to backtrack during the constructive search, and to modify the graph of the automaton, by removing parts of it when no longer interesting for exploration.

1. Introduction

We propose an on-the-fly Ant Colony algorithm for the emptiness check on transition-based generalized Büchi automata (TGBA). These automata allow more compact translation of LTL formulæ [1], [2] (e.g. the property ϕ to be checked) and can be used to represent the model ${\cal M}$ of the finite state system being checked. By taking the intersection (the synchronized product) of the languages accepted by both the automata of the model and the automata of the property, the emptiness check tells us if \mathcal{M} contains a counter-example of ϕ . The emptiness check reduces to the search of strongly connected components (SCC) labeled with a specific property, on the associated directed graph (more precisely, a SCC containing all accepting conditions labeling their arcs). Existing methods are based on the Tarjan algorithm: they don't propose optimizations of the solution and perform the search over a completely stored data structure (they are thus exposed to the common "State Space Explosion Problem" drawback). The Ant Colony Algorithm we propose has the following main advantages: (i) it supports on-the-fly computations, that is a new generic method for applying ACO: the artificial ants build the graph, or remove a part of it when needed, while performing their exploration. (ii) Every ant looks for a strongly connected component that contains all accepting conditions (these SCCs are called failure components). (iii) Each ant can backtrack while executing its stochastic search. (iv) We obtain a method to find the shortest counterexample, which is a fundamental well known problem to understand the failure of the Model. (v) The pheromone updating rules tend to give more pheromone to edges contained in interesting SCCs: once a new SCC is found by an ant, a local, delayed, pheromone update rule is applied to the edges belonging to the intermediate SCCs found in the path. The amount of pheromone is proportional to the closeness from the starting node, to the number of possible backtracking moves the ant still has to do, to the number of accepting conditions found in the SCC and to the fraction of those conditions that are novel.

2. Emptiness check over transition-based generalized Büchi automata

2.1. Transition-based generalized Büchi automata

Definition 1. A Transition-Based Generalized Büchi Automaton (briefly TGBA) can be defined as a tuple $\langle \Sigma, \mathcal{Q}, \mathcal{F}, q_0, \delta \rangle$ where $\Sigma = 2^{AP}$ is an alphabet with AP a set of atomic proposition, \mathcal{Q} is the finite set of states, \mathcal{F} is a finite set of accepting conditions, $q_0 \in \mathcal{Q}$ is a distinct initial state and $\delta \subseteq \mathcal{Q} \times (2^{\Sigma} \setminus \{\emptyset\}) \times 2^{\mathcal{F}} \times \mathcal{Q}$ is the transition relation labelled by a set of accepting conditions and a set of letters of Σ .

Let $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{F}, q_0, \delta \rangle$ a TGBA, a *run* of \mathcal{A} is an infinite sequence $\langle q_0, g_0, f_0, q_1 \rangle \dots \langle q_i, g_i, f_i, q_{(i+1)} \rangle \dots$ of transitions of δ , starting with the initial state q_0 . Such a run

Algorithm 1 The emptiness check algorithm

```
1: procedure ACO_EMPTINESS_CHECK
2:
       INITIALIZATION()
3:
       while STOP_CRITERIA_NOT_SATISFIED() do
           ANTS_GENERATION_AND_ACTIVITY()
4:
5:
           DAEMON_ACTIONS()
           PHEROMONE_EVAPORATION()
6:
7:
       end while
8: end procedure
9: procedure ANTS_GENERATION_AND_ACTIVITY
10:
       for i=1,\ldots,m do
           EXPLORE_GRAPH(i, Todo_i, Root_i, Arc_i)
11:
12:
       end for
13: end procedure
14: procedure DAEMON_ACTIONS
       MERGE_COMPONENTS()
15:
16:
       LOCAL_UPDATE()
       GLOBAL_UPDATE()
17:
18: end procedure
   procedure PHEROMONE_EVAPORATION
19:
20:
       \forall q \in \mathcal{Q}, \tau_q \leftarrow \alpha \cdot \tau_q
21: end procedure
   procedure INITIALIZATION
22:
       for i = 1, \ldots, m do
23:
           s_i = 1; Num_i = 0; H[q_0, i] = 1;
24:
           Root_i.PUSH(\langle q_0, Num_i \rangle);
25:
           Arc_i.PUSH(\emptyset);
26:
           Todo_i.PUSH(\langle q_0, \perp, \delta(q_0) \rangle);
27:
       end for
28:
29: end procedure
```

is said to be *accepted* if $\forall f \in \mathcal{F}, \forall i \geq 0, \exists j \geq i$ such that $f \in f_j$, i.e. if its transitions are labelled by each accepting condition infinitely often.

