Ant Colony Optimization in Model Checking

Francisco Chicano and Enrique Alba

University of Málaga, Spain,
{chicano, eat}@lcc.uma.es

Abstract. Most of model checkers found in the literature use exact deterministic algorithms to check the properties. The memory required for the verification with these algorithms usually grows in an exponential way with the size of the system to verify. When the search for errors with a low amount of computational resources (memory and time) is a priority (for example, in the first stages of the implementation of a program), non-exhaustive algorithms using heuristic information can be used. In this work we summarize our observations after the application of Ant Colony Optimization to find property violations in concurrent systems using a explicit state model checker. The experimental studies show that ACO finds optimal or near optimal error trails in faulty concurrent systems with a reduced amount of resources, outperforming in most cases the results of algorithms that are widely used in model checking, like Nested Depth First Search. This fact makes ACO suitable for checking properties in large faulty concurrent programs, in which traditional techniques fail to find counterexamples because of the model size.

1 Introduction

Model checking [7] is a fully automatic technique that allows to check if a given concurrent system satisfies a property like, for example, the absence of deadlocks, the absence of starvation, the fulfilment of an invariant, etc. The use of this technique is a must when developing software that controls critical systems, such as an airplane or a spacecraft. However, the memory required for the verification usually grows in an exponential way with the size of the system to verify. This fact is known as the *state explosion problem* and limits the size of the system that a model checker can verify.

When the search for errors with a low amount of computational resources (memory and time) is a priority (e.g., in the first stages of the development), non-exhaustive algorithms using heuristic information can be used. A well-known class of non-exhaustive algorithms for solving complex problems is the class of metaheuristic algorithms [3]. They are search algorithms used in optimization problems that can find good quality solutions in a reasonable time. In this work we summarize the approaches used for the application of one metaheuristic, Ant Colony Optimization (ACO), to the problem of finding property violations in concurrent systems. We also show the results of some experimental studies analyzing the performance of the different approaches.

The paper is organized as follows. The next section presents the background information. Section 3 describes our algorithmic proposals. In Section 4 we present some experimental studies to analyze the performance of our proposals. We also compare our proposals against the most popular algorithms utilized in model checking. Finally, Section 5 outlines the conclusions and future work.

2 Background

In this section we give some details on the way in which properties are checked in explicit state model checking. In particular, we will focus on the model checker HSF-SPIN [9], an experimental model checker by Edelkamp, Lluch-Lafuente and Leue based on the popular model checker SPIN [12]. First, we formally define the concept of *property* of a concurrent system and we detail how the properties are checked. Then, we define the concepts of *strongly connected components* (SCC), *partial order reduction* (POR) and the use of heuristic information.

2.1 Properties and Checking

Let S be the set of possible *states* of a program (concurrent system), S^ω the set of infinite sequences of program states, and S^* the set of finite sequences of program states. The elements of S^ω are called *executions* and the elements of S^* are *partial executions*. However, (partial) executions are not necessarily real (partial) executions of the program. The set of real executions of the program, denoted by M, is a subset of S^ω, that is, $M \subseteq S^\omega$. A *property* P is also a set of executions, $P \subseteq S^\omega$. We say that an execution $\sigma \in S^\omega$ *satisfies* the property P if $\sigma \in P$, and σ *violates* the property if $\sigma \notin P$. In the former case we use the notation $\sigma \vdash P$, and the latter case is denoted with $\sigma \nvdash P$. A property P is a *safety property* if for all executions σ that violate the property there exists a prefix σ_i (partial execution) such that all the extensions of σ_i violate the property. Formally,

$$\forall \sigma \in S^\omega : \sigma \nvdash \mathcal{P} \rightarrow (\exists i \geq 0 : \forall \beta \in S^\omega : \sigma_i \beta \nvdash \mathcal{P}) \enspace , \tag{1}$$

where σ_i is the partial execution composed of the first i states of σ. Some examples of safety properties are the absence of deadlocks and the fulfilment of invariants. On the other hand, a property P is a *liveness property* if for all the partial executions α there exists at least one extension that satisfies the property, that is,

$$\forall \alpha \in S^* : \exists \beta \in S^\omega, \alpha\beta \vdash \mathcal{P} \enspace . \tag{2}$$

One example of liveness property is the absence of starvation. The only property that is a safety and liveness property at the same time is the trivial property $P = S^\omega$. It can be proved that any given property can be expressed as an intersection of a safety and a liveness property [2].

