

Searching for Liveness Property Violations in Concurrent Systems with ACO

Francisco Chicano
University of Málaga, Spain
chicano@lcc.uma.es

Enrique Alba
University of Málaga, Spain
eat@lcc.uma.es

ABSTRACT

Liveness properties in concurrent systems are, informally, those properties that stipulate that something good eventually happens during execution. In order to prove that a given system satisfies a liveness property, model checking techniques are utilized. However, most of the model checkers found in the literature use exhaustive deterministic algorithms that require huge amounts of memory if the concurrent system is large. Here we propose the use of an algorithm based on ACOhg, a new kind of Ant Colony Optimization algorithm, for searching for liveness property violations in concurrent systems. We also take into account the structure of the liveness property in order to improve the efficacy and efficiency of the search. 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 and Improved-Nested-DFS, two algorithms used in the literature for this task in the model checking community. 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.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*model checking* ; I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search—*Heuristic methods*; G.1.6 [Numerical Analysis]: Optimization—*Global optimization*

General Terms

Verification, Algorithms, Experimentation

Keywords

Liveness properties, HSF-SPIN, SPIN, ant colony optimization, metaheuristics

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.

Copyright 2008 ACM 978-1-60558-130-9/08/07...\$5.00.

1. INTRODUCTION

Model checking [5] 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. Unlike test case generation, model checking is a formal method that can prove that the software system fulfils a specified property. 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.

Several techniques exist to alleviate the state explosion problem. They reduce the amount of memory required for the search by following different approaches (see [13]). However, exhaustive search algorithms are always handicapped in large concurrent systems because most of these systems are too complex even for the most advanced techniques. 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.

Non-exhaustive algorithms can find errors in programs using less computational resources than exhaustive algorithms, 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 [16]. 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.

A well-known class of non-exhaustive algorithms for solving complex problems is the class of metaheuristic algorithms [4]. They are search algorithms used in optimization problems that can find good quality solutions in a reasonable time. In fact, Genetic Algorithms [11] and Ant Colony Optimization [2] have been applied in the past for searching for safety property violations in concurrent systems. The search for liveness property violations is not so direct and, in fact, no metaheuristic algorithm has been ever applied to it in the literature to the best of our knowledge.

In this work we propose an algorithm based on the ACOhg algorithm presented in [2] for searching for liveness property violations in concurrent systems. We also take into account the internal structure of the property itself in order to im-

prove the search. The paper is organized as follows. The next section presents the foundations of the problem. In Section 3 the problem is formalized as a graph search. Section 4 describes our algorithmic proposal, called ACOhg-live. In Section 5 we present some experimental results analyzing the influence on the performance measures of the knowledge about the property structure and we compare our proposal against two exhaustive algorithms utilized for checking liveness properties in explicit state model checking: Nested-DFS and Improved-Nested-DFS. Finally, Section 6 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 [8], an experimental model checker by Edelkamp, Lluch-Lafuente and Leue based on the popular model checker SPIN [13]. First, we formally define the concept of *property* of a concurrent system. After that, we detail how the properties can be represented using automata and we define the concept of *strongly connected components*, which allows us to implement improvements on the search algorithms. Finally, we detail how heuristic information is used to guide the search for property violations.

2.1 Properties

Let S be the set of *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 real executions of the program form a subset of S^ω . A *property* P is 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 $\sigma \not\vdash P$. 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,

$$\forall \sigma \in S^\omega : \sigma \not\vdash P \Rightarrow (\exists i \geq 0 : \forall \beta \in S^\omega : \sigma_i \beta \not\vdash P) , \quad (1)$$

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,

$$\forall \alpha \in S^* : \exists \beta \in S^\omega, \alpha \beta \vdash P . \quad (2)$$

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 [3].