In model checking, classical Büchi automata are often used, they have accepting conditions on state rather than on arcs and often are not generalized (i.e. there is only one accepting condition). Even though, with respect to the process of translating LTL formulæ (i.e. the property to be checked), the benefit of TGBA is already quite clear, the actual problem is that classical emptiness check algorithms are not adapted and usually perform a *de-generalization* that multiplies the size of the automaton by a factor of $|\mathcal{F}|$. Also to our knowledge there are no heuristics proposed to find the shortest counter-example when TGBA are in use. Our algorithm is adapted to TGBA and proposes an optimization heuristic for the well known problem of the *shortest* failure.

For the purpose of finding accepting runs we will not be concerned by Σ , AP and g_i s since they have solely been

introduced in order to have a characterization of the words recognized by the automata.

2.2. Emptiness check

In the automata-theoretic approach to model checking [2] both automata (the one that represent the language of words satisfying the formula and the one of the system) are synchronized, and a key operation is to determine whether the resulting automaton is empty, i.e. contains no accepting run. Such operation is called *emptiness check* (EC). There are two approaches to perform on-the-fly the emptiness-check: nested depth-first search, and algorithms that compute strongly connected component (SCC). A run is accepted if the union of all the acceptance conditions occurring in a non-trivial SCC accessible from q_0 , is \mathcal{F} . This explain the essence of the algorithms of [2, 3, 4, 5]. The goal of the *emptiness check* is to verify if there exist at least one run accepted by the automaton. We present a generalisation of Couvreur's algorithm [2] in algorithm 2.

3. Ant colony heuristic for generalized EC

Ant Colony Optimization (ACO) algorithms form a meta-heuristic for discrete optimization problems [6]. Ant algorithms were first devised in [7], taking inspiration from the foraging behaviour of biological ants and have been proved successful in finding high quality solutions to various NP-hard problems, (e.g. travelling salesman [8], quadratic assignment problem), including dynamical optimization problems (e.g. packet routing in connection-less networks). The main concepts of the ACO meta-heuristic are the employment of a colony of *m* artificial ants and the use of a *stigmergetic* communication model coupled with the use of an *autocatalytic* mechanism for the evaluation of a solution.

Biological ants are able to sense the amount of a volatile chemical released by other ants. While foraging, the amount of pheromone on a path to a source of food is a rough estimate of the closeness of the source from the colony. Thus, shortest paths to food sources will receive more quickly pheromone than the longer ones. Stigmergy is the name of that form of indirect communication, while the reinforcement of shortest paths is called autocatalysis and are the two main explanations of the successfulness of the foraging strategy of biological ants. On the other hand, artificial ants are cognitive problem-solving agents, whose characteristics are inspired by those of real ants and can be enhanced with specific computational capabilities that have no counterpart in the real world; memory or backtracking are two examples.

3.1. Graph exploration by ant colony

The pseudo-code, conforming to a classical structure for the ACO meta-heuristic [9], is shown in algorithm 1. The main exploration algorithm is further shown in algorithm 2. Let m be the number of artificial ants. Our algorithm carries over iteratively two main activities. In an *iteration*, the m ants are fired in a serial way from the starting node in the exploration of the graph. Whenever an ant asks for a successor of the node in which it is, the algorithm produces it on-the-fly computing the synchronized product of the automata of the model and of the property. A fired ant explores the graph with a depth-first visit, choosing successors with a stochastic and greedy transition rule (we'll introduce it formally later) and eventually backtracking if the current node q has no successors, or if all successors have been explored (this is equivalent to say that the DFS subtree rooted in q the ant is building, has been completely explored). To do so, we augment the capabilities of the simple ant by giving it a $Todo_i$ stack in which it can push pairs $(q, succ_q) \in \mathcal{Q} \times 2^{\mathcal{Q}}$ where q is a new node and $succ_q = \{q' \in \mathcal{Q} : \langle q, A, q' \rangle \in \delta\}$ is the set of neighbours of node q (computed on-the-fly). Whenever q' is designated as successor of node q, it is removed from the ant's stacked copy of $succ_q$; thus we directly memorize the feasible neighbourhood of each ant in a stack. The use of a stack is de facto mandatory since our ants can backtrack and thus need to know the order in which each node is visited.