In explicit state model checking the concurrent system M and the property P are represented by finite state ω-automata, $\mathcal{A}(M)$ and $\mathcal{A}(P)$ respectively, that accept those executions they contain. In HSF-SPIN (and SPIN) the automaton $\overline{\mathcal{A}(P)}$, which captures the violations of the property, is called *never claim*. In order to find a violation of a given property, HSF-SPIN explores the intersection (or synchronous product) of the concurrent model and the *never claim*, $\mathcal{A}(M) \cap \overline{\mathcal{A}(P)}$, also called Büchi automaton. HSF-SPIN searches in the Büchi automaton for an execution $\sigma = \alpha\beta^\omega$ composed of a partial execution $\alpha \in S^*$ and a cycle of states $\beta \in S^*$ containing an accepting state. If such an execution is found it violates the liveness component of the property and, thus, the whole property. During the search, it is also possible to find a state in which the end state of the *never claim* is reached (if any). This means that an execution has been found that violates the safety component of the property and the partial execution $\alpha \in S^*$ that leads the model to that state violates the property.

Nested Depth First Search algorithm (NDFS) [11] is the most popular algorithm for performing the search. However, if the property is a safety one (the liveness component is *true*) the problem of finding a property violation is reduced to find a partial execution $\alpha \in S^*$, i.e., it is not required to find an additional cycle containing the accepting state. In this case, classical graph exploration algorithms such as Breadth First Search (BFS), or Depth First Search (DFS) can be used for finding property violations.

2.2 Strongly Connected Components

In order to improve the search for property violations it is possible to take into account the structure of the *never claim*. The idea is based on the fact that a cycle of states in the Büchi automaton entails a cycle in the *never claim* (and in the concurrent system). For improving the search first we need to compute the *strongly connected components* (SCCs) of the *never claim*. Then, we classify the SCCs into three categories depending on the accepting cycles they include. By an N-SCC, we denote an SCC in which no cycle is accepting. A P-SCC is an SCC in which there exists at least one accepting cycle and at least one non-accepting cycle. Finally, a F-SCC is an SCC in which all the cycles are accepting [10].

All the cycles found in the Büchi automaton have an associated cycle in the *never claim*, and, according to the definition of SCC, this cycle is included in one SCC of the *never claim*. Furthermore, if the cycle is accepting (which is the objective of the search) this SCC is necessarily a P-SCC or an F-SCC. The classification of the SCCs of the *never claim* can be used to improve the search for property violations. In particular, the accepting states in an N-SCC can be ignored, and the cycles found inside an F-SCC can be considered as accepting.

2.3 Partial Order Reduction

Partial order reduction (POR) is a method that exploits the commutativity of asynchronous systems in order to reduce the size of the state space. The interleaving model in concurrent systems imposes an arbitrary ordering between concurrent events. When the automaton of the concurrent system is built, the events are interleaved in all possible ways. The ordering between independent concurrent instructions is meaningless. Hence, we can consider just one ordering for checking one given property since the other orderings are equivalent. This fact can be used to construct a reduced state graph hopefully much easier to explore compared to the full state graph (original automaton).

We use here a POR proposal based on *ample sets*. The main idea of ample sets is to explore only a subset of the enabled transitions of each state such that the reduced state space is equivalent to the full state space. This reduction of the state space is performed on-the-fly while the graph is generated.

2.4 Using Heuristic Information

In order to guide the search to the accepting states, a heuristic value is associated to each state of the transition graph of the model. Different kinds of heuristic functions have been defined in the past to better guide exhaustive algorithms. Formula-based heuristics, for example, are based on the expression of the LTL formula checked [9]. Using the logic expression that must be false in an accepting state, these heuristics estimate the number of transitions required to get such an accepting state from the current one. Given a logic formula φ, the heuristic function for that formula H_φ is defined using its subformulae. In this work we use a formula-based heuristic that is defined in [9].

There is another group of heuristic functions called state-based heuristics that can be used when the objective state is known. From this group we can highlight the distance of finite state machines H_{fsm}, in which the heuristic value is computed as the sum of the minimum number of transitions required to reach the objective state from the current one in the local automaton of each process.