The properties of a concurrent system are usually specified using a temporal logic, like Linear Temporal Logic (LTL) or computation Tree logic (CTL). We use the former in this work. In this case, atomic propositions are defined on the base of the variables and program counters of the processes

of the system. The property is expressed using an LTL formula composed of the atomic propositions. For example, an LTL property can be $\Box p$ (read “henceforth p ”), where $p \equiv x > 3$. This property specifies an invariant: for a concurrent system to fulfil this property the variable x must always be greater than 3. This property consists of all the executions σ in which the variable x is greater than 3 for all the states of σ .

2.2 Property Automaton and Checking

The way in which LTL properties are checked in HSF-SPIN is by means of an automaton that captures the violations of the property, that is, the automaton accepts the executions in which the property is violated. This automaton is called *never claim* in SPIN and HSF-SPIN. This never claim can be automatically computed from the negation of the LTL formula [10]. In order to find a violation of a given LTL property, HSF-SPIN (and SPIN) explores the synchronous product of the concurrent model and the never claim, also called Büchi automaton. As an illustration, in Fig. 1 we show the automaton of a simple concurrent system (left box), the never claim used to check the LTL formula $\Box(p \rightarrow \Diamond q)$ (which means that an occurrence of p is always followed by an occurrence of q , not necessarily in the next state), and the synchronous product of these two automata.

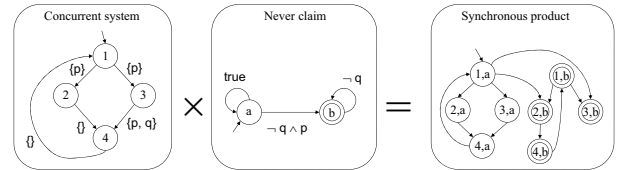


Figure 1: Synchronous product of a simple concurrent system and a never claim.

HSF-SPIN (and 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 LTL formula and, thus, the whole LTL formula. During the search it can also be found a state in which the end state of the never claim is reached. This means that an execution has been found that violates the safety component of the LTL formula and the partial execution $\alpha \in S^*$ that leads the model to that state violates the LTL formula¹. In HSF-SPIN and SPIN model checkers the search can be done using the Nested Depth First Search algorithm (NDFS) [14]. However, if the LTL formula represents a safety property (the liveness component is *true*) the problem of finding a property violation is reduced to find a partial execution $\alpha \in S^*$ violating the LTL formula, 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. These classical algorithms cannot be used when we are searching for liveness property violations (as we do in this work) because they are not designed to find the cycle of states β above mentioned.

¹A deeper explanation of the foundations of the automata-based model checking can be found in [5] and [13].

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. For improving the search first we need to compute the *strongly connected components* (SCCs) of the never claim. A strongly connected subgraph $G = (V, A)$ of a directed graph is that in which for all pairs of different nodes $u, v \in V$ there exist two paths: one from u to v and another one from v to u . The strongly connected components of a directed graph are its maximal strongly connected subgraphs. Once we have the SCCs of the never claim we have to classify them into three categories depending on the accepting cycles they include. We denote with N-SCC those SCCs in which no cycle is accepting. A P-SCC is that in which there exists at least one accepting cycle and at least one non-accepting cycle. Finally, F-SCC are the SCCs in which all the cycles are accepting [9]. All the cycles found in the Büchi automaton have an associated cycle 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. In HSF-SPIN, there is an implementation of an improved version of NDFS called Improved-Nested-DFS (INDFS) that takes into account the classification of the SCCs of the never claim [9].

2.3 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 to better guide exhaustive algorithms. In [12] structural heuristics are introduced that attempt to explore the structure of a program in a way conducive to find errors. One example of this kind of heuristic information is code coverage, a well known metric in the software testing domain. Another example is thread interleaving, in which states yielding a thread scheduling with many context changes are rewarded.

Unlike structural heuristics, property-specific heuristics [12] rely on features of the particular property checked. Formula-based heuristics, for example, are based on the expression of the LTL formula checked [8]. Using the logic expression that must be false in an objective state, these heuristics estimate the number of transitions required to get such an objective 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 [8].