Once fired, each ant explores the graph in search of strongly connected components, associated to sets of accepting conditions. Let $C \subseteq \mathcal{Q}$ be a SCC and consider the restriction of δ to C, then the accepting condition set associated to C is simply the union of all possible accepting conditions labeling edges of C. To identify a SCC, the ant must seek its root, that is the node from which it entered into the component, and thus labelled with the minimum order of visit (with respect to the nodes of the component only, since the starting node of the TGBA always receives order of visit 1). This work can be accomplished with two other stacks, the first called $Root_i$ in which are stored pairs $(r, A) \in \mathbb{N} \times 2^{\mathcal{F}}$, where r is the order of visit of the SCC's root and A is the set of accepting conditions on the arc between nodes belonging to the SCC; the second is called Arc_i and stores sets of accepting conditions labelling the arc between two distinct SCCs. Since the stack $Root_i$ stores only the order of visit for root nodes, a last data structure is needed to store the order of visit of all nodes. This structure is a hash table H, shared among all the m ants. Any entry in H is indexed with a node q of the graph and it contains the orders of visit of q, for every ant. These numbers are arranged in a vector with m components. A newly explored node is root of a trivial component made up of only itself, and with an empty conditions set associated to

it. Exploration in the context of an iteration is stopped as soon an ant find an already visited node q' (lines 21 and 22 of algorithm 2). We pop from Root and from $Arc \triangleq Arc_1$ elements repeteadly, until we find the root of the component q' belongs to; in order to augment the accepting condition set of the component, we merge into the variable new_{acc} the accepting conditions sets that we pop from both Root and Arc during this process (lines 23 to 34 of algorithm 2). In the case with m > 1, the node q' may have been visited by several ants. In general the fact that multiple ants are exploring the same graph, may lead to situations in which the information about the components built by two or more ants can be exchanged among them. For example, consider an ant i that reaches a node q' already visited by ant j, and suppose that q' is in a strongly connected component that ant j already identified. Let's assume that A is the accepting condition set of that component. Since ant i stopped at q', we know that the top of $Root_i$ is (r, B), where r is the order of visit of q' and $B \subset \mathcal{F}$. Then $A \cup B \subseteq \mathcal{F}$, so we can update $Root_i$ with $B \leftarrow A \cup B$ to take advantage of the work of ant j.

Theorem 1. (Soundness, Completeness and Termination) Let t be a step, if it exists an accepting SCC S_{acc} considering all the search paths generated by all ants in t, then the algorithm 2 computes S_{acc} at t. Moreover, all strongly connected components considered taking all possible paths induced by the m ants are computed by algorithm 2.

Proof. In the standard approach, the search is stopped only when a node has already been visited. Our generalisation of that algorithm adapts to the ACO procedure, thus we need to take into account whether a visited node has been visited also by any other ant, and if a SCC is generated by this connection between ant's paths. The proof of the soundness and completeness is described by the finite step of operations performed by Algorithm 2. There are three conditions in which such updates may occur, depending on where an ant stopped. Let us consider ant $i \in [m]$, where $[m] = \{1, \ldots, m\}$:

1. i stopped in q' with order $r_{q'}^i$ and q' has been also previously visited by ant $j \in [m], i \neq j$ with order $r_{q'}^j$; let's say that $Root_j$ contains $n \triangleq |Root_j|$ entries for a respective number of roots of SCCs. Their order of visits are:

$$r_1^j < \ldots < r_n^j \le r_{n+1}^j \triangleq Num_j$$

. Then exists a pair $\langle r_k^j, A^j \rangle$ such that $r_k^j \leq r_{q'}^j < r_{k+1}^j$ for $1 \leq k \leq n_j$. Since i stopped in q', $Root_i.top() = \langle r_{q'}^i, A^i \rangle$, so we update $A^i \leftarrow A^i \cup A^j$. This means that ant i gains the accepting conditions set of the component q' belongs that, from ant j.