3 Algorithmic proposals

In order to find property violations in concurrent systems we proposed in the past an algorithm that we call ACOhg, a new variant of ACO [1]. This algorithm can be used when the property to check is a safety property. In the case of liveness properties we use a different algorithm, called ACOhg-live, that contains ACOhg as a component. We describe ACOhg in the next section and ACOhg-live in Section 3.2.

3.1 ACOhg algorithm

The objective of ACOhg is to find a path from the initial node to one objective node from a set O in a very large exploration graph. We denote with f a function that maps the paths of the graph into real numbers. This function must be designed to reach minimum values when the shortest path to an objective node is found. ACOhg minimizes this objective function. In Algorithm 1 we show the pseudocode of ACOhg.

The algorithm works as follows. At the beginning, the variables are initialized (lines 1-5). All the pheromone trails are initialized with the same value: a random number between τ_0^{min} and τ_0^{max}. In the `init` set (initial nodes for the ants construction), a starting path with only the initial node is inserted (line 1). This way, all the ants of the first stage begin the construction of their path at the initial node.

After the initialization, the algorithm enters in a loop that is executed until a given maximum number of steps (*msteps*) set by the user is performed (line 6). In a loop, each ant builds a path starting in the final node of a previous path

Algorithm 1 ACOhg

```
1: init = {initial_node};
2: next_init = $\emptyset$;
3: $\tau$ = initializePheromone();
4: step = 1;
5: stage = 1;
6: while step $\leq$ msteps do
7:     for k=1 to colsize do {Ant operations}
8:         $a^k = \emptyset$;
9:         $a^k_1$ = selectInitNodeRandomly (init);
10:        while $|a^k| < \lambda_{ant} \wedge T(a^k_*) - a^k \neq \emptyset \wedge a^k_* \notin O$ do
11:            node = selectSuccessor ($a^k_*$, $T(a^k_*)$, $\tau$,$\eta$);
12:            $a^k = a^k$ + node;
13:            $\tau$ = localPheromoneUpdate($\tau$,$\xi$,node);
14:        end while
15:        next_init = selectBestPaths(init, next_init, $a^k$);
16:        if $f(a^k) < f(a^{best})$ then
17:            $a^{best} = a^k$;
18:        end if
19:    end for
20:    $\tau$ = pheromoneEvaporation($\tau$, $\rho$);
21:    $\tau$ = pheromoneUpdate($\tau$, $a^{best}$);
22:    if step $\equiv$ 0 mod $\sigma_s$ then
23:        init = next_init;
24:        next_init = $\emptyset$;
25:        stage = stage+1;
26:        $\tau$ = pheromoneReset();
27:    end if
28:    step = step + 1;
29: end while
```

(line 9). This path is randomly selected from the `init` set. For the construction of the path, the ants enter a loop (lines 10-14) in which each ant k stochastically selects the next node according to the pheromone (τ_{ij}) and the heuristic value (η_{ij}) associated to each arc (a^k_*, j) with $j \in T(a^k_*)$ (line 11). The expression used is the standard random proportional rule used in ACOs [8].

After the movement of an ant from a node to the next one the pheromone trail associated to the arc traversed is updated as in Ant Colony Systems (ACS) [8] using the expression $\tau_{ij} \leftarrow (1-\xi)\tau_{ij}$ (line 13) where ξ, with $0 < \xi < 1$, controls the evaporation of the pheromone during the construction phase. This mechanism increases the exploration of the algorithm, since it reduces the probability that an ant follows the path of a previous ant in the same step. The construction process is iterated until the ant reaches the maximum length λ_{ant}, it finds an objective node, or all the successors of the last node of the current path, $T(a^k_*)$, have been visited by the ant during the construction phase. This last condition prevents the ants from constructing cycles in their paths.

After the construction phase, the ant is used to update the `next_init` set (line 15), which will be the `init` set in the next stage. In `next_init`, only starting paths are allowed and all the paths must have different last nodes. This rule is ensured by `selectBestPaths`. The cardinality of `next_init` is bounded by a given parameter ι. When this limit is reached and a new path must be included in the set, the starting path with higher objective value is removed from the set.