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. In the experimental section we will use H_{fsm} .

3. PROBLEM FORMALIZATION

In this article we tackle the problem of searching for liveness property violations in concurrent systems. As we previously mentioned, this problem can be translated into the search of a path in a graph (the Büchi automaton) starting in the initial state and ending in an objective node (accepting state) and an additional cycle involving the objective node. 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 *final 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 hand consists in finding a starting path π ending in an final node plus a cycle ν containing the final 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 Büchi automaton B (synchronous product of the concurrent system and the never claim). 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 final 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.

4. ALGORITHMIC PROPOSAL

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 [2]. We describe ACOhg in the next section and ACOhg-live in Section 4.2. Finally, we describe how the improvement based on SCCs is applied to ACOhg-live.

4.1 ACOhg algorithm

ACOhg is a new kind of Ant Colony Optimization model proposed in [2] 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 [2] for more details). In Algorithm 1 we present the pseudocode of ACOhg.

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 colsizes 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

```

In the following we will describe the algorithm, but previously we are going to clarify some issues related to the notation used in Algorithm 1. In the pseudocode, the path traversed by the k th artificial ant is denoted with a^k . For this reason we use the same notation as in Section 3 for referring to the length of the path ($|a^k|$), the j th node of the path (a_j^k), and the last node of the path (a_*^k). We use the operator $+$ to refer to the concatenation of two paths. In line 10, we use the expression $T(a_*^k) - a^k$ to refer to the elements of $T(a_*^k)$ that are not in the sequence a^k . That is, in that expression we interpret a^k as a set of nodes.

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 (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 (line 9). This path is randomly selected from the **init** set using a fitness proportional probability distribution. For the path construction, 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 [7]. 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 $\eta_j = 1/(1 + H(j))$. 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). 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. 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.

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 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 using the expression $\tau_j \leftarrow \tau_j + 1/f(a^{best})$, $\forall j \in a^{best}$ (line 21). 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

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

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.

4.2 ACOhg-live

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 (see Fig. 2). 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 is empty. That is, although the algorithm searches for accepting states, there is no preference on a specific set

Algorithm 2 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;
```

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).

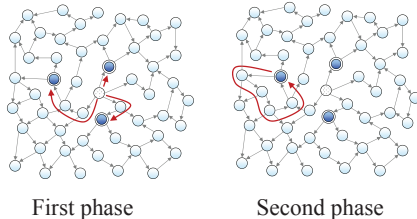


Figure 2: An illustration of the search that ACOhg-live performs in the first and second phase.

The configuration of the ACOhg algorithms is, 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 2: `acohg1` and `acohg2`. For example, in the first phase (`acohg1`) a more exploratory search is required in order to find a diverse set of accepting 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 (`acohg2`) 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.

4.3 Improvement using SCCs

We can make the search for liveness errors more efficient if we take into account the classification of the never claim SCCs. The improvements are localized in two places of

ACOhg-live. During the first phase, in which accepting states are searched for, those accepting states that belong to an N-SCC in the never claim are ignored. The reason is that an accepting state in an N-SCC cannot be part of an accepting cycle. This way, we reduce the number of accepting states to be explored in the second phase.

The second improvement is localized in the computation of the successors of a state (line 10 in Algorithm 1) in both, the first and the second phase of ACOhg-live. When the successors are computed, ACOhg checks if they are included in the path that the ant has traversed up to the moment. If they are, the state is not considered as the next node to visit since the ant would build a cycle. The improvement consists in checking if this cycle is in an F-SCC. This can be easily checked by finding if the state that closes the cycle is in an F-SCC of the never claim. If it is, then an accepting cycle has been found and the global search stops.

The advantages of these improvements depend on the structure of the LTL formula and the model to check. We can notice no advantages in some cases, especially when the number of N-SCC and F-SCC is small. However, the computational cost of the improvements is negligible, since it is possible to check the kind of SCC associated to a state in constant time.