2. Ant i stopped in q', with $H[q',j]=-1 \ \forall j \in [m], j \neq i$. Thus the order of visit must be $r_{q'}^i < Num_i$.

This means q' is root of a strongly connected component. Ant i updates both $Root_i$ and Arc_i (lines 23 to 34 of algorithm 2) so that $Root_i.top() = \langle r_{q'}^i, A^i \rangle$. Let us call SCC_i this new component rooted in q', and also assume that $\exists j \in [m]: \exists q \in SCC_i \ s.t. \ H[q,j] > 0$. This means that ant j may have already visited part of SCC_i . For any such j, let

$$\hat{q}^{j} = \arg \left\{ \min_{\substack{q \in SCC_i \\ H[q,j] > 0}} \left\{ H[q,j] \right\} \right\}$$

Then either \hat{q}^j is part of another strongly connected component for ant j, or not. In the last case this means that $Root_j$ contains a pair $\langle r_{\hat{q}^j}, \emptyset \rangle$, then we may update it to $\langle r_{\hat{q}^j}, A^i \rangle$; in the former case there exist in $Root_j$ a pair $\langle r_k^j, A^j \rangle$ such that $r_k^j \leq r_{\hat{q}^j} < r_{k+1}^j$ for $1 \leq k < n_j$: the we can update $A^j \leftarrow A^j \cup A^i$. This means that ant i gives the accepting conditions set of the component q' is root of, to all ants j that visited some nodes of that component.

3. There exist ants $\{i_1,\ldots,i_n\}\subseteq [m]$ such that ant i_1 stopped in node q_1 , with $H[q_1,i_2] > 0$, i_2 stopped in node q_2 , with $H[q_2, i_3] > 0$ and so on until ant i_n , that stopped in q_n with $H[q_n, i_1] > 0$. This means that collectively these ants identified a new strongly connected component, but no one actually closed a loop on its path. We may update $Root_{i_1}, \ldots, Root_{i_n}$ with the accepting conditions set of this new component. The topmost elements of $Root_{i_k}$ are the pairs (from top to bottom): $\langle H[q_k, i_k], A_1 \rangle, \dots, \langle H[q_j, i_k], A_{m_k} \rangle$ where $k = 1, \dots, n$ and $j = i_{k-1 \mod [n]}$, e.g. there are m_k strongly connected components (on the path of ant i_k) between the stop node of the "previous" ant i_j and the stop node of i_k . Similarly the top-most m_k elements of the Arc_{i_k} stack will be B_1, \ldots, B_{m_k} . The accepting condition set B, associated to the strongly connected component that ants i_1, \ldots, i_n identified, can be computed as: $B \triangleq$ $\bigcup_{k=1}^n [(A_1 \cup B_1) \cup \ldots \cup (A_{m_k} \cup B_{m_k})]$. Once computed B, we may update the top element of $Root_{i_k}$, since every ant "sees" the new SCC rooted in its stop node q_k .

4. Pheromone update and state transition rules based on SCC properties

Usually, when a given node q is reached, most algorithms don't impose any sophisticated order on the elements of $succ_q$, or just apply naïve techniques (like exploring first those neighbouring nodes that are already in the hash H, see [10]). In contrast to such approaches, our artificial ants perform a stochastic search; moreover, two kinds of informations are evaluated in order to decide what element of $succ_q$ must be visited next, every time an ant ends up in q either by direct discovery of it or by means of backtracking.

In fact, the probabilistic transition rule we use, provides a direct way to balance between exploitation of knowledge about the problem, accumulated by the colony (τ) , and exploration of new edges, driven by *a priori* knowledge (η) .

The first information comes from the amount of pheromone $\tau_{qq'}$ deposited on the edge $\{q,q'\}$, for any $q' \in succ_q$. This information is built by the whole colony during the whole computation, as a result of the evaluation of the various solutions built by every ant (in our context a solution is just a path from the starting node q_0 to the root of any strongly connected component). The second kind is an heuristic value $\eta_{qq'}$ whose definition corresponds to the intuitive assumption that an ant should prefer edges with a greater number of accepting conditions, to edges with a lesser number.