When all the ants have built their paths, a pheromone update phase is performed. First, all the pheromone trails are reduced according to the expression $\tau_{ij} \leftarrow (1-\rho)\tau_{ij}$ (line 20), where ρ is the *pheromone evaporation rate* and it holds that $0 < \rho \leq 1$. Then, the pheromone trails associated to the arcs traversed by the best-so-far ant (a^{best}) are increased using the expression $\tau_{ij} \leftarrow \tau_{ij} + 1/f(a^{best})$, $\forall (i,j) \in a^{best}$ (line 21). This way, the best path found is awarded with an extra amount of pheromone and the ants will follow that path with higher probability in the next step. We use here the mechanism introduced in Max-Min Ant Systems ($\mathcal{MM}$AS) [8] for keeping the value of pheromone trails in a given interval $[\tau_{min}, \tau_{max}]$ in order to maintain the probability of selecting one node above a given threshold. The values of the trail limits are $\tau_{max} = 1/\rho f(a^{best})$ and $\tau_{min} = \tau_{max}/a$ where the parameter a controls the size of the interval.

Finally, with a frequency of σ_s steps, a new stage starts. The `init` set is replaced by `next_init` and all the pheromone trails are removed from memory (lines 22-27). In addition to the pheromone trails, the arcs to which the removed pheromone trails are associated are also discarded (unless they also belong to a path in *next_init*). This removing step allows the algorithm to reduce the amount of memory required to a minimum value. This minimum amount of memory is the one utilized for storing the best paths found in one stage (the *next_init* set).

3.2 ACOhg-live

In this section we present ACOhg-live, an algorithm based on ACOhg for searching for general property violations in concurrent systems. In Algorithm 2 we show a high level object oriented pseudocode of ACOhg-live. We assume that `acohg1` and `acohg2` are two instances of a class implementing ACOhg.

The search that ACOhg-live performs is composed of two different phases. In the first one, ACOhg is used for finding accepting states in the Büchi automaton (line 2 in Algorithm 2). In this phase, the search of ACOhg starts in the initial node of the graph q and the set of objective nodes O is empty. That is, although the algorithm searches for accepting states, there is no preference on a specific set of them. If the algorithm finds accepting states, in a second phase a new search is performed using ACOhg again for each accepting state discovered (lines 3 to 8). In this second search the objective is to find a cycle involving the accepting state. The search starts in one accepting state and the algorithm searches for the same state in order to find a cycle. That is, the initial node of the search and the only objective node are the same: the accepting state. If a cycle is found ACOhg-live returns the complete accepting path (line 6). If no cycle is found for any of the accepting states ACOhg-live runs again the first phase after including the accepting states in a tabu list (line 9). This tabu list prevents the algorithm from searching again cycles containing the just explored accepting states. If one of the accepting states in the tabu list is reached it will not be included in the list of accepting states to be explored in the second phase. ACOhg-live alternates between the two phases until no accepting state is found in the first one (line 10).

The algorithm can also stop its search due to another reason: an end state has been found. That is, when an end state is found either in the first or the second phase of the search the algorithm stops and returns the path from the initial

Algorithm 2 ACOhg-live

```
repeat
    accpt = acohg1.findAcceptingStates(); {First phase}
    for node in accpt do
        acohg2.findCycle(node); {Second phase}
        if acohg2.cycleFound() then
            return acohg2.acceptingPath();
        end if
    end for
    acohg1.insertTabu(accpt);
until empty(accpt)
return null;
```

state to that end state. If this happens, an execution of the concurrent system has been found that violates the safety component of the checked property.

When the property to check is the absence of deadlocks only the first phase of the search is required. In this case, ACOhg-live searches for deadlock states (states with no successors) instead of accepting states. When a deadlock state is found the algorithm stops returning the path from the initial state to that deadlock state. The second phase of the search, the objective of which is to find an accepting cycle, is never run in this situation.

Now we are going to give the details of the ACOhg algorithms used inside ACOhg-live. First of all, we use a node-based pheromone model, that is, the pheromone trails are associated to the nodes instead of the arcs. This means that all the values τ_{xj} associated to the arcs which head is node j are in fact the same value and is associated to node j. The heuristic values η_{ij} are defined after the heuristic function H using the expression $\eta_{ij} = 1/(1 + H(j))$. This way, η_{ij} increases when $H(j)$ decreases (high preference to explore node j).

