

Finding Liveness Errors with ACO

Francisco Chicano and Enrique Alba

Abstract—Model Checking is a well-known and fully automatic technique for checking software properties, usually given as temporal logic formulae on the program variables. Most of model checkers found in the literature use exact deterministic algorithms to check the properties. These algorithms usually require huge amounts of memory if the checked model is large. We propose here the use of an algorithm based on ACOhg, a new kind of Ant Colony Optimization model, to search for liveness property violations in concurrent systems. This algorithm has been previously applied to the search for safety errors with very good results and we apply it here for the first time to liveness errors. The results state that our algorithmic proposal, called ACOhg-live, is able to obtain very short error trails in faulty concurrent systems using a low amount of resources, outperforming by far the results of Nested-DFS, the traditional algorithm used for this task in the model checking community and implemented in most of the explicit state model checkers. This fact makes ACOhg-live a very suitable algorithm for finding liveness errors in large faulty concurrent systems, in which traditional techniques fail because of the model size.

I. INTRODUCTION

From the very beginning of computer research, computer engineers have been interested in techniques allowing them to know if a software module fulfils a set of requirements (its specification). These techniques are especially important in critical software, such as airplane or spacecraft controllers, in which people’s lives depend on the software system; or even bank information systems, in which a great amount of money can be lost due to software errors. In addition, modern non-critical software is very complex and these techniques have become a necessity in most software companies. One of these techniques is *formal verification*, in which some properties of the software can be checked much like a mathematical theorem defined on the source code. A very well-known logic used in software verification is *Hoare logic* [1]. However, formal verification using logics is not fully automatic. Although automatic theorem provers can assist the process, human intervention is still needed.

Model checking [2] is another well-known and fully automatic formal method. In this case all the possible program states are analyzed (in an explicit or implicit way) in order to prove (or refute) that the program satisfies a given property such as absence of deadlocks or starvation. Some other more general properties can be specified using a temporal logic like Linear Temporal Logic (LTL) [3] or Computation Tree Logic (CTL) [4]. When the property is specified using LTL,

model checkers transform the model and the negation of the LTL formula into Büchi automata¹ in order to perform their intersection. The intersection Büchi automaton is explored to search for a cycle of states containing an accepting state reachable from the initial state. If such a cycle is found, then there exists at least one execution of the system not fulfilling the LTL property (see [5] for more details). If such kind of cycle does not exist then the system fulfils the property and the verification ends with success.

The amount of states of the transition graph or the intersection automaton associated to a concurrent model is very large even in the case of small systems, and it increases exponentially with the size of the model. This fact is known as *the state explosion problem* and limits the size of the model that an explicit state model checker can verify. This limit is reached when the model checker is not able to explore more states due to the absence of free computer memory. Several techniques exist to alleviate this problem. They reduce the amount of memory required for the search by following different approaches. On the one hand, there are techniques which reduce the number of states to explore, such as partial order reduction [6] or symmetry reduction [7]. On the other hand, we find techniques that reduce the memory required for storing one state, such as state compression, minimal automaton representation of reachable states, and bitstate hashing [5]. Symbolic model checking [8] is another very popular alternative to the explicit state model checking which can reduce the amount of memory required for the verification. In this case, a set of states is represented by a finite propositional formula. However, exhaustive search techniques are always handicapped in real concurrent programs because most of these programs are too complex even for the most advanced techniques. Therefore, techniques of bounded (low) complexity as metaheuristics will be needed for medium/large size programs working in real world scenarios.

In this work we propose an algorithm based on a new model² of Ant Colony Optimization (ACO) called ACOhg for searching for liveness property violations in concurrent systems. This paper follows the research line opened in [9] and extends the approach utilized for the search for safety property violations in [10] and [11] to liveness properties. An additional contribution of this work is the application of metaheuristic algorithms to the search for liveness property violations in concurrent systems. The search for safety

Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, (email: {chicano, eat}@lcc.uma.es).

This work has been partially funded by the Spanish Ministry of Education and Science and FEDER under contract TIN2005-08818-C04-01 (the OPLINK project). It has also been partially funded by the Spanish Ministry of Industry under contract FIT-330225-2007-1 (the European EUREKA-CELTIC project CARLINK)

¹Other formal structures used in explicit state model checking are transition systems and Kripke structures.

²Along this work we use the word “model” to refer to two different concepts: software models and Ant Colony Optimization models. The meaning is clarified in each case by the context in which the word appears.

property violations has been tackled using metaheuristics in the past (in particular, genetic algorithms have been used in [12]) but, to the best of our knowledge, no metaheuristic algorithm has been applied to the search for liveness errors in concurrent systems.