4.1. Heuristic value and pheromone updates

The heuristic value η associated to a transition defines a priori its attractiveness and is defined as the ratio between the number of accepting conditions on the transition and the overall number of conditions. In this sense we can claim that the exploratory behaviour follows a greedy heuristic, privileging those unexplored nodes with a greater number of accepting conditions.

Definition 2. (*Heuristic*) For any transition $\langle q, A_{qq'}, q' \rangle \in \delta$, the heuristic value $\eta_{qq'}$ associated to it is defined as:

$$\eta_{qq'} = \frac{|A_{qq'}|}{|\mathcal{F}|}$$

In contrast to the costant value $\eta_{qq'}$, the amount of pheromone on the transition $\{q,q'\}$ is a dinamical quantity, e.g. $\tau_{qq'} \triangleq \tau_{qq'}(t)$, where t is the discrete time step over the sequence of *iterations*. The dinamics of τ is a result of two factors: depositation and evaporation; the depositation is due to the local delayed pheromone update, performed by those ants that correctly identify a strongly connected component, and to another form of global pheromone update, performed whenever a group of several ants collectively identifies a SCC (see discussion at point 3.1 in section 3.1). In addition to depositation, we allow a certain amount of pheromone to evaporate from every transition, at the end of every iteration.

Definition 3. (*Pheromone*) Let $t \in \mathbb{N}$ be an iteration step. For any transition $\langle q, A_{qq'}, q' \rangle \in \delta$ the amount of pheromone on it during t is $\tau_{qq'}(t)$ and is defined as follows:

(Initialization, t=0) $au_{qq'}(0)= au_0$ where au_0 is a calibration parameter.

(Local update, $t \geq 0$) We update the amount of pheromone according to an online delayed rule, after every ant has stopped, only for those ants that in the current

Algorithm 2 EXPLORE_GRAPH

```
1: input: i, Todo_i, Root_i, Arc_i
                                                                                                                     if H[q',i] \neq -1 then
                                                                                          23:
 2: output: \top if \mathcal{L}(\mathcal{A}) \neq \emptyset
                                                                                                                          \langle r, acc_r \rangle \leftarrow Root_i.TOP()
                                                                                          24:
 3: while ((Todo_i \neq \emptyset) \land (s_i = 0)) do
                                                                                                                          Root_i.POP()
                                                                                          25:
          \langle q, succ \rangle \leftarrow Todo_i.TOP()
                                                                                                                          new_{acc} \leftarrow acc_{qq'} \cup acc_r
 4:
                                                                                          26:
                                                                                                                          while r > H[q^i, i] do
          if succ = \emptyset then
 5:
                                                                                          27:
               Todo_i.POP()
                                                                                          28:
                                                                                                                               \langle r', acc_{r'} \rangle \leftarrow Root_i.TOP()
 6:
                                                                                                                               new_{acc} \leftarrow new_{acc} \cup acc_{r'} \cup
                \langle r, acc_r \rangle \leftarrow Root_i.TOP()
 7:
                                                                                          29:
 8:
               if (r = H[q, i]) then
                                                                                                Arc_i.TOP()
 9:
                     REMOVE\_COMPONENT(q)
                                                                                          30:
                                                                                                                               Arc_i.POP()
                                                                                                                               Root_i.POP()
                     Root_i.POP()
                                                                                          31:
10:
                                                                                                                               r \leftarrow r'
                     Arc_i.POP()
11:
                                                                                          32:
               else
                                                                                                                          end while
                                                                                          33:
12:
                     \langle acc_{qq'}, q' \rangle \leftarrow succ. \texttt{DECIDE}()
                                                                                                                          Root_i.PUSH(\langle H[q',i], new_{acc} \rangle)
13:
                                                                                          34.
                    if q' \notin H then
                                                                                          35.
                                                                                                                          if F = new_{acc} then
14:
                          H[q',i] \leftarrow Num_i
                                                                                                                               return \top
15:
                                                                                          36:
                          Todo_i.PUSH(\langle q', succ_{q'} \rangle)
                                                                                                                          end if
16:
                                                                                          37:
                          Root_i.PUSH(\langle Num_i, \emptyset \rangle)
                                                                                                                     end if
                                                                                          38:
17:
                                                                                                                     return \perp
18:
                          Arc_i.PUSH(acc_{qq'})
                                                                                          39.
19:
                          Num_i \leftarrow Num_i + 1
                                                                                          40:
                                                                                                               end if
                    else
                                                                                                          end if
20:
                                                                                          41:
                                                                                                     end if
21:
                          s_i \leftarrow 1
                                                                                          42:
                          Stop\_Node[i] \leftarrow q'
                                                                                          43: end while
22:
```