Finally, the objective function f to be minimized is defined as

$$f(a^k) = \begin{cases} |\pi + a^k| \text{ if } a^k_* \in O \\ |\pi + a^k| + H(a^k_*) + p_p + p_c \frac{\lambda_{ant} - |a^k|}{\lambda_{ant} - 1} \text{ if } a^k_* \notin O \end{cases}, \tag{3}$$

where π is the starting path in `init` whose last node is the first one of a^k, p_p, and p_c are penalty values that are added when the ant does not end in an objective node and when a^k contains a cycle, respectively. The last term in the second row of Eq. (3) makes the penalty higher in shorter cycles (see [4] for more details).

4 Experimental studies

In this section we present some experimental studies aimed at analyzing the performance of our ACO proposals for the problem of finding property violations in concurrent systems. In the following section we present the Promela models used in the experimentation. Then we show the results of four different analyses: two of them related to the violation of safety properties and the other two related to liveness properties. In all the cases, 100 independent runs of the ACOhg algorithms are performed and the average and the standard deviation are shown.

4.1 Models

In the empirical studies we used nine Promela models, some of them scalable. In Table 1 we present the models with some information about them. They can be found in `oplink.lcc.uma.es` together with the HSF-SPIN and ACOhg source code. In the table we also show the safety and liveness properties that we check in the models.

Table 1. Promela models used in the experiments

Model	LoC	Processes	Safety property	Liveness property
phi$j$	57	$j+1$	deadlock	$\Box(p \rightarrow \Diamond q)$
giopi, j	740	$i+3(j+1)$	deadlock	$\Box(p \rightarrow \Diamond q)$
marriers$j$	142	$j+1$	deadlock	
leader$j$	178	$j+1$	assertion	
needham	260	4	LTL formula	
pots	453	8	deadlock	
alter	64	2		$\Box(p \rightarrow \Diamond q) \wedge \Box(r \rightarrow \Diamond s)$
elev$j$	191	$j+3$		$\Box(p \rightarrow \Diamond q)$
sgc	1001	20		$\Diamond p$

4.2 Safety properties

In this section we compare the results obtained with ACOhg for safety properties against the ones obtained with exact algorithms previously found in the literature. These algorithms are Breadth First Search (BFS), Depth First Search (DFS), A*, and Best First Search (BF). BFS and DFS do not use heuristic information while the other two do. In order to make a fair comparison we use two different ACOhg algorithms: one not using heuristic information (ACOhg-b) and another one using it (ACOhg-h). We show the results of all the algorithms in Table 2. In the table we can see the hit rate (number of executions that got an error trail), the length of the error trails found (number of states), the memory required (in Kilobytes), and the CPU time used (in milliseconds) by each algorithm. We highlight with a grey background the best results (maximum values for hit rate and minimum values for the rest of the measures). For ACOhg-b and ACOhg-h we omit here the standard deviation due to room problems. The parameters used in the ACOhg algorithms are the ones of [1].

In general terms, we can state that ACOhg-b is a robust algorithm that is able to find errors in all the proposed models with a low amount of memory. In addition, it combines the two good features of BFS and DFS: it obtains short error trails, like BFS, while at the same time requires a reduced CPU time, like DFS. Regarding the algorithms using heuristic information, we can state that ACOhg-h is the best trade-off between solution quality and memory required: it obtains almost optimal solutions with a reduced amount of memory.

4.3 Influence of POR

In this section we are going to analyze how the combination of partial order reduction plus ACOhg can help in the search for safety property violations in concurrent models. In Table 3 we present the results of applying ACOhg and ACOhg^{POR} to nine models: three instances of `giop`, `marriers`, and `leader`. The hit rate is always 100 %, and for this reason we omit it. In order to clarify that

Table 2. Results of ACOhg-b and ACOhg-h against the exhaustive algorithms.