The paper is organized as follows. The next section presents the foundations of the problem. In Section III the problem is formalized as a graph search. Section IV describes our algorithmic proposal, ACOhg-live, for tackling the problem and the metaheuristic algorithm in which it is based: ACOhg. In Section V we present some experimental results comparing two versions of ACOhg-live using different heuristic functions for guiding the search. After that, we compare the results of ACOhg-live against the traditional algorithm utilized for checking liveness properties in explicit state model checking: Nested-DFS. Finally, Section VI presents a discussion on the applicability of our proposal and Section VII outlines the conclusions and future work.

II. BACKGROUND

The properties that can be specified with LTL formulae can be classified into two groups: *safety* and *liveness* properties [13]. Safety properties are those in which a finite execution can be a counterexample, while liveness properties can only be violated by infinite executions (for a formal definition of safety and liveness see [14]). Safety properties can be checked by searching for a single accepting state in the intersection Büchi automaton of the concurrent model and the negation of the LTL formula. That is, when safety properties are checked, it is not required to find an additional cycle containing the accepting state. This means that safety property verification can be transformed into the search for one objective node (one accepting state) in a graph (intersection Büchi automaton) and general graph exploration algorithms like DFS and BFS can be applied to the problem. Furthermore, in [15] and [16] the authors utilize heuristic information for guiding the search. They assign a heuristic value to each state that depends on the safety property to verify. After that, they utilize classical algorithms for graph exploration such as A*, Weighted A* (WA*), Iterative Deepening A* (IDA*), and Best First Search (BF). The results show that, by using heuristic search, the length of the counterexamples can be shortened (they can find optimal error trails using A* and BFS) and the amount of memory required to obtain an error trail is reduced, allowing the exploration of larger models.

The utilization of heuristic information for guiding the search for errors in model checking is known as *heuristic* or *directed model checking*. The heuristics are designed to lead the exploration first to the region of the state space in which an error is likely to be found. This way, the time and memory required to find an error in faulty concurrent systems is reduced in average. However, no benefit from heuristics is obtained when the goal is to verify that a given program fulfils a given property. In this case, all the state space must be exhaustively explored.

The search for liveness errors requires to find an accepting state and a cycle involving the accepting state. In this case, the traditional algorithm applied is Nested-DFS [17]. However, some improvements on Nested-DFS have been proposed in the literature [16], [18]. Anyway, all the algorithms utilized for discovering liveness errors have been exhaustive algorithms, which require a large amount of memory even in small concurrent models.

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. One example of this class of algorithms is *Beam-search*, included in the Java PathFinder model checker [19], [20]. Non-exhaustive algorithms can find errors in programs using less computational resources than exhaustive algorithms (as we will see in this paper), but they cannot be used for verifying a property: when no error is found using a non-exhaustive algorithm we still cannot ensure that no error exists. Due to this fact we can establish some similarities between heuristic model checking using non-exhaustive algorithms and software testing [21]. In both cases, a large region of the state space of the program is explored in order to discover errors; but the absence of errors does not imply the correctness of the program. This relationship between model checking and software testing has been used in the past for generating test cases using model checkers [22].

A well-known class of non-exhaustive algorithms for solving complex problems is the class of metaheuristic algorithms [23]. They are search algorithms used in optimization problems that can find good quality solutions in a reasonable time. The search for accepting states in the Büchi automaton can be translated into an optimization problem and, thus, metaheuristic algorithms can be applied to the search for safety errors. In fact, Genetic Algorithms (GAs), have been applied in the past [12]. The search for liveness errors is not so direct and, in fact, no metaheuristic algorithm has been applied to it in the literature to the best of our knowledge.

Unlike GA, ACO is a metaheuristic designed for searching short paths in graphs. This makes it very suitable for the problem at hand and for this reason our proposal is based on ACO. In order to guide the search we use some of the heuristic functions defined in [18]. In fact, we have extended their experimental model checker, HSF-SPIN, in order to include our ACOhg-live algorithm. In this way, we can use all the heuristic functions implemented in HSF-SPIN and, at the same time, all the existing work related to parsing Promela models and interpreting them.

A. Using Heuristic Information

In order to guide the search, 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. In [20] structural heuristics are introduced that attempt to explore the structure of a program in a way conducive to find general errors. One example of this kind of heuristic information is code coverage, a well known metric in the

software testing domain. Another example of this kind of heuristics is thread interleaving, in which states yielding a thread scheduling with many context changes are rewarded.

Unlike structural heuristics, property-specific heuristics [20] rely on features of the particular property checked. Formula-based heuristics, for example, are based on the expression of the LTL formula checked [15]. These heuristics estimate the number of transitions required to get such an objective node 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 [15].