iteration identified a new SCC (e.g. those that stopped to a node, already visited by themselves). For any such ant k, let $S \subseteq \mathcal{Q}$ be the new SCC, then $\tau_{qq'}(t)$ is updated if:

- (i). Both $q \in S$ and $q' \in S$; the arc belongs to the strongly connected component.
- (ii). $q \in S$ and $q' \notin S$; this mean that following the arc would lead an ant outside the component.
- (iii). Both q and q' belong to the path from q_0 to the root of S that ant k followed during its exploration.

Ant k is allowed to deposit a certain amount of pheromone on the transition:

$$\tau_{qq'}(t) \leftarrow \tau_{qq'}(t) + \Delta \tau(S, k, t)$$

with

$$\Delta \tau(S, k, t) = \frac{|A_S|}{|\mathcal{F}|} \cdot \left(\frac{|A_S \setminus \mathcal{F}(t)|}{|\mathcal{F}|}\right)^{\beta} \cdot \left(\frac{1}{R_k}\right)^{\beta'} \cdot \left(\frac{B_k}{|S|}\right)^{\beta''}$$

where |S| is the number of nodes in S, A_S is the set of accepting condition present in S, $\mathcal{F}(t)$ is the set of accepting conditions found up to iteration t by the whole colony, R_k is the length of the path from q_0 to S followed by ant k, e.g. the number of node updated due to condition (iii). B_k is the number of backtracking moves (computed using $Todo_k$)

that ant k has still to do in S, e.g. the number of outgoing arcs it encountered in S that still has to be visited and $\beta, \beta', \beta'' \in [0,1]$ are calibration parameters.

(Global update, $t \ge 0$) Whenever multiple ants identify collectively a SCC associated to the accepting condition set B (see point 3.1 in section 3.1), an amount of pheromone

$$\Delta \tau(S, t) = \frac{|B|}{|\mathcal{F}|} \cdot \left(\frac{|B \setminus \mathcal{F}(t)|}{|\mathcal{F}|}\right)^{\beta}$$

is placed additionally as a daemon action.

(Evaporation, $t \geq 0$) After every ant has deposited pheromone (note that multiple ants may deposit pheromone on the transition, and that conversely a transition may not be updated during step t), and any global update has occurred, a certain amount of pheromone evaporates from $\tau_{qq'}(t)$. The amount left will be used to compute the probabilities for the decision rule in the next iteration.

$$\tau_{qq'}(t+1) \leftarrow \alpha \cdot \tau_{qq'}(t)$$
 with $\alpha \in (0,1].$

4.2. State transition rule depends on accepting conditions

Let ant k be at node q during step t; k decides for the next successor node k in the following manner: the feasible neighborhood of k is stored in the variable $succ_q^k$

in the backtracking $Todo_k$ stack, since $Todo_k.Top() = \langle q, succ_q^k \rangle$. The probability for ant k of choosing node $h \in succ_q^k$ while in node q is given by :

$$P_{q,k}(h) = \begin{cases} \frac{\tau_{qh}(t)}{\displaystyle \sum_{h' \in succ_q^k} \tau_{qh'}(t)} & if \ F_q^k = \emptyset \\ \frac{\tau_{qh}(t) \cdot \eta_{qh}^{\gamma}}{\displaystyle \sum_{h' \in succ_q^k} \tau_{qh'}(t) \cdot \eta_{qh'}^{\gamma}} & otherwise \end{cases}$$

where $\gamma \in (0,1]$ is calibration parameter and $F_q^k \triangleq \{A \in 2^{\mathcal{F}} \setminus \{\emptyset\} : \langle q,A,q' \rangle \in \delta \wedge q' \in succ_q^k \}$. If $F_q^k \neq \emptyset$, then the heuristic value η_{qh} , will be equal to zero for those arcs labeled with an empty set of accepting condition. Thus these arcs will be neglected, in the choice of possible successors of q. This implies that the ants will greedily choose to explore first the transitions labeled with some accepting condition. Contrarily, when there are no transitions labeled with some accepting conditions, the ants will choose the successor node with a probability proportional to the amount of pheromone, thus exploiting the shared knowledge of the problem built so far by the whole colony.