5. EXPERIMENTS

In this section we present some results obtained with our ACOhg-live algorithm. For the experiments we have selected five Promela² models (three of them scalable) that are presented in the following section. After that, we discuss the algorithmic parameters used in the experiments in Section 5.2. In Section 5.3 we compare two versions of the algorithm with and without the improvement based on the SCC classification of the never claim. Next, in Section 5.4 we compare the results obtained with ACOhg-live against NDFS and INDFS.

5.1 Promela Models

We have selected five Promela models implementing faulty concurrent systems. Most of them have been previously reported in the literature [9]. All of these models violate a liveness property that is specified in LTL. In Table 1 we present the models with some information about them (lines of code, scalability, processes, and kind of LTL formula).

Table 1: Promela models used in the experiments

Model	LoC	Sc.	Processes	LTL formula
alter	64	no	2	$\Box(p \rightarrow \Diamond q) \wedge \Box(r \rightarrow \Diamond s)$
giopj	740	yes	$j + 6$	$\Box(p \rightarrow \Diamond q)$
phi _j	57	yes	$j + 1$	$\Box(p \rightarrow \Diamond q)$
elev _j	191	yes	$j + 3$	$\Box(p \rightarrow \Diamond q)$
sgc	1001	no	20	$\Diamond p$

The first model, `alter`, implements the alternating bit protocol. The `giopj` model is a scalable implementation of the CORBA Inter-ORB protocol for j clients and one server [15]. The `phij` model is an implementation of the Dijkstra dining philosophers problem for j philosophers. The fourth model, `elevj` simulates an elevator working on a building with j floors. Finally, `sgc` implements an operator procedure of a power plant [18].

²Promela is the modeling language used by SPIN and HSF-SPIN for specifying the concurrent system.

Out of these models, the smallest one is **alter** with 393 states. In the experiments we use for the **giopj** model the values $j = 10, 15, 20$. In the case of **phij** we set $j = 20, 30, 40$. For the third scalable model, **elevj**, we use $j = 10, 15, 20$. The versions of the scalable models used are very large: the only model that fits into the memory of the machine used for the experiments is **alter**. As an illustration we can say that **phi20** requires more than 1039 GB of memory for storing all the states.

5.2 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 2. These parameters have been selected according to the recommendations in [1]. As mentioned in Section 4, 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). It is possible that these parameters can be tuned in order to improve the efficiency and the efficacy of the search. However, this tuning requires time and we must take into account this time when the algorithm has to be applied, especially if we want to compare it against parameter-free algorithms like NDFS or INDFS. In this work, we do not tune the parameters of the algorithms; we just run the algorithm with the parameters recommended in the literature for ACOhg. This way the tuning time is zero.

Table 2: Parameters for ACOhg-live

First phase ACOhg		Second phase ACOhg	
Parameter	Value	Parameter	Value
$msteps$	100	$msteps$	100
$colsize$	10	$colsize$	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 distance of finite state machines H_{fsm} . ACOhg-live has been implemented inside HSF-SPIN.

We need to perform several independent runs in order to get quantitative information of the behaviour of the algorithm. 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 with kernel version 2.4.19-4GB.

5.3 Influence of the SCC Improvement

In this first experiment 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 3 we show the hit rate, the length

of the error trails, the memory required (in Kilobytes), and the run time (in milliseconds) of the ACOhg-live algorithms. 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 the 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 (last column). 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-Schuirman test of equivalence of two independent proportions, for the rest of the measures we use a Kruskal-Wallis test [17].