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 Hamming distance H_{ham} and the distance of finite state machines H_{fsm} . In the first case, the heuristic value is computed as the Hamming distance between the binary representations of the current and the objective state. In the latter, the heuristic value is the sum of the minimum number of transitions required to reach the objective state from the current one in the local automata of each process. Both heuristics will be used and compared in the experimental section.

III. PROBLEM FORMALIZATION

We tackle in this article the problem of searching for liveness property violations in concurrent systems. As we have detailed in the previous section, this problem can be translated into the search of a path in a graph (the intersection Büchi automaton) starting in the initial state and ending in an objective node (accepting state) and an additional cycle involving the accepting state. We formalize here the problem as follows.

Let $G = (S, T)$ be a directed graph where S is the set of nodes and $T \subseteq S \times S$ is the set of arcs. Let $q \in S$ be the *initial node* of the graph and $F \subseteq S$ a set of distinguished nodes that we call *objective nodes*. We denote with $T(s)$ the successors of node s . A finite path over the graph is a sequence of nodes $\pi = s_1 s_2 \dots s_n$ where $s_i \in S$ for $i = 1, 2, \dots, n$ and $s_i \in T(s_{i-1})$ for $i = 2, \dots, n$. We denote with π_i the i th node of the sequence and we use $|\pi|$ to refer to the length of the path, that is, the number of nodes of π . We say that a path π is a *starting path* if the first node of the path is the initial node of the graph, that is, $\pi_1 = q$. We will use π_* to refer to the last node of the sequence π , that is, $\pi_* = \pi_{|\pi|}$. We say that a path π is a cycle if the first and the last nodes of the path are the same, that is, $\pi_1 = \pi_*$.

Given a directed graph G , the problem at hands consists in finding a starting path π ending in an objective node and a cycle ν containing the objective node. That is, find π and ν subject to $\pi_1 = q \wedge \pi_* \in F \wedge \pi_* = \nu_1 = \nu_*$.

The graph G used in the problem is derived from the intersection Büchi automaton B of the model and the negation of the LTL formula of the property. The set of nodes S in G is the set of states in B , the set of arcs T in G is the set of transitions in B , the initial node q in G is the initial state in B , and the set of objective nodes F in G is the set

of accepting states in B . In the following, we will also use the words *state*, *transition*, and *accepting state* to refer to the elements in S , T , and F , respectively.

IV. ALGORITHMIC PROPOSAL: ACOHG-LIVE

In order to find accepting paths in the Büchi automaton we propose here an algorithm that we call ACOhg-live. This algorithm is based on ACOhg, a new kind of ACO that has been applied to the search for safety errors in concurrent systems. We describe ACOhg at the end of this section. In Algorithm 1 we show a high level object oriented pseudocode of ACOhg-live. We assume that `acohg1` and `acohg2` are two instances of a class implementing the ACOhg algorithm.

Algorithm 1 ACOhg-live

```

1: repeat
2:   acpt = acohg1.findAcceptingStates(); {First phase}
3:   for node in acpt do
4:     acohg2.findCycle(node); {Second phase}
5:     if acohg2.cycleFound() then
6:       return acohg2.acceptingPath();
7:     end if
8:   end for
9:   acohg1.insertTabu(acpt);
10: until empty(acpt)
11: return null;

```

The search that ACOhg-live performs is composed of two different phases (see Fig. 1 for an illustration of the search in the two phases). In the first one, ACOhg is utilized for finding accepting states in the Büchi automaton (line 2 in Algorithm 1). In this phase, the search of ACOhg starts in the initial node of the graph and the algorithm searches for accepting states. If they are found, 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. 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, ACOhg expands the state as a normal state and it is not 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 configuration of the ACOhg algorithms are, in general, different in the two phases since they tackle different objectives. We highlight this fact by using different variables for referring to both algorithms in Algorithm 1: `acohg1` and `acohg2`. For example, in the first phase a more exploratory search is required in order to find a diverse set of accepting

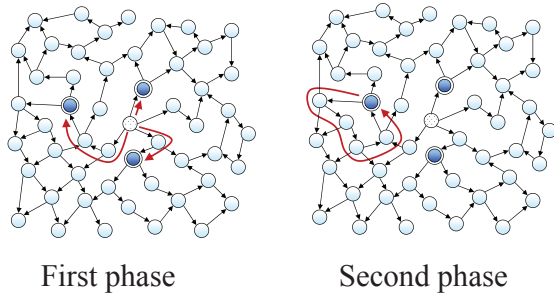


Fig. 1. An illustration of the search that ACOhg-live performs in the first and second phase

states. In addition, the accepting states are not known and no state-based heuristic can be used; a formula-based heuristic must be used instead. On the other hand, in the second phase the search must be guided to search for one concrete state and, in this case, a state-based heuristic like the Hamming distance or the finite state machines distance is more suitable.

