Finding Safety Errors with ACO

Enrique Alba
GISUM Group, Dpto. Lenguajes y
Ciencias de la Computación
E.T.S. Ingeniería Informática
University of Málaga, Spain
eat@lcc.uma.es

Francisco Chicano
GISUM Group, Dpto. Lenguajes y
Ciencias de la Computación
E.T.S. Ingeniería Informática
University of Málaga, Spain
chicano@lcc.uma.es

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 model checkers found in the literature use exact deterministic algorithms to check the properties. These algorithms usually require huge amounts of computational resources if the checked model is large. We propose here the use of a new kind of Ant Colony Optimization (ACO) model, ACOhg, to refute safety properties in concurrent systems. ACO algorithms are stochastic techniques belonging to the class of metaheuristic algorithms and inspired by the foraging behaviour of real ants. The traditional ACO algorithms cannot deal with the model checking problem and thus we use ACOhg to tackle it. The results state that ACOhg algorithms find optimal or near optimal error trails in faulty concurrent systems with a reduced amount of resources, outperforming algorithms that are the state-of-the-art in model checking. This fact makes them suitable for checking safety properties in large concurrent systems, in which traditional techniques fail to find errors 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

Ant colony optimization, metaheuristics, SPIN, HSF-SPIN

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'07, July 7–11, 2007, London, England, United Kingdom.
Copyright 2007 ACM 978-1-59593-697-4/07/0007 ...$5.00.

1. INTRODUCTION

From the very beginning of computer research, computer engineers have been interested in techniques allowing them to know if a software module fulfills 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. 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. Two very well-known logics used in this verification are *predicate calculus* and *Hoare logic*. However, formal verification using logics is not fully automatic. Although automatic theorem provers can assist the process, human intervention is still needed.

Model checking [6] 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. This property is specified using a temporal logic like Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). One of the best known explicit model checkers is SPIN [14], which takes a software model codified in Promela and a property specified in LTL as inputs. SPIN transforms the model and the negation of the LTL formula into Büchi automata in order to perform the synchronous product of them. The resulting product 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 [15] for more details). If such kind of cycle does not exist then the system fulfills the property and the verification ends with success.

The amount of states of the product automaton is very high 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 a model checker can verify. This limit is reached when it is not able to explore more states due to the absence of free memory. Several techniques exist to alleviate this problem. They reduce the amount of memory required for the search by following different approaches. On one hand, there are techniques which reduce the number of states to explore, such as partial order reduction [20] and symmetry reduction [19]. 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 [15]. Symbolic model checking [5] is another very popular alternative to the explicit state model checking that can reduce the amount of memory required for the verification by means of a compact representation for set of states. However, although it is possible to check larger models with symbolic model checking, this technique also suffers from state explosion.

In this work we propose the use of a new model of Ant Colony Optimization (ACO) [7] for finding counterexamples (refuting) of LTL formulae in concurrent systems. ACO algorithms belong to the metaheuristic class of algorithms [4], which are able to find near optimal solutions using a reasonable amount of resources. For this reason, they can be suitable for searching accepting states in the graph of large system models, for which traditional exploration algorithms fail. However, existing ACO models cannot be applied to the problem of finding error trails in concurrent systems due to the large size of the corresponding Büchi automata and thus a new ACO model was proposed in [2].

The paper is organized as follows. In the next section we present previous algorithms used for state explicit model checking. The heuristic search is carefully analyzed because our proposal is based on it and we dedicate Section 3 to show some previously used heuristics. In Section 4 the ACO model used in this paper, ACOhg, is described. In Section 5 we present some experimental results comparing the ACOhg-based algorithms against traditional exact algorithms for explicit state model checking. We also compare one ACOhg algorithm against a Genetic Algorithm (GA), which has been previously used for this task. Finally, Section 6 outlines the conclusions and future work.

2. BACKGROUND

For the verification of a general LTL formula in explicit state model checking it is necessary to search for a cycle in the state graph with at least one accepting state. Furthermore, such a cycle must be reachable from the initial state. For this task, SPIN uses the Nested Depth First Search algorithm (Nested-DFS) [16]. The algorithm first tries to find an accepting state using Depth First Search. When found, it tries to reach the same state starting on it, that is, it searches for a cycle including the found accepting state. If this cycle is not found, it searches for another accepting state starting on the initial node and repeat the process. As we said before, if the cycle is found, it represents a counterexample for the LTL formula. Otherwise, there is no counterexample and the checked model fulfills the LTL formula. The reason is that Nested-DFS is an exhaustive algorithm: if it does not find an accepting reachable cycle, there is no such kind of cycle. On the contrary, most of the canonical metaheuristic algorithms [4], due to their approximate nature, cannot ensure that the system fulfills the property, but they can refute it. For this reason we talk about a problem of properties refutation instead of verification when metaheuristic algorithms are used.