Model	Measure	BFS	DFS	ACOhg-b	A*	BF	ACOhg-h
giop2,2	Hit rate	0/1	1/1	100/100	1/1	1/1	100/100
	Length	-	112.00	45.80	44.00	44.00	44.20
	Mem. (KB)	-	3945.00	4814.12	417792.00	2873.00	4482.12
	Time (ms)	-	30.00	113.60	46440.00	10.00	112.40
marriers4	Hit rate	0/1	0/1	57/100	0/1	1/1	84/100
	Length	-	-	92.18	-	108.00	86.65
	Mem. (KB)	-	-	5917.91	-	41980.00	5811.43
	Time (ms)	-	-	257.19	-	190.00	233.33
needham	Hit rate	1/1	1/1	100/100	1/1	1/1	100/100
	Length	5.00	11.00	6.39	5.00	10.00	6.12
	Mem. (KB)	23552.00	62464.00	5026.36	19456.00	4149.00	4865.40
	Time (ms)	1110.00	18880.00	262.00	810.00	20.00	229.50
phi16	Hit rate	0/1	0/1	100/100	1/1	1/1	100/100
	Length	-	-	31.44	17.00	81.00	23.08
	Mem. (KB)	-	-	10905.60	2881.00	10240.00	10680.32
	Time (ms)	-	-	289.40	10.00	40.00	243.80
pots	Hit rate	1/1	1/1	49/100	1/1	1/1	99/100
	Length	5.00	14.00	5.73	5.00	7.00	5.44
	Mem. (KB)	57344.00	12288.00	9304.67	57344.00	6389.00	6974.56
	Time (ms)	4190.00	140.00	441.63	6640.00	50.00	319.49

the reduced amount of memory required by the ACOhg algorithms is not due to the use of the heuristic information, we also show the results obtained with A* for all the models using the same heuristic functions as the ACOhg algorithms. This clearly states that memory reduction is a very appealing attribute of ACOhg itself.

Table 3. Comparison among ACOhg, ACOhg^{POR} and A*.

Model	Measure	ACOhg		ACOhg^{POR}		A*
`giop2,1`	Length	42.30	1.71	42.10	0.99	42.00
	Mem. (KB)	3428.44	134.95	2979.48	98.33	27648.00
	Time (ms)	202.00	9.06	162.50	5.55	1000.00
`giop4,1`	Length	70.21	7.56	59.76	5.79	-
	Mem. (KB)	9523.67	331.76	7420.08	422.94	-
	Time (ms)	354.50	42.39	264.90	40.46	-
`giop6,1`	Length	67.59	13.43	61.74	3.16	-
	Mem. (KB)	11970.56	473.59	11591.68	477.67	-
	Time (ms)	440.60	71.02	391.70	43.86	-
`leader6`	Length	50.90	4.52	56.36	3.04	37.00
	Mem. (KB)	16005.12	494.39	3710.64	410.29	132096.00
	Time (ms)	494.00	21.12	98.80	8.16	1250.00
`leader8`	Length	60.83	4.66	74.11	4.51	-
	Mem. (KB)	24381.44	515.98	4831.40	114.10	-
	Time (ms)	1061.20	211.47	198.90	4.67	-
`leader10`	Length	73.84	4.79	80.86	6.36	-
	Mem. (KB)	30167.04	586.82	7178.05	2225.78	-
	Time (ms)	1910.70	45.02	294.90	66.96	-
`marriers10`	Length	307.11	34.87	233.19	21.91	-
	Mem. (KB)	34170.88	494.39	18319.36	804.93	-
	Time (ms)	8847.00	634.06	1306.60	126.56	-
`marriers15`	Length	540.41	60.88	395.10	40.07	-
	Mem. (KB)	51148.80	223.18	26050.56	1256.81	-
	Time (ms)	19740.50	1935.54	3595.00	316.59	-
`marriers20`	Length	793.62	80.45	569.99	54.63	-
	Mem. (KB)	68003.84	503.64	33351.68	1442.75	-
	Time (ms)	49446.30	7557.40	8174.00	707.71	-

From the results in the table we conclude that the memory required by ACOhg^{POR} is always smaller than the one required by ACOhg. The length of the error paths is smaller for ACOhg^{POR} in six out of the nine models. Finally, the CPU time required by ACOhg^{POR} is up to 6.8 times lower (in `marriers10`) than the time required by ACOhg. Although it is not our objective to optimize the length of the error paths in this work, we can say that, in six out of the nine models, the length of the error paths obtained by ACOhg^{POR} is shorter than the one obtained by ACOhg. We finally also remind that other popular algorithm like A* cannot even be applied to most of these instances (only `giop2,1` and `leader6` can be tackled with A*), and thus we are investigating in a new frontier of high dimension models usually not found in literature.