In the rest of this section we are going to briefly describe the ACOhg algorithm that is used inside ACOhg-live.

A. ACOhg algorithm

ACOhg is a new kind of Ant Colony Optimization model proposed in [10] that can deal with construction graphs of unknown size or too large to fit into the computer memory. Actually, this new model was proposed for applying an ACO-like algorithm to the problem of searching for safety property violations in very large concurrent systems.

In short, the two main differences between ACOhg and the traditional ACO models are the following ones. First, the length of the paths (defined as the number of arcs in the path) traversed by ants in the construction phase is limited. That is, when the path of an ant reaches a given maximum length λ_{ant} the ant is stopped. Second, the ants start the path construction from different nodes during the search. At the beginning, the ants are placed on the initial node of the graph, and the algorithm is executed during a given number of steps σ_s (called *stage*). If no objective node is found, the last nodes of the best paths constructed by the ants are used as starting nodes for the ants in the next stage. In this way, during the next stage the ants try to go further in the graph (see [10] for more details). In Algorithm 2 we present the pseudocode of ACOhg.

In this work we use a node-based pheromone model, that is, the pheromone trails are associated to the nodes instead of the arcs. 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 0.1 and 10. In the *init* set (of 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. When ACOhg is executed inside ACOhg-live the initial node is q (the initial node of the graph) in the first phase and one accepting state in the second phase (line 4 in Algorithm 1).