The properties that can be specified with LTL formulae can be classified into two groups: *safety* and *liveness* properties [21]. Safety properties can be expressed as assertions that must be fulfilled by all the states of the model, while liveness properties refer to assertions that must fulfilled by execution paths in the model. Safety properties of a model can be checked by searching for a single accepting state in the product Büchi automaton. That is, when safety properties are checked, it is not required to find an additional cycle containing the accepting state. This means that safety properties verification can be transformed into a search for one objective node (one accepting state) in a graph (Büchi automaton). Furthermore, the path from one initial node to one objective node represents an execution of the concurrent system in which the given safety property is violated: an error trail. Short error trails in faulty system models are preferred to long ones. The reason is that a human programmer analyzing the error trail can understand a short trail in less time than a long one.

The simplification of the graph exploration when dealing with safety properties has been used in previous works to verify safety properties using classical algorithms in the graph exploration domain. Edelkamp, Lluch-Lafuente, and Leue [8, 9, 10] apply Depth First Search (DFS) and Breadth First Search (BFS) to the problem of verifying safety properties using SPIN. Furthermore, they use heuristic search for this task in their own tool called HSF-SPIN, an extension of SPIN. In order to perform a heuristic search they assign to every state a heuristic value that depends on the property to verify. They apply 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 and the amount of memory required to obtain an error trail is reduced, allowing the exploration of larger models. In addition, they show that the use of heuristic search can be combined with partial order reduction [20] and symmetry reduction [19]. They also use heuristic search to guide the search for counterexamples of liveness properties [8, 10].

Genetic Algorithms (GAs) have also been applied to the problem of refuting safety properties in concurrent systems. In an early proposal, Alba and Troya [3] used GAs for detecting deadlocks, useless states, and useless transitions in communication protocols. In their work, one path in the state graph is represented by a finite sequence of numbers. For the evaluation of a solution, a protocol simulator follows the suggested path and accounts for the number of states and transitions not used in the Finite State Machines during the simulation. To the best of our knowledge, this is the first application of a metaheuristic algorithm to model checking. Later, Godefroid and Kurshid [12, 13], in an independent work, applied GAs to the same problem using a similar encoding of the paths in the chromosome. Their GA is integrated with VeriSoft [11] and it can check C programs.

In the present work we propose the utilization of another metaheuristic algorithm: Ant Colony Optimization. Unlike GA, ACO is a metaheuristic designed for searching short paths in graphs. This makes it very suitable for the problem at hand. In order to guide the search we use the same heuristic functions defined by Edelkamp et al. [8]. In fact, we have extended their tool, HSF-SPIN, in order to include our ACO 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.

3. HEURISTIC FUNCTIONS

In order to guide the search, a heuristic value is associated to every automaton state. The computation of this value can be based on the LTL formula [9] or in the objective node (if known beforehand) [19]. Formula-based heuristics are a kind of heuristic functions that can be applied when the objective state is not known. Using the logic expression that must be false in an objective node, these heuristics estimate the number of transitions required to get an objective node from the current one. Given a logic formula φ (without temporal operators), the heuristic function for that formula H_φ is defined using its subformulae. Table 1 shows the recursive definition of a formula-based heuristic.

Table 1: Formula-based heuristic function

φ	$H_\varphi(s)$	$\overline{H}_\varphi(s)$
$true$	0	∞
$false$	∞	0
p	**if** p **then** 0 **else** 1	**if** p **then** 1 **else** 0
$a \otimes b$	**if** $a \otimes b$ **then** 0 **else** 1	**if** $a \otimes b$ **then** 1 **else** 0
$\neg\psi$	$\overline{H}_\psi(s)$	$H_\psi(s)$
$\psi \vee \xi$	$\min\{H_\psi(s), H_\xi(s)\}$	$\overline{H}_\psi(s) + \overline{H}_\xi(s)$
$\psi \wedge \xi$	$H_\psi(s) + H_\xi(s)$	$\min\{\overline{H}_\psi(s), \overline{H}_\xi(s)\}$
$full(q)$	$capa(q) - len(q)$	**if** $full(q)$ **then** 1 **else** 0
$empty(q)$	$len(q)$	**if** $empty(q)$ **then** 1 **else** 0
$q?[t]$	minimal prefix of q without t	**if** $head(q) \neq t$ **then** 0 **else** maximal prefix of t's
$i@s$	$D_i(pc_i, s)$	**if** $pc_i = s$ **then** 1 **else** 0