Table 3: Influence of the SCC improvement

Models	Meas.	ACOhg-live ⁻	ACOhg-live ⁺	T
alter	Hit	100/100	100/100	-
	Len.	10.00	15.82	+
	Mem.	1929.00	1929.00	-
	CPU	241.80	10.40	+
giop10	Hit	84/100	89/100	-
	Len.	68.57	67.60	-
	Mem.	6375.90	5098.75	+
	CPU	7816.55	935.84	+
giop15	Hit	46/100	57/100	-
	Len.	81.26	78.30	+
	Mem.	9001.17	8538.54	-
	CPU	11725.65	2016.84	+
giop20	Hit	14/100	30/100	-
	Len.	93.29	88.47	+
	Mem.	11132.71	10403.17	-
	CPU	11360.00	2575.33	+
phi20	Hit	98/100	97/100	-
	Len.	88.29	108.73	+
	Mem.	3398.63	3385.04	-
	CPU	5162.04	851.75	+
phi30	Hit	94/100	95/100	-
	Len.	122.60	139.15	+
	Mem.	5146.62	5148.12	-
	CPU	10980.64	2701.79	+
phi40	Hit	77/100	81/100	-
	Len.	154.74	166.83	+
	Mem.	7573.68	7545.35	+
	CPU	20422.60	5807.41	+
elev10	Hit	100/100	100/100	-
	Len.	126.56	127.76	-
	Mem.	2617.60	2617.04	-
	CPU	2577.30	2372.90	-
elev15	Hit	100/100	100/100	-
	Len.	182.02	180.04	-
	Mem.	3163.56	3164.64	-
	CPU	2683.00	2812.10	-
elev20	Hit	100/100	100/100	-
	Len.	233.00	231.62	-
	Mem.	3716.44	3716.92	-
	CPU	3900.60	3034.00	-
sgc	Hit	32/100	100/100	+
	Len.	24.00	24.00	-
	Mem.	2699.00	2285.00	+
	CPU	575191.88	710.20	+

With respect to the hit rate both algorithms obtain the maximum value (100%) in the models **alter** and **elevj**. In **giopj** ACOhg-live⁺ obtains higher hit rate than ACOhg-live⁻ and the difference increases with the model size. In **phij** the hit rate is similar in both algorithms (not statistically significant). In **sgc**, ACOhg-live⁺ obtains the maximum hit rate while ACOhg-live⁻ is only able to find an error in 32% of the executions. Thus, the first conclusion of the experiment is that ACOhg-live⁺ is able to obtain higher hit rates than ACOhg-live⁻.

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

its quality). We observe in Table 3 that the length of the error paths obtained with ACOhg-live⁻ is shorter with statistical confidence than the length of the ones obtained with ACOhg-live⁺ in `alter` and `phij`. On the other hand, the lengths of the error paths obtained by ACOhg-live⁺ are shorter than the ones obtained by ACOhg-live⁻ with statistical confidence in `giop15` and `giop20`. In the rest of the models there is no statistically significant difference. In general, the SCC improvement can entail an increase in the length of the error paths found. This happens when a cycle is found inside an F-SCC. In that case the algorithm immediately stops yielding an error path that can be longer than the one obtained if the search would continue.

Finally, concerning the computational resources (memory and time required for finding an error path) we can observe that ACOhg-live⁺ outperforms the results of ACOhg-live⁻. All the statistically significant differences in memory and CPU time support this claim. This is the expected result, since the cost of the SCC improvement in terms of memory and computation time is negligible and, in addition, the search is more effective.

In summary, we can 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.

5.4 ACOhg-live vs. Exhaustive Algorithms

In the next experiment we compare the results obtained with ACOhg-live⁺ against a classical algorithm used for finding liveness errors in concurrent systems, NDFS, and an improved version of NDFS that takes into account the SCCs of the never claim, INDFS. Both algorithms are deterministic and for this reason we only perform one single run. In Table 4 we show the results of the three 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 results of NDFS and INDFS). The results of the Wilcoxon test are the same in all the measures and models: all the differences are statistically significant. On the other hand, the Westlake-Schuirmann test reveals that the only statistically significant differences in the hit rate are those of the `giopj` models, `phi30`, and `phi40`. Due to this homogeneity in the statistical analysis we do not show the tests results in Table 4 in order to save room.