Algorithm 2 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 colsiz do {Ant operations}
8:      $a^k = \emptyset$ ;
9:      $a_1^k = \text{selectInitNodeRandomly}(\text{init})$ ;
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:       $a^k = a^k + \text{node}$ ;
13:       $\tau = \text{localPheromoneUpdate}(\tau, \xi, \text{node})$ ;
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:   $\tau = \text{pheromoneEvaporation}(\tau, \rho)$ ;
21:   $\tau = \text{pheromoneUpdate}(\tau, a^{best})$ ;
22:  if step  $\equiv 0 \pmod{\sigma_s}$  then
23:    init = next_init;
24:    next_init =  $\emptyset$ ;
25:    stage = stage+1;
26:     $\tau = \text{pheromoneReset}()$ ;
27:  end if
28:  step = step + 1;
29: end while

```

After the initialization, the algorithm enters in a loop that is executed until a given maximum number of steps is performed (line 6). When ACOhg is used in the second phase of ACOhg-live it also stops when the objective node (the initial node) is found. In a loop, each ant builds a path starting in the final node of a previous path (line 9). This path is randomly selected from the *init* set using a fitness proportional probability distribution. 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 traditional stochastic rule used in ACO [24]. This rule takes into account the amount of pheromone of the following nodes and the heuristic values associated to these nodes. This heuristic value is defined after the heuristic function H used for guiding the search of the objective node. The exact expression we use is $\eta_j = 1/(1 + H(j))$. This way, η_j increases when $H(j)$ decreases (short distance to the objective node). After the movement of an ant from a node to the next one the pheromone trail associated to the new node is updated as in Ant Colony Systems (ACS) using the expression $\tau_j \leftarrow (1 - \xi)\tau_j$ (line 13). All this construction phase 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. When ACOhg is called from ACOhg-live the set of objective nodes O is the set of final states F in the first phase and the initial node of the search (accepting state) in the second one.

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. 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 (evaporation) according to the expression $\tau_j \leftarrow (1 - \rho)\tau_j$ (line 20). Then, the pheromone trails associated to the nodes traversed by the best-so-far ant (a^{best}) are increased (line 21) using the expression $\tau_j \leftarrow \tau_j + 1/f(a^{best})$, $\forall j \in a^{best}$. We use here the mechanism introduced in Max-Min Ant Systems (MMAS) 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).

The objective function f to be minimized is defined as follows

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

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. (1) makes the penalty higher in shorter cycles (see [11] for more details).

V. EXPERIMENTS

In this section we present some results obtained with our ACOhg-live algorithm. For the experiments we have selected three Promela models (two of them scalable) that are presented in the following section. After that, we discuss the algorithm parameters used in the experiments in Section V-B. In Section V-C we compare two state-based heuristics for the second phase of ACOhg-live. Next, in Section V-D we compare the results obtained with ACOhg-live against Nested-DFS, the traditional algorithm utilized for model checking in explicit state model checkers.

A. Promela Models

We have selected three Promela models implementing faulty concurrent systems previously reported in the literature [16]. All these models violate a liveness property that is

specified in LTL. In Table I we present the models with some information about them (lines of code, scalability, number of processes, and the LTL formulae they violate). They can be found with the source code of ACOhg and HSF-SPIN in <http://oplink.lcc.uma.es/software>.

TABLE I
PROMELA MODELS USED IN THE EXPERIMENTS

Model	LoC	Scalable	Processes	LTL formula
<code>alter</code>	64	no	2	$\square(p \rightarrow \diamond q) \wedge \square(r \rightarrow \diamond s)$
<code>giopij</code>	740	yes	$i + 3(j + 1)$	$\square(p \rightarrow \diamond q)$
<code>phij</code>	57	yes	$j + 1$	$\square(p \rightarrow \diamond q)$

The first model, `alter`, implements the alternating bit protocol [25]. The `giopij` model is an scalable implementation of the CORBA Inter-ORB protocol for i clients and j servers [26]. The last model, `phij`, is an implementation of the Dijkstra dining philosophers problem for j philosophers.

Out from these models, the smallest one is `alter`. In the experiments we use for the `giopij` model the values $j = 2$ (two servers) and $i = 2, 6, 10$. In the case of `phij` we set $j = 8, 14, 20$. The versions of the scalable models used are very large. As an illustration we can say that the smallest `giop` model, that is `giop22`, does not fit completely in the memory of the machine used for the experiments (512 MB). On the other hand, `phi20` requires more than 1039 GB of memory for storing all the states.

B. Parameters of the Algorithm

The parameters used in the experiments for the ACOhg algorithms in the two phases of ACOhg-live are shown in Table II. These parameters are not set in an arbitrary way: they are the result of a previous study aimed at finding the best configuration for them. One portion of this study was published in [27]. As mentioned in Section IV, in the first phase we use an explorative configuration ($\xi = 0.7$, $\lambda_{ant} = 20$) while in the second phase the configuration is adjusted to search in the region near the accepting state found (intensification).

TABLE II
PARAMETERS FOR THE ACOHG ALGORITHMS IN ACOHG-LIVE

First phase ACOhg		Second phase ACOhg	
Parameter	Value	Parameter	Value
<code>msteps</code>	100	<code>msteps</code>	100