ψ, ξ: formulae without temporal operators
p: logic proposition
a, b: variables or constants
$\otimes$: relational operator ($=, \neq, <, \leq, \geq, >$)
q: queue
$capa(q)$: capacity of queue q
$len(q)$: length of queue q
$head(q)$: message in the head of queue q
t: tag of a message
i: process
s: state of a process automaton
pc_i: current state of process i in its corresponding automaton
$D_i(u, v)$: minimum number of transitions for reaching v from u in the local automaton of process i

Formula-based heuristic functions can be used to guide the search for safety LTL properties. For searching deadlocks several heuristic functions can be used. On one hand, the number of active processes can be used as heuristic value of a state. We denote this heuristic as H_{ap}. On the other hand, the number of executable (enabled) transitions in a state can also be used as heuristic value, denoted with H_{ex}. Another option consists in approximating the deadlock situation with a logic predicate and deriving the heuristic function of that predicate using the rules of Table 1 (see [9]).

From the heuristic functions that can be used when the objective node is known we can highlight the *Hamming distance* and the *Finite State Machines* (FSM) distance. 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 in the local transition graphs of the processes required to reach the local objective state from the local current state [19]. As we will see in the experimental section, in this paper we search for unknown objective states, so we do not use any of these two last heuristic functions.

4. ANT COLONY OPTIMIZATION

The ACO Metaheuristic is inspired by the foraging behaviour of real ants. The main idea consists of simulating the ants' behaviour in a graph (the so-called *construction graph*) in order to search for the lowest cost path from an initial node to an objective one. The cooperation among the different simulated ants is a key factor in the search. This cooperation is performed indirectly by means of *pheromone trails*, which is a model of the chemicals the real ants use for their communication.

```
procedure ACOMetaheuristic
    ScheduleActivities
        ConstructAntsSolutions
        UpdatePheromones
        DaemonActions // optional
    end ScheduleActivities
end procedure
```

Figure 1: Pseudo-code of the ACO Metaheuristic.

In Figure 1 we reproduce a general ACO pseudo-code found in [7]. It consists of three procedures executed during the search: `ConstructAntsSolutions`, `UpdatePheromones`, and `DaemonActions`. They are executed until a given stopping criterion is fulfilled, such as finding a solution or reaching a given number of steps. In the first procedure each artificial ant follows a path in the construction graph. The ant starts in an initial node and then it stochastically selects the next node according to the pheromone and the heuristic value associated with each arc (or the node itself). The ant appends the new node to the traversed path and selects the next node in the same way. This process is iterated until a candidate solution is built. In our case, when ant k is in node i it selects node j with probability

$$p_{ij}^{k} = \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l \in N_i} [\tau_{il}]^{\alpha}[\eta_{il}]^{\beta}}, \text{ if } j \in N_i, \tag{1}$$

where N_i is the set of successor nodes for node i, and α and β are two parameters of the algorithm determining the relative influence of the heuristic value and the pheromone trail on the path construction, respectively.

In the `UpdatePheromones` procedure, pheromone trails associated to arcs are modified. In our case, the pheromone trails associated to the arcs that ants traverse are updated during the construction phase using the expression

$$\tau_{ij} \leftarrow (1 - x_i)\tau_{ij} \tag{2}$$

where x_i controls the evaporation of the pheromone during the construction phase, where $0 \leq x_i \leq 1$. This mechanism increases the exploration of the algorithm, since it reduces the probability that an ant follows the path of a previous ant. After the construction phase pheromone trails are updated again in order to take into account the quality of the candidate solutions built by the ants. In this case the pheromone update follows the expression

$$\tau_{ij} \leftarrow \rho\tau_{ij} + \Delta\tau_{ij}^{bs}, \ \forall (i, j) \in L, \tag{3}$$

where ρ is the *pheromone evaporation rate* and it holds that $0 \leq \rho \leq 1$. On the other hand, $\Delta\tau_{ij}^{bs}$ is the amount of pheromone that the best ant path ever found deposits on arc (i, j). This quantity is usually in direct relation with the quality of the solution. In our case, we try to minimize an objective function (also called fitness function) and thus we set $\Delta\tau_{ij}^{bs}$ to the inverse of the minimum fitness value found. In addition, we adopt here the idea introduced in Max-Min Ant Systems ($\mathcal{MM}$AS) of 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

$$\tau_{max} = \frac{Q}{1-\rho} \qquad (4)$$

$$\tau_{min} = \frac{\tau_{max}}{a} \qquad (5)$$

where Q is the inverse of the minimum fitness value. The parameter a controls the size of the interval. When one pheromone trail is greater than τ_{max} it is set to τ_{max} and, in a similar way, when it is lower than τ_{min} it is set to τ_{min}. Each time a new better solution is found the interval limits are updated consequently and all pheromone trails are checked in order to keep them inside the interval.