Concerning the hit rate we can observe that ACOhg-live⁺ is the only one that is able to find error paths in all the models. NDFS and INDFS are not able to find error paths in the `giopj` models and the largest `phij` models because they require more than the available memory. This is a very important result since NDFS is a very popular algorithm in the formal methods community for checking liveness properties using an explicit state model checker. We must clarify here that NDFS and INDFS do not store the visited states in memory (as other algorithms like BFS do), they only store the states belonging to a branch of the search as they are needed. If we focus on the remaining models we observe a similar hit rate in both algorithms (in NDFS and INDFS the hit rate is always 100% since they are deterministic algorithms).

Table 4: Comparison of ACOhg-live⁺ and (I)NDFS

Models	Meas.	ACOhg-live ⁺	NDFS	INDFS
alter	Hit	100/100	1/1	1/1
	Len.	15.82	64.00	64.00
	Mem.	1929.00	1887.00	1887.00
	CPU	10.40	0.00	0.00
giop10	Hit	89/100	0/1	0/1
	Len.	67.60	•	•
	Mem.	5098.75	•	•
	CPU	935.84	•	•
giop15	Hit	57/100	0/1	0/1
	Len.	78.30	•	•
	Mem.	8538.54	•	•
	CPU	2016.84	•	•
giop20	Hit	30/100	0/1	0/1
	Len.	88.47	•	•
	Mem.	10403.17	•	•
	CPU	2575.33	•	•
phi20	Hit	97/100	1/1	1/1
	Len.	108.73	10001.00	10001.00
	Mem.	3385.04	392192.00	388096.00
	CPU	851.75	15670.00	15320.00
phi30	Hit	95/100	0/1	0/1
	Len.	139.15	•	•
	Mem.	5148.12	•	•
	CPU	2701.79	•	•
phi40	Hit	81/100	0/1	0/1
	Len.	166.83	•	•
	Mem.	7545.35	•	•
	CPU	5807.41	•	•
elev10	Hit	100/100	1/1	1/1
	Len.	127.76	629.00	619.00
	Mem.	2617.04	2195.00	2015.00
	CPU	2372.90	0.00	10.00
elev15	Hit	100/100	1/1	1/1
	Len.	180.04	849.00	839.00
	Mem.	3164.64	2271.00	2271.00
	CPU	2812.10	10.00	10.00
elev20	Hit	100/100	1/1	1/1
	Len.	231.62	1069.00	1059.00
	Mem.	3716.92	2595.00	2399.00
	CPU	3034.00	20.00	10.00
sgc	Hit	100/100	1/1	1/1
	Len.	24.00	9999.00	46.00
	Mem.	2285.00	15360.00	2014.00
	CPU	710.20	500.00	10.00

With respect to the length of the error paths we observe that ACOhg-live⁺ obtains shorter error paths than NDFS and INDFS in all the models (with statistical significance). Furthermore, we limited the exploration depth of NDFS and INDFS to 10,001 in order to avoid stack overflow problems. If we allowed these algorithms to explore deeper regions we would obtain longer error paths with them. In fact, we run NDFS and INDFS using a depth limit of 50,001 in `phi20` and we got an error path of 50,001 states. This means that the lengths of the error paths that are shown in Table 4 for NDFS and INDFS in `phi20` and for NDFS in `sgc` are in fact a lower bound of the real length that these algorithms would obtain in theory. In conclusion, ACOhg-live⁺ obtains error paths that are by far shorter than the ones obtained with NDFS and INDFS. 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 both NDFS and INDFS in `giopj` and `phij`. In the other models the difference between the memory required by ACOhg-live⁺ and INDFS is less than 1.5 MB. The biggest differences are that of `giopj`, `phi30`, and `phi40`, in which NDFS and INDFS require more than the available memory while ACOhg-live⁺

obtains error paths with 10 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, NDFS and INDFS are faster than ACOhg-live⁺ in all the models except in phi20. 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 three seconds (in elev20), which is not large if we take into account that the error path obtained is much shorter.