<code>colsize</code>	10	<code>colsize</code>	20
λ_{ant}	20	λ_{ant}	4
σ_s	4	σ_s	4
ι	10	ι	10
ξ	0.7	ξ	0.5
a	5	a	5
ρ	0.2	ρ	0.2
α	1.0	α	1.0
β	2.0	β	2.0
p_p	1000	p_p	1000
p_c	1000	p_c	1000

With respect to the heuristic information, we use H_φ (the formula-based heuristic) in the first phase of the search when the objective is to find accepting states. In the second phase

we use the Hamming distance H_{ham} and the distance of finite state machines H_{fsm} . In fact, the first experiment performed consists in comparing the two heuristic functions for the second phase.

Since we are working with stochastic algorithms, we need to perform several independent runs in order to get quantitative information of the behaviour of the algorithm. For this reason we perform 100 independent runs to get a high statistical confidence, and we report the mean and the standard deviation of the independent runs. The machine used in the experiments is a Pentium IV at 2.8 GHz with 512 MB of RAM and Linux operative system with kernel version 2.4.19-4GB. In all the experiments the maximum memory assigned to the algorithms is 512 MB: when a process exceeds this memory it is automatically stopped. We do this in order to avoid a high amount of data flow from/to the secondary memory, which could affect significantly the CPU time required in the search.

C. Influence of the Heuristic Functions in the Second Phase

In this first experiment we compare the two heuristic functions utilized in the second phase of the search: the Hamming distance (H_{ham}) and the finite state machines distance (H_{fsm}). With this experiment we want to analyze which heuristic function is better for the models utilized in this work. The finite state machines distance requires more information about the concurrent system: the control flow of each process. This contrasts with the Hamming distance that can be computed from the binary representations of the current and goal states. Thus, we expect H_{fsm} to be a better guide for the algorithm and better results are expected when this heuristic is used.

In Table III we show the results obtained with both heuristic functions. The average values of the 100 independent runs are shown in normal size while the standard deviation values are shown as subscript. We highlight with a grey background the best results (maximum values for hit rate and minimum values for the rest of measures). 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. A plus sign means that the difference is significant and a minus sign means that it is not. In the case of the hit rate we use a Westlake-Schuirmann test of equivalence of two independent proportions, for the rest of the measures we use a Kruskal-Wallis test [28]. In the following we comment the results obtained for each measure.

With respect to the hit rate the results obtained using the two heuristics are similar except in the case of `giop10`. In this last case the use of H_{fsm} outperforms the hit rate obtained when H_{ham} is used with statistical confidence. In the other models the observed hit rate is similar and usually maximum. Thus, the first conclusion of the experiment is that ACOhg-live is able to obtain a high hit rate and the use of H_{fsm} seems to be beneficial for increasing the probability of finding an error trail.

Now we focus on the length of the error paths, which is a measure of their quality (the shorter the path the higher its

TABLE III
COMPARISON OF H_{ham} AND H_{fsm} IN ACOHG-LIVE

Models	Measure	H_{ham}		H_{fsm}		Test
alter	Hit rate	100/100		100/100		-
	Length	28.54	9.44	30.68	10.72	-
	Mem. (KB)	1925.00	0.00	1925.00	0.00	-
	Time (ms)	88.90	15.03	90.00	13.86	-
giop2	Hit rate	100/100		100/100		-
	Length	43.09	5.33	43.76	5.82	-
	Mem. (KB)	2834.48	369.42	2953.76	327.48	+
	Time (ms)	817.50	560.73	747.50	408.09	-
giop6	Hit rate	100/100		100/100		-
	Length	58.41	7.16	58.77	7.21	-
	Mem. (KB)	5418.32	1161.17	5588.04	631.36	-
	Time (ms)	16049.10	15256.93	8733.50	3304.90	-
giop10	Hit rate	60/100		86/100		+
	Length	61.10	6.42	62.85	7.03	-
	Mem. (KB)	9669.80	1597.14	9316.67	700.44	+
	Time (ms)	87236.00	69218.19	43059.07	21417.74	+
phi8	Hit rate	100/100		100/100		-
	Length	52.38	9.26	51.36	6.95	-
	Mem. (KB)	2097.52	21.74	2014.32	18.87	+
	Time (ms)	2271.40	573.56	2126.10	479.64	-
phi14	Hit rate	99/100		99/100		-
	Length	74.68	8.66	76.05	9.35	-
	Mem. (KB)	2593.37	179.03	2496.07	41.81	+
	Time (ms)	9369.90	3706.83	8070.30	1530.12	+
phi20	Hit rate	99/100		98/100		-
	Length	95.28	9.97	97.39	10.14	-
	Mem. (KB)	3324.26	104.27	3244.67	91.33	+
	Time (ms)	21323.54	10600.89	18064.90	5538.30	+

quality). We observe in Table III that the length of the error paths obtained when H_{ham} is utilized is shorter, in general, than the length of the ones obtained with H_{fsm} (with one exception in `phi8`). However, the difference is small and not statistically significant; therefore we cannot assure that one heuristic is better than the other in this aspect.

Finally, concerning the computational resources (memory and time required for finding an error path) we can observe that H_{fsm} outperforms, in general, the results of H_{ham} . The algorithm ACOhg-live using H_{fsm} requires less memory and time to find a counterexample of the liveness property. All the statistically significant differences in memory and time support this observation except that of the memory in `giop2`.

In summary, we can conclude that using H_{fsm} in the second phase of ACOhg-live instead of H_{ham} increases the hit rate and decreases the computational resources required for the search while at the same time the length of the error paths is maintained. This observation is in concordance with the intuition we mentioned previously in this section: better results are expected when H_{fsm} is used because this heuristic reveals more information of the model than H_{ham} .

D. ACOhg-live vs. Nested-DFS