Finally, the last (and optional) procedure `DaemonActions` performs centralized actions that are not performed by individual ants. For example, a local optimization algorithm can be implemented in this procedure in order to improve the tentative solution held in every ant.

4.1 Modifications to the basic models: ACOhg

The ACO models we found in the literature can be applied (and they have been) to problems with a number of nodes n of several thousands. In these problems the construction graph has a number of arcs of $O(n^2)$, that is, several millions of arcs, and hence the pheromone trails require several megabytes of memory in a computer to be stored. These models are not suitable in problems in which the construction graph has more than 10^6 nodes (i.e. 10^{12} arcs). They are also not suitable when the amount of nodes is not known beforehand and the nodes and arcs of the construction graph are dynamically generated as the search progresses. We tackle here such a kind of problem. In effect, the number of states of a concurrent system is usually very large even in small models. For example, the number of states of the dining philosophers model used in Section 5 is 3^n, where n is the number of philosophers. That is, the number of states grows in an exponential way with respect to the size of the model.

Let us discuss the issues that prevent existing ACO models from solving such kind of problems. First, in the construction phase, ants of a regular ACO walk until a candidate solution is completed. However, if we would allow the ants to walk on the huge unknown graph without repeating a node until they find an objective node they can reach a dead end (a node without non-visited successors). Even although they find an objective node they can wander in the graph for a long time requiring a lot of memory to build a candidate solution since the objective nodes can be very far from the initial node. Thus, in general it is not viable to work with complete candidate solutions as current models do. We must allow the construction of partial candidate solutions. Second, some ACO models assign to the initial pheromone trails a value that depends on the number of graph nodes. This kind of initialization of the pheromone trails is not suitable when we work with unknown sized graphs.

In order to solve these obstacles that arise when working with large graphs, we use here a new ACO model called ACOhg (ACO for huge graphs) presented in [2] that is able to tackle combinatorial optimization problems with an underlying construction graph of unknown size that is built as the search progresses. The main ideas this model introduces aim at exploring the construction graph with a small amount of memory. We detail the ideas in the following paragraphs.

In order to avoid the, in general unviable, construction of complete candidate solutions we limit the length of the paths traversed by ants in the construction phase. That is, when the path of an ant reaches a given maximum length λ_{ant}, the ant is stopped. In this way, the construction phase can be performed in a bounded time and with a bounded amount of memory. However, the limitation of the ant path length implies that most (if not all) of the paths are partial solutions and therefore we need a fitness function that can evaluate partial solutions (what, in some cases, can be far from trivial).

The limitation in the ant path length solves the problem of the "wandering ants" but introduces a new one. There is a new parameter for the algorithm (λ_{ant}) whose optimal value is not easy to establish a priori. If we select a value smaller than the depth[1] of all the objective nodes, the algorithm will not find any solution to the problem. Thus, we must select a value larger than the depth of one objective node (if known). This is not difficult when we know where the objective node is, but the usual situation is the opposite one. In the last case, two alternatives are proposed. In Figure 2 we show graphically the way in which the two alternatives work.

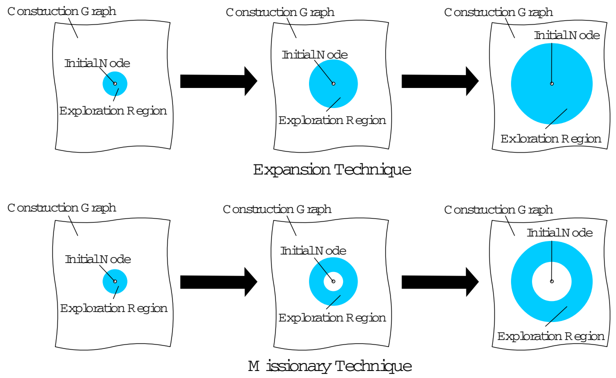

Figure 2: Two alternatives for reaching objective nodes of unknown depth: expansion and missionary techniques. We show snapshots of different moments of the search.

The first consists in dynamically increasing λ_{ant} during the search if no objective node is found. At the beginning, a low value is assigned to λ_{ant} and it is increased in a given quantity δ_l every certain number of steps σ_i. In this way, the length will be hopefully high enough to reach at least one objective node. This is called *expansion technique*. This mechanism can be useful when the depth of the objective nodes is not very high. Otherwise, the length of the ant paths will increase a lot and the same happens with the time and the memory required to build the paths, since it will approach the behaviour of a regular ACO incrementally.