In summary, the results obtained with ACOhg-live⁺ outperform the ones of the exhaustive algorithms traditionally used in model checking. 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.

Although the concurrent models used in this work are really large, real concurrent systems can be even larger. That is, we are not dealing with excessively large unreal models. In fact, in order to find errors in real concurrent systems using existing approaches it is necessary to apply techniques that require human involvement, such as abstract interpretation [6]. In this context, ACOhg-live⁺ is a completely automatic algorithm that is able to outperform existing approaches. Thus, we think that it is suitable for finding liveness property violations in real concurrent systems.

6. CONCLUSIONS AND FUTURE WORK

We have presented here a novel proposal based on ant colony optimization for finding liveness property violations in concurrent systems: ACOhg-live. This problem is of capital importance in the development of software for critical systems and no metaheuristic algorithm has been applied to it in the literature, to the best of our knowledge. We have also proposed an improvement of the technique based on the classification of the strongly connected components of the never claim. We have shown the performance of the proposals with a series of experiments. First, we have compared two versions of the algorithm with and without the SCC improvement. After that, we have compared the results obtained with our proposal against two algorithms used for finding liveness errors in concurrent systems: NDFS and INDFS. The results show that ACOhg-live is able to outperform both algorithms in efficacy and efficiency. It requires a very low amount of memory and it is able to find errors even in models in which NDFS and INDFS fail in practice due to memory requirements.

As future work we plan to combine ACOhg-live with other techniques for reducing the amount of memory required in the search such as partial order reduction, symmetry reduction, or state compression. We have observed in a preliminary (not published) study that a version of ACOhg-live that does not use pheromone trails for guiding the search is also able to obtain competitive results and requires even less memory. An additional advantage of such kind of algorithm is that it has fewer parameters than a version using pheromone trails. We will study the advantages and limitations of this alternative.

7. ACKNOWLEDGEMENTS

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)

8. REFERENCES

- [1] E. Alba and F. Chicano. ACOhg: Dealing with huge graphs. In *Proc. of GECCO*, pages 10–17, 2007.
- [2] E. Alba and F. Chicano. Finding safety errors with ACO. In *Proc. of GECCO*, pages 1066–1073, 2007.
- [3] B. Alpern and F. B. Schneider. Defining liveness. *Inform. Proc. Letters*, 21:181–185, 1985.
- [4] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.
- [5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, January 2000.
- [6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, 1977.
- [7] M. Dorigo and T. Stützle. *Ant Colony Optimization*. The MIT Press, 2004.
- [8] S. Edelkamp, A. L. Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation Intelligence*, pages 75–83, 2001.
- [9] S. Edelkamp, S. Leue, and A. 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.
- [10] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of IFIP/WG6.1 Symposium on Protocol Specification, Testing, and Verification (PSTV95)*, pages 3–18, Warsaw, Poland, June 1995.
- [11] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *Intl. Jnl. on Soft. Tools for Tech. Transfer*, 6(2):117–127, 2004.
- [12] A. Groce and W. Visser. Heuristics for model checking Java programs. *Intl. Jnl. on Software Tools for Technology Transfer*, 6(4):260–276, 2004.
- [13] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.
- [14] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32, 1996.
- [15] M. Kamel and S. Leue. Validation of a remote object invocation and object migration in CORBA GIOP using Promela/Spin. In *Intl. SPIN Workshop*, 1998.
- [16] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Trans. on Soft. Eng.*, 27(12):1085–1110, 2001.
- [17] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 2007.
- [18] W. Zhang. Model checking operator procedures. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 200–215. Springer-Verlag, 1999.