5. Conclusions

In this paper we presented a new algorithm for the emptiness check on transition-based generalized Büchi, a subclass of Büchi automata with many interesting properties that has gained recently attention for the possibility to perform on-the-fly the exhaustive search of the state space.

Our algorithm features many novelties that are not found in other mainstream algorithms for the emptiness check of TGBA, the most notable of which is the use of a fully fledged stochastic search performed by cognitive agents, guided by two types of information. We defined a heuristic greedy strategy that privileges those strongly connected components with a greater number of accepting conditions labelling their arcs, and reachable from the starting state of the automaton by means of a shorter path. This kind of heuristic is not present in other classic algorithms for the emptiness check [10]. The other kind of information is collectively built by the colony and represents the knowledge about the part of the graph explored so far.

Our algorithm features also some interesting aspects as an ACO algorithm, since our artificial ants perform their exploration on a directed graph that's completely built on-thefly from the two digraphs of the automaton of the model $\mathcal M$ of the systems under check, and the automaton of the LTL formula ϕ being checked. This paradigm could well open up the possibility for the application of approximate algorithms to problems of model checking were the goals are

both the exhaustive search on large graphs and the fast convergence on faulty states, whenever they exist. We extend the framework Spot [1] with our algorithm; the experiments and choices of the parameters will be discussed in another paper. The main contribution of our approach is explained by the fact that it is an effective optimization approach to compute the smallest failure. Also, we exploit the fact that any graph contains at least one maximal SCC without any outgoing arc: to list all maximal SCCs one needs to find those terminal SCCs. The effectiveness of our approach is also emphasized by the fact that, compared to any existing approach, the use of artificial ants dramatically improves the time to find those maximal SCCs that are not *accepting component*, and that can thus be removed.

6. References

- [1] A. Duret-Lutz and D. Poitrenaud, "Spot: an extensible model checking library using transition-based generalized Büchi automata," in *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MAS-COTS'04)*, IEEE Computer Society Press, Oct. 2004, pp. 76–83.
- [2] J.-M. Couvreur, "On-the-fly verification of linear temporal logic.," in *World Congress on Formal Methods* (J. M. Wing, J. Woodcock, and J. Davies, eds.), 1999, pp. 253–271.
- [3] J. Geldenhuys and A. Valmari, "Tarjan's algorithm makes on-the-fly LTL verification more efficient," in *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'04), 2004.
- [4] J. Geldenhuys and A. Valmari, "More efficient on-the-fly LTL verification with Tarjan's algorithm," *Theoretical Com*puter Science, vol. 345, Nov. 2005, pp. 60–82.
- puter Science, vol. 345, Nov. 2005, pp. 60–82.
 [5] M. Hammer, A. Knapp, and S. Merz, "Truly on-the-fly LTL model checking," in Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), Apr. 2005.
- [6] M. Dorigo and T. Stützle, Ant Colony Optimization. Cambridge, MA: MIT Press, 2004.
- [7] M. Dorigo, Optimization, Learning and Natural Algorithms (in Italian). PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, 1992.
- [8] M. Dorigo and L. M. Gambardella, "Ant Colony System: A cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, 1997, pp. 53–66.
- [9] M. Dorigo, G. D. Caro, and L. M. Gambardella, "Ant Algorithms for Discrete Optimization," *Journal of Artificial Life*, vol. 5, April 1999, pp. 137–172.
- [10] J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud, "On-the-fly emptiness checks for generalized Büchi automata," in *Proceedings of the 12th International SPIN Workshop on Model Checking of Software* (P. Godefroid, ed.), vol. 3639 of *Lecture Notes in Computer Science*, Springer-Verlag, Aug. 2005, pp. 143–158.