The second alternative consists in starting the path construction of the ants from different nodes during the search. That is, at the beginning the ants are placed on the initial nodes of the graph and the algorithm is executed during a given number of steps σ_s (called *stage*). If no objective node

[1] The *depth* of a node in the construction graph is the length of the shortest path from an initial node to it.

is found, the last nodes of the paths constructed by the ants are used as starting nodes for the next ants. In the next steps (the second stage) of the algorithm the new ants traverse the graph starting in the last nodes of paths computed in the first stage. In this way, the new ants start at the end of previous ant paths trying to go beyond in the graph. This mechanism is called *missionary technique*. The length of the ant paths (λ_{ant}) is kept always constant and the pheromone trails are discarded from one stage to another in order to keep almost constant the amount of computational resources (memory and CPU time) in all the stages. The assignment of ants to starting nodes at the beginning of one stage is performed in several phases. First, we need to select the solutions of the previous stage whose last visited nodes will be used as starting points in the new stage. For this, we store the best solutions (according to the fitness value) found in the previous stage. We denote with s the number of solutions stored. Once we have the set of starting nodes we need to assign the new ants to those nodes. For each new ant we select its starting node using roulette selection; that is, the probability of selecting one node is proportional to the fitness value of the solution associated with it.

5. EXPERIMENTAL SECTION

In this section we present some results obtained with the ACOhg algorithms. For the experiments we have selected 5 Promela models that are presented in the following section. After that, we discuss the algorithm parameters used in the experiments in Section 5.2. In Section 5.3 we show the first results of the ACOhg algorithms and we compare them against the obtained by exhaustive algorithms. Next, in Section 5.4 we compare an ACOhg algorithm against a Genetic Algorithm, the previous metaheuristic algorithm applied to find errors in concurrent systems.

5.1 Models

We have selected 5 Promela models previously reported in the literature by Edelkamp et al. [10]. All these models violate a safety property. In Table 2 we present the models with some information about them. They can be found in `http://web.tiscali.it/ikaria/alberto` together with the HSF-SPIN source code.

Table 2: Promela models used in the experiments

Model	LoC	States	Processes	Safety prop.
giop22	717	unknown	11	Deadlock
marriers4	142	unknown	5	Deadlock
needham	260	18242	4	LTL formula
phi16	34	*43046721	17	Deadlock
pots	453	unknown	8	Deadlock

* Theoretical result.

From the models the smallest one is `needham`. The remaining models have a large associated Büchi automaton that does not fit in the main memory of the machines used for the experiments (512MB). The first model, `giop22`, is an implementation of the CORBA Inter-ORB protocol for 2 clients and 2 servers [18]. The next model, `marriers4` is a protocol solving the stable marriage problem for 4 suitors [22]. The Needham-Schroeder protocol [23] with one initiatior, one responder, and one intruder is implemented in `needham` with the objective of finding the Lowe attack. The Dijkstra dining philosophers problem is implemented in `phi16` with 16 philosophers. The Plain Old Telephone Service model is implemented in `pots` [17].

5.2 Algorithms and Parameters

For the experiments we use ACOhg algorithms with the configuration shown in Table 3. The missionary technique is used. These parameters are not set in an arbitrary way, they are the result of a previous study aimed at finding the best configuration for tackling the models shown in the previous section. That is, we selected a configuration that obtains a good trade-off between efficacy of the algorithm and resources used. It is worth mentioning that one configuration that is good for one model can be bad for another one. In fact, this happens here, so the configuration shown in Table 3 was selected in order to get a good trade-off between efficacy and resources in all the models simultaneously.

Table 3: Parameters for the ACOhg

Parameter	Value
Steps	100
Colony size	10
λ_{ant}	10
σ_s	2
s	10
x_i	0.5
a	5
ρ	0.8
α	1.0
β	2.0

With respect to the heuristic information, η_{ij}, we use $\eta_{ij} = 1/(1+H_{\varphi}(j))$ where $H_{\varphi}(j)$ is the formula-based heuristic evaluated in state j when the objective is to find a counterexample of an LTL formula (`needham`). In the case of deadlock detection, we use $\eta_{ij} = 1/(1 + H_{ap}(j))$ where $H_{ap}(j)$ is the active processes heuristic evaluated in state j (see Section 3). However, we use two versions of ACOhg: one not using heuristic information and another one using it. The fitness value (to minimize) of a solution is the sum of the solution length, the heuristic value of the last state, and a penalty term for partial solutions. The stopping criterion used in our ACOhg algorithms is to find an error trail or to reach a maximum number of allowed steps (100). The algorithm could follow the search after an error trail is found in order to optimize the length of the error trail. However, we are interested here in observing the effort required by the algorithm for obtaining an error trail, since it points out an error in the model, and thus reducing the error trails is deferred for a future work.