In the next experiment we compare the results obtained with ACOhg-live using H_{fsm} (the best heuristic according to the previous experiment) 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 IV we show the results of both algorithms (for ACOhg-live we only show the average). For comparing the hit rate we use again the Westlake-Schuirmann test. However, for the other measures we utilize this time the one sample Wilcoxon sign rank test

because we compare one sample (the results of ACOhg-live) with one single value (the result of Nested-DFS).

TABLE IV
COMPARISON BETWEEN ACOHG-LIVE AND NESTED-DFS

Models	Measure	ACOhg-live	Nested-DFS	Test
alter	Hit rate	100/100	1/1	-
	Length	30.68	64.00	+
	Mem. (KB)	1925.00	1873.00	+
	Time (ms)	90.00	0.00	+
giop2	Hit rate	100/100	1/1	-
	Length	43.76	298.00	+
	Mem. (KB)	2953.76	7865.00	+
	Time (ms)	747.50	240.00	+
giop6	Hit rate	100/100	0/1	+
	Length	58.77	•	•
	Mem. (KB)	5588.04	•	•
	Time (ms)	8733.50	•	•
giop10	Hit rate	86/100	0/1	+
	Length	62.85	•	•
	Mem. (KB)	9316.67	•	•
	Time (ms)	43059.07	•	•
phi8	Hit rate	100/100	1/1	-
	Length	51.36	3405.00	+
	Mem. (KB)	2014.32	4005.00	+
	Time (ms)	2126.10	40.00	+
phi14	Hit rate	99/100	1/1	-
	Length	76.05	10001.00	+
	Mem. (KB)	2496.07	59392.00	+
	Time (ms)	8070.30	2300.00	+
phi20	Hit rate	98/100	1/1	-
	Length	97.39	10001.00	+
	Mem. (KB)	3244.67	392192.00	+
	Time (ms)	18064.90	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` and `giop10` because it requires more than the memory available in the machine used for the experiments (512 MB). This result is very important since Nested-DFS is a standard *de facto* in the formal methods community for checking liveness properties. Our first conclusion is that ACOhg-live is able to find error paths running in regular machines while Nested-DFS is not. If we focus on the remaining models we observe a similar hit rate in both algorithms (in Nested-DFS the hit rate is always 100% since it is a deterministic algorithm).

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). The difference increases with the size of the models. For example, for `alter` the average length obtained with ACOhg-live (30.68) is half the length obtained with Nested-DFS (64.00) and for `phi8` (that is a larger model) the length obtained with ACOhg-live (51.36) is approximately one sixtieth of the length obtained with Nested-DFS (3405.00). The biggest differences can be observed in `phi14` and `phi20` in which ACOhg-live finds error paths that are more than one hundred times shorter than the ones found by Nested-DFS. Furthermore, we limited the exploration depth of Nested-DFS to 10000 in order to avoid stack overflow problems. This means that the lengths of the error paths that are shown in

Table IV for Nested-DFS in `phi14` and `phi20` are in fact a lower bound of the real length that Nested-DFS would obtain in theory. In conclusion, ACOhg-live obtains error paths that are by far shorter than the ones obtained with Nested-DFS. This is a very important result since short error paths are preferred in order for the programmers to understand faster what is wrong in the concurrent model.

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 that of `giop6` and `giop10` in which Nested-DFS requires more than 512 MB of memory while ACOhg-live obtains error paths with 38 MB at most. Memory is the main problem of the traditional model checking techniques and we can observe here that ACOhg-live is more tolerant to the state explosion problem. 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 the error path obtained is much shorter, as it happens.

In summary, the results obtained with ACOhg-live outperform the ones of Nested-DFS, the traditional algorithm utilized in model checking for finding liveness errors. ACOhg-live is able to find much shorter error paths using much less memory. This improvement implies that ACOhg-live can find errors for large models in machines with a regular amount of memory (512 MB in our case) and, in addition, these error paths are more suitable for the programmers to understand where is the problem of the concurrent system.

VI. DISCUSSION

In this section we discuss the utility of our proposal from the software engineer point of view, giving some guidelines that could help practitioners to decide when to use ACOhg-live for searching for errors in concurrent systems.

First of all, it must be clear from the beginning that ACOhg-live can find short counterexamples in faulty concurrent systems, but it cannot be used for verifying that a concurrent system satisfies a given liveness property. Thus, ACOhg-live should be used in the first/middle stages of the software development and after any maintenance modification made on the concurrent system. In these phases errors are expected to exist in the concurrent software. In spite of the previous considerations, ACOhg-live can also be used to assure with high probability that the software satisfies a given desirable property (perhaps obtained from the specification). In this case it can be used at the end of the software life cycle. This is similar to state that the software is “probably correct” after a testing phase in which the software has been run on a set of test cases. Unlike this, in critical systems (like airplane controllers) an exhaustive algorithm must be used in the final stages to verify that the software really satisfies the liveness property.