4.4 Liveness Results

In the next experiment we compare the results obtained with ACOhg-live against the classical algorithm utilized for finding liveness errors in concurrent systems: Nested-DFS. This last algorithm is deterministic and for this reason we only perform one single run. In Table 4 we show the results of both algorithms. We also show the results of a statistical test (with level of significance $\alpha = 0.05$) in order to check if there exist statistically significant differences (last column). A plus sign means that the difference is significant and a minus sign means that it is not. For more details on the experiments see [5].

Table 4. Comparison between ACOhg-live and Nested-DFS

Model	Measure	ACOhg-live		Nested-DFS	Test
`alter`	Hit rate	100/100		1/1	-
	Length	30.68	10.72	64.00	+
	Mem. (KB)	1925.00	0.00	1873.00	+
	Time (ms)	90.00	13.86	0.00	+
`giop2,2`	Hit rate	100/100		1/1	-
	Length	43.76	5.82	298.00	+
	Mem. (KB)	2953.76	327.48	7865.00	+
	Time (ms)	747.50	408.09	240.00	+
`giop6,2`	Hit rate	100/100		0/1	+
	Length	58.77	7.21	●	●
	Mem. (KB)	5588.04	631.36	●	●
	Time (ms)	8733.50	3304.90	●	●
`giop10,2`	Hit rate	86/100		0/1	+
	Length	62.85	7.03	●	●
	Mem. (KB)	9316.67	700.44	●	●
	Time (ms)	43059.07	21417.74	●	●
`phi8`	Hit rate	100/100		1/1	-
	Length	51.36	6.95	3405.00	+
	Mem. (KB)	2014.32	18.87	4005.00	+
	Time (ms)	2126.10	479.64	40.00	+
`phi14`	Hit rate	99/100		1/1	-
	Length	76.05	9.35	10001.00	+
	Mem. (KB)	2496.07	41.81	59392.00	+
	Time (ms)	8070.30	1530.12	2300.00	+
`phi20`	Hit rate	98/100		1/1	-
	Length	97.39	10.14	10001.00	+
	Mem. (KB)	3244.67	91.33	392192.00	+
	Time (ms)	18064.90	5538.30	17460.00	-

The first observation concerning the hit rate is that ACOhg-live is the only one that is able to find error paths in all the models. Nested-DFS is not able to find error paths in `giop6,2` and `giop10,2` because it requires more than the memory available in the machine used for the experiments (512 MB). With respect to the length of the error paths we observe that ACOhg-live obtains shorter error executions than Nested-DFS in all the models (with statistical significance). If we focus on the computational resources we observe that ACOhg-live requires less memory than Nested-DFS to find the error paths with the only exception of `alter`. The biggest differences are those of `giop6,2` and `giop10,2` in which Nested-DFS requires more than 512 MB of memory while ACOhg-live obtains error paths with 38 MB at most. With respect to the time required for the search, Nested-DFS is faster than ACOhg-live. The mechanisms included in ACOhg-live in order to be able to find short error paths with high hit rate and low amount of memory extend the time required for the search. Anyway, the maximum difference with respect to the time is around six seconds (in `phi14`), which is not too much if we take into account that the error path obtained is much shorter.

4.5 Influence of the SCC improvement

In this final study we compare two versions of the ACOhg-live algorithm: one of them using the SCC improvement (called ACOhg-live$^+$ in the following) and the other one without that improvement (called ACOhg-live$^-$). With this experiment we want to analyze the influence on the results of the SCC improvement. All the properties checked in the experiments have at least one F-SCC in the never claim; none of them has a P-SCC; and all except `sgc` have exactly one N-SCC. In Table 5 we show the results. For more details on the experiments see [6].

Table 5. Influence of the SCC improvement

Model	Measure	ACohg-live$^-$		ACOhg-live$^+$		T	Model	ACohg-live$^-$		ACOhg-live$^+$		T
giop10,2	Hit rate		84/100		89/100	-	elev10		100/100		100/100	-
	Length	68.57	5.29	67.60	6.09	-		126.56	18.32	127.76	16.89	-
	Mem. (KB)	6375.90	542.50	5098.75	1580.90	+		2617.60	7.93	2617.04	9.72	-
	Time (ms)	7816.55	4779.41	935.84	1009.74	+		2577.30	2258.38	2372.90	1963.04	-
giop15,2	Hit rate		46/100		57/100	-	elev15		100/100		100/100	-
	Length	81.26	3.64	78.30	6.49	+		182.02	9.75	180.04	16.83	-
	Mem. (KB)	9001.17	483.22	8538.54	1610.63	-		3163.56	10.98	3164.64	13.44	-
	Time (ms)	11725.65	7307.22	2016.84	1254.88	+		2683.00	3274.20	2812.10	3540.73	-
giop20,2	Hit rate		14/100		30/100	+	elev20		100/100		100/100	-
	Length	93.29	2.08	88.47	4.72	+		233.00	0.00	231.62	13.73	-
	Mem. (KB)	11132.71	894.26	10403.17	1920.50	-		3716.44	13.15	3716.92	11.29	-
	Time (ms)	11360.00	4564.72	2575.33	1103.46	+		3900.60	7141.02	3034.00	4709.07	-
phi20	Hit rate		98/100		97/100	-	alter		100/100		100/100	-
	Length	88.29	6.91	108.73	10.08	+		10.00	0.00	15.82	6.74	+
	Mem. (KB)	3398.63	34.05	3385.04	63.41	-		1929.00	0.00	1929.00	0.00	-
	Time (ms)	5162.04	645.64	851.75	1462.71	+		241.80	59.35	10.40	3.98	+
phi30	Hit rate		94/100		95/100	-	sgc		32/100		100/100	+
	Length	122.60	9.58	139.15	9.06	+		24.00	0.00	24.00	0.00	-
	Mem. (KB)	5146.62	44.70	5148.12	57.48	-		2699.00	23.13	2285.00	0.00	+
	Time (ms)	10980.64	2156.73	2701.79	3876.34	+		575191.88	62021.86	710.20	48.58	+
phi40	Hit rate		77/100		81/100	-						
	Length	154.74	9.74	166.83	9.44	+						
	Mem. (KB)	7573.68	66.50	7545.35	81.04	+						
	Time (ms)	20422.60	5795.93	5807.41	7588.17	+						

From the results in Table 5, we conclude that the use of the SCC improvement increases the hit rate and decreases the computational resources required for the search. The length of error paths could be slightly increased depending on the particular model.

5 Conclusions and Future Work

In this paper we summarize our observations using ACO algorithms for the problem of searching for property violations in concurrent systems. The numerical results shown here are an excerpt from the research work performed during the last two years on this topic. From the results we conclude that ACO algorithms are promising for the model checking domain. They can find short error trails using a low amount of computational resources.

At present, we are investigating how other metaheuristic algorithms perform on this problem. We are also working on new heuristic functions that can guide the search in a better way. As future work, we plan to design and develop new models of parallel ACO algorithms in order to profit from the computational power of a cluster or a grid of computers.

6 Acknowledgements

This work has been partially funded by the Spanish Ministry of Science and Innovation and FEDER under contract TIN2008-06491-C04-01 (the M* project). It has also been partially funded by the Andalusian Government under contract P07-TIC-03044 (DIRICOM project).

References

1. Enrique Alba and Francisco Chicano. Finding safety errors with ACO. In *Proc. of GECCO*, pages 1066–1073, 2007.
2. Bowen Alpern and Fred B. Schneider. Defining liveness. *Inform. Proc. Letters*, 21:181–185, 1985.
3. C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.
4. Francisco Chicano and Enrique Alba. Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models. *Information Processing Letters*, 106(6):221–231, June 2008.
5. Francisco Chicano and Enrique Alba. Finding liveness errors with ACO. In *Proceedings of the World Conference on Computational Intelligence*, pages 3002–3009, Hong Kong, China, 2008.
6. Francisco Chicano and Enrique Alba. Searching for liveness property violations in concurrent systems with ACO. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 1727–1734, Atlanta, USA, 2008.
7. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, January 2000.
8. Marco Dorigo and Thomas Stützle. *Ant Colony Optimization.* The MIT Press, 2004.
9. Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Protocol Verification with Heuristic Search. In *AAAI-Spring Symposium on Model-based Validation Intelligence*, pages 75–83, 2001.
10. Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Intl. Jnl. of Soft. Tools for Tech. Transfer*, 5:247–267, 2004.
11. G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32, 1996.
12. Gerald J. Holzmann. *The SPIN Model Checker.* Addison-Wesley, 2004.