Since ACOhg is a stochastic algorithm, we need to perform several independent runs in order to get an idea of the behaviour of the algorithm. In the specialized literature it is well established that a minimum of 30 independent runs is required to get statistical confidence of the results. In our experiments we perform 100 independent runs in order to get a high statistical confidence. The machine used in the experiments is a Pentium IV at 2.8 GHz with 512 MB of RAM. 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 swap area, which could affect significantly the CPU time required in the search.

5.3 ACOhg vs. Exact Algorithms

Here we present the first results obtained with ACOhg. We compare these results against exact algorithms previously found in the literature. These algorithms are BFS, DFS, A*, and BF (see Section 2). BFS and DFS do not use heuristic information while the other two do. In order to make a fair comparison we use two different ACOhg algorithms: one not using heuristic information (ACOhg-b) and another one using it (ACOhg-h). We compare ACOhg-b against BFS and DFS in Table 4, and ACOhg-h against A* and BF in Table 5. In the tables we can see the hit rate (number of executions that got an error trail), the length of the error trails found (number of states), the memory required (in Kilobytes), the number of expanded states, and the CPU time used (in milliseconds) by each algorithm. For ACOhg-b and ACOhg-h we show average values over 100 independent runs.

Table 4: Results of the algorithms without heuristic information

Models	Aspects	BFS	DFS	ACOhg-b
giop22	hit rate	0/1	1/1	100/100
	len (states)	-	112.00	45.80
	mem (KB)	-	3945.00	4814.12
	exp (states)	-	220.00	1048.52
	cpu (ms)	-	30.00	113.60
marriers4	hit rate	0/1	0/1	57/100
	len (states)	-	-	92.18
	mem (KB)	-	-	5917.91
	exp (states)	-	-	2045.84
	cpu (ms)	-	-	257.19
needham	hit rate	1/1	1/1	100/100
	len (states)	5.00	11.00	6.39
	mem (KB)	23552.00	62464.00	5026.36
	exp (states)	1141.00	11203.00	100.21
	cpu (ms)	1110.00	18880.00	262.00
phi16	hit rate	0/1	0/1	100/100
	len (states)	-	-	31.44
	mem (KB)	-	-	10905.60
	exp (states)	-	-	832.08
	cpu (ms)	-	-	289.40
pots	hit rate	1/1	1/1	49/100
	len (states)	5.00	14.00	5.73
	mem (KB)	57344.00	12288.00	9304.67
	exp (states)	2037.00	1966.00	176.47
	cpu (ms)	4190.00	140.00	441.63

The first observation that we can make from the results of Table 4 is that ACOhg-b is the only algorithm able to find an error in all the models. DFS fails in `marriers4` and `phi16`, while BFS fails in these two models and in `giop22`. The reason for these fails is that the memory required by the algorithms exceeds the memory available on the computer. Furthermore, ACOhg-b finds an error in all the independent runs (hit rate of 100%) for 3 out of the 5 models. In view of these results we can state that ACOhg-b is better than DFS and BFS in the task of searching for errors.

Concerning the quality of solutions (the length of error trails), we observe that ACOhg-b obtains almost optimal (minimal) error trails. The optimal length for error trails are those obtained by BFS (when it finds an error), since it is designed to obtain an optimal error trail. The length of the error trails found by DFS are much longer (bad quality) than those of the other two algorithms (BFS and ACOhg-b).

Let us discuss now the computational resources used by the algorithms. With respect to the memory used, ACOhg-b requires less memory than BFS in all the models. In some models the difference is very large. For example, in `needham` BFS requires more than 4 times the memory of ACOhg-b (and more than 4 times its CPU time). The memory used by ACOhg-b is also less than the one required by DFS for 4 out of 5 with an exception for `giop22`. The number of expanded states of ACOhg-b is the minimum in 4 out of 5 models. Only DFS is able to reduce the number of expanded states in one model: `giop22`. Observing the CPU time required by the algorithms we can notice that BFS is the slowest algorithm (this is the price of its optimality). DFS is faster than ACOhg-b in `giop22` and `pots` (between 3 and 4 times faster) but ACOhg-b is much faster than DFS in `needham` (72 times faster).

In general terms, we can state that ACOhg-b is a robust algorithm that is able to find errors in all the proposed models with a low amount of memory. In addition, it combines the two good features of BFS and DFS: it obtains short error trails, like BFS, while at the same time requires a reduced CPU time, like DFS.

Table 5: Results of the algorithms using heuristic information

Models	Aspects	A*	BF	ACOhg-h
giop22	hit rate	1/1	1/1	100/100
	len (states)	44.00	44.00	44.20
	mem (KB)	417792.00	2873.00	4482.12
	exp (states)	83758.00	168.00	1001.78
	cpu (ms)	46440.00	10.00	112.40
marriers4	hit rate	0/1	1/1	84/100
	len (states)	-	108.00	86.65
	mem (KB)	-	41980.00	5811.43
	exp (states)	-	9193.00	1915.30
	cpu (ms)	-	190.00	233.33
needham	hit rate	1/1	1/1	100/100
	len (states)	5.00	10.00	6.12
	mem (KB)	19456.00	4149.00	4865.40
	exp (states)	814.00	12.00	87.47
	cpu (ms)	810.00	20.00	229.50
phi16	hit rate	1/1	1/1	100/100
	len (states)	17.00	81.00	23.08
	mem (KB)	2881.00	10240.00	10680.32
	exp (states)	33.00	893.00	587.53
	cpu (ms)	10.00	40.00	243.80
pots	hit rate	1/1	1/1	99/100
	len (states)	5.00	7.00	5.44
	mem (KB)	57344.00	6389.00	6974.56
	exp (states)	1257.00	695.00	110.48
	cpu (ms)	6640.00	50.00	319.49

Let us focus on the results of Table 5. Again, ACOhg-h is able to find the design errors in all the models (as BF). We observe that A* fails to find an error state in `mariers4`. We can state (comparing Tables 5 and 4) that A* and BF outperform the results of BFS and DFS: A* and BF can find errors in almost all the models obtaining shorter error trails and using less resources than DFS and BFS. The reason is the use of heuristic information in A* and BF. We can say the same for the ACOhg algorithms: ACOhg-h obtains higher hit rate than ACOhg-b due to the heuristic information (we can notice this in `mariers4` and `pots` models, since for the remaining models 100% of hit rate is obtained using both algorithms). Thus, we can state that heuristic information has a positive influence on the search. In spite of this fact, most of the current popular model checkers do not use heuristic information during the search.

With respect to the solution quality, we observe in Table 5 that ACOhg-h obtains almost optimal error trails, similar to that of the A* algorithm (that are optimal since the heuristic used is admissible). In addition, the error trails of ACOhg-h are shorter than the ones of BF in 4 out of the 5 models.

If we focus on the memory required by the algorithms, ACOhg-h usually requires less memory than A* (except for `phi16`) but more than BF (except for `marriers4`). On the other hand, ACOhg-h expands less states than A* and BF in 2 out of the 5 models. There exists a relationship between expanded states and memory used in A* and BF: the more the number of expanded states, the more the number of states stored in main memory. Since the states stored in memory are the main source of memory consumption, we expect that the memory required by A* and BF be higher when more states are expanded. We can notice this fact in Table 5. However, this statement does not necessarily hold for ACOhg-h, since in this last algorithm there is another important source of memory consumption: the pheromone trails. For this reason we can observe that ACOhg-h requires more memory than BF in `phi16` and `pots` in spite of the fact that ACOhg-h expands less states (in average) than BF.

In general, we can state that ACOhg-h is the best trade-off between solution quality and memory required: it obtains almost optimal solutions with a reduced amount of memory.

In order to summarize the results discussed in this section we present in Figure 3 the quality of the solutions (length of error trails) plotted against the memory required by all the algorithms in all the models. Since different models have different optimal lengths and different memory requirements, we plot normalized values of the length and the memory consumption. In this way, we can keep all the points in the same graph and we can compare the algorithms globally (without restricting the discussion to a specific model). For each model we divide the length of the error trails obtained by all the algorithms by the minimum length value obtained by any algorithm for this model. The same normalization is performed for the memory.

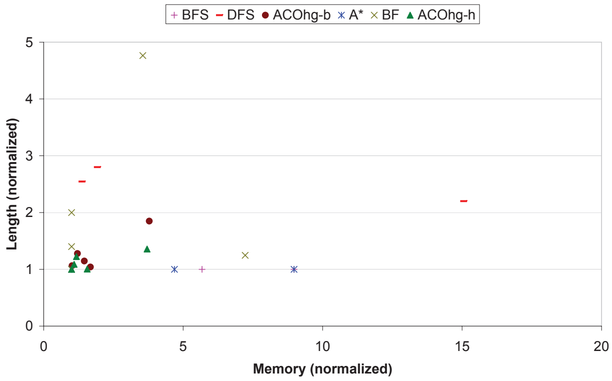

Figure 3: Normalized length of error trails against normalized memory required by all the algorithms in all the models.

We can observe in Figure 3 that the results of the ACOhg algorithms are all focused on the left bottom corner of the plot. That is, ACOhg algorithms are able to get short error trails (good quality solutions) with a low amount of memory. Furthermore, ACOhg algorithms are the only ones that keep all their points on a good-quality region. The points of the remaining algorithms are scattered in the plot (in fact, one point belonging to A* has been omitted in the plot because it is out of the area shown in the figure). This means that ACOhg algorithms are the most robust algorithms of the experiments. They have a similar behaviour for all the models. On the contrary, the behaviour of the exact state-of-the-art algorithms depends to a large extent on the model they solve. In view of this fact, we point out the ACOhg algorithms as very promising techniques for finding errors in concurrent systems.

5.4 ACOhg vs GA

In this section we compare the ACOhg-h algorithm against the GA used by Godefroid and Kurshid in [13]. We seek to find a deadlock in the dining philosophers problem with 17 philosophers and the Lowe attack in the Needham-Schroeder protocol as they do in their work. The main differences between their experiments and ours are the machine used, the programming language of the models (C in their case and Promela in ours), and the tool used for the verification (VeriSoft and HSF-SPIN, respectively). In Table 6 we show the hit rate, CPU time (in seconds), and memory required (in Kilobytes) by the algorithms in the two models.

Table 6: Comparison of ACOhg-h against a GA

Model	Algorithm	Hit (%)	Time (s)	Mem. (KB)
phi17	GA	52	197.00	n/a
	ACOhg-h	100	0.28	11274
needham	GA	3	3068.00	n/a
	ACOhg-h	100	0.23	4865

We can observe that ACOhg-h is able to find always an error (100% hit rate) while GA finds an error only in 52% of the cases in `phi17` and 3% in `needham`. We can state that ACOhg-h has better efficacy than GA in these two models. This confirms our hypothesis that ACO metaheuristic is especially suitable for this problem because it is designed for searching paths in graphs.

With respect to the execution time we can observe a large difference between the algorithms. This large difference cannot be only explained by the different machines used in the experiments. The GA is executed on a Pentium III at 700 MHz with 256 MB of RAM and the ACOhg-h is executed on a Pentium 4 at 2.8 GHz with 512 MB of RAM. The maximum memory required by ACOhg-h is 11 MB that is smaller than 256 MB. This means that if ACOhg-h were executed in the machine used in [13] for GA, no virtual memory would be used and the required CPU time would not be much more than four times the CPU time required in the Pentium 4 (since the CPU clock is four times faster in the Pentium 4). Unlike this, the CPU time required by ACOhg-h would be less than the time required by GA in two or three order of magnitude. Thus, we conclude that ACOhg-h is more accurate and faster than GA for these models and we conjecture that this can be extended, in general, to the problem of refutation of safety properties in concurrent systems.

6. CONCLUSIONS AND FUTURE WORK

We have presented here a novel application of a new Ant Colony Optimization model, called ACOhg, to the problem of checking safety properties in concurrent systems. We have compared the ACOhg algorithms against the state-of-the-art exhaustive methods and the results show that ACOhg algorithms are able to outperform the state-of-the-art algorithms in efficacy and efficiency. They require a very low amount of memory and CPU time and are able to find errors even in models in which the state-of-the-art algorithms fail because of the high amount of memory required. The results also show that ACOhg algorithms are definitely better than GA, which is, to the best of our knowledge, the only other metaheuristic previously used in the past for checking safety properties on real software.

ACOhg algorithms 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. We also plan to extend the application of ACOhg to the search for liveness properties violations. This can be done in two phases. First the algorithm searches an accepting state and then it tries to find a path to this state from itself. We can also 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 [8]. In the present paper it has been shown that the way in which ACOhg performs the search is very useful for finding design errors. However, ACOhg does not ensure the correctness of the models in which no error is found. We plan to use ACOhg as the base of an exhaustive stochastic algorithm that is able to ensure the correctness when no error is found while at the same time is able to find errors very fast when they exist.

Model checkers working in parallel on a cluster of machines are gaining importance in the formal methods community nowadays. In addition, a lot of work exists stating the high efficiency and efficacy of parallel metaheuristics [1]. We plan to design a parallel version of ACOhg for reducing the time required and increasing the available memory, thus able to work with even larger models. We also want to integrate the algorithm inside Java PathFinder, which is able to work with programs in Java language, much more familiar for the computer science community than Promela.

7. ACKNOWLEDGEMENTS

This work has been partially funded by the Ministry of Education and Science and FEDER under contract TIN2005-08818-C04-01 (the OPLINK project). Francisco Chicano is supported by a grant (BOJA 68/2003) from the Junta de Andalucía.

8. REFERENCES

[1