We have seen in the experimental section what are the main advantages of using ACOhg-live against exhaustive techniques (such as Nested-DFS) in the search for liveness property violations: shorter error paths can be obtained with higher probability and less memory. But, what about the drawbacks? The main drawback we have found from the point of view of the applicability of ACOhg-live is the large amount of parameters of the algorithm. These parameters make ACOhg-live more flexible, since it is possible to tackle models with different features changing the parameterization. However, software practitioners have no time to adjust the parameters of the algorithm and they want a robust algorithm that works well in most situations with minimum cost. In this sense, we understand that a parameterization study must be a priority in the following steps of this research. In fact, from the experiments performed for this and previous work we have outlined a set of rules for assigning values to the parameters (some of them are published in [27]). We already know how the number of parameters of ACOhg-live can be largely reduced (work in progress).

VII. CONCLUSIONS AND FUTURE WORK

We have presented here a novel proposal based on ant colony optimization for finding liveness property violations in concurrent systems. This problem is of capital importance in the development of software for critical systems. In addition to the description of the proposal we have shown its validity with a series of experiments. First, we have compared the use of two different heuristic functions for guiding the search. After that, we have compared the results obtained by our proposal with the traditional algorithm utilized for finding liveness errors in concurrent systems: Nested-DFS. The results show that ACOhg-live is able to outperform Nested-DFS in efficacy and efficiency. It requires a very low amount of memory and it is able to find errors even in models in which Nested-DFS fails in practice due to memory requirements.

As future work we plan to use the idea of classifying the strongly connected components of the Büchi automaton in order to change the way in which the search is performed, as done in [18]. This way a more efficient search can be performed and larger models can be tackled. ACOhg-live can be used with other techniques for reducing the amount of memory required in the search such as partial order reduction, symmetry reduction, or state compression. As a future work we plan to combine these techniques with ACOhg-live.

REFERENCES

- [1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.
- [2] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, January 2000.
- [3] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs, Workshop*. London, UK: Springer-Verlag, 1982, pp. 52–71.
- [4] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.

- [5] G. J. Holzmann, *The SPIN Model Checker*. Addison-Wesley, 2004.
- [6] A. Lluch-Lafuente, S. Leue, and S. Edelkamp, "Partial Order Reduction in Directed Model Checking," in *9th International SPIN Workshop on Model Checking Software*. Grenoble: Springer, April 2002.
- [7] A. L. Lafuente, "Symmetry Reduction and Heuristic Search for Error Detection in Model Checking," in *Workshop on Model Checking and Artificial Intelligence*, August 2003.
- [8] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 4, April 1994.
- [9] E. Alba and F. Chicano, "Ant colony optimization for model checking," in *EUROCAST 2007 (LNCS)*, ser. Lecture Notes in Computer Science, vol. 4739, Gran Canaria, Spain, February 2007, pp. 523–530.
- [10] —, "Finding safety errors with ACO," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*. London, UK: ACM Press, July 2007, pp. 1066–1073.
- [11] F. Chicano and E. Alba, "Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models," *Information Processing Letters*, 2007, (to appear).
- [12] P. Godefroid and S. Khurshid, "Exploring very large state spaces using genetic algorithms," *International Journal on Software Tools for Technology Transfer*, vol. 6, no. 2, pp. 117–127, 2004 2004.
- [13] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems*. New York, NY, USA: Springer-Verlag New York, Inc., 1992.
- [14] B. Alpern and F. B. Schneider, "Defining liveness," *Inform. Proc. Letters*, vol. 21, pp. 181–185, 1985.
- [15] S. Edelkamp, A. L. Lafuente, and S. Leue, "Protocol Verification with Heuristic Search," in *AAAI-Spring Symposium on Model-based Validation Intelligence*, 2001, pp. 75–83.
- [16] S. Edelkamp, S. Leue, and A. Lluch-Lafuente, "Directed explicit-state model checking in the validation of communication protocols," *International Journal of Software Tools for Technology Transfer*, vol. 5, pp. 247–267, 2004.
- [17] G. J. Holzmann, D. Peled, and M. Yannakakis, "On nested depth first search," in *Proc. Second SPIN Workshop*. American Mathematical Society, 1996, pp. 23–32.
- [18] S. Edelkamp, A. L. Lafuente, and S. Leue, "Directed Explicit Model Checking with HSF-SPIN," in *Lecture Notes in Computer Science, 2057*. Springer, 2001, pp. 57–79.
- [19] A. Groce and W. Visser, "Model checking java programs using structural heuristics," in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM Press, 2002, pp. 12–21.
- [20] —, "Heuristics for model checking java programs," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6, no. 4, pp. 260–276, 2004.
- [21] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1085–1110, December 2001.
- [22] P. Ammann, P. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods*. Brisbane, Australia: IEEE Computer Society Press, December 1998, pp. 46–54.
- [23] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys*, vol. 35, no. 3, pp. 268–308, 2003.
- [24] M. Dorigo and T. Stützle, *Ant Colony Optimization*. The MIT Press, 2004.
- [25] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson, "A note on reliable full-duplex transmission over half-duplex links," *Commun. ACM*, vol. 12, no. 5, pp. 260–261, 1969.
- [26] M. Kamel and S. Leue, "Validation of a remote object invocation and object migration in CORBA GIOP using Promela/Spin," in *International SPIN Workshop*, 1998.
- [27] E. Alba and F. Chicano, "ACOhg: Dealing with huge graphs," in *Proceedings of the Genetic and Evolutionary Conference*. London, UK: ACM Press, July 2007, pp. 10–17.
- [28] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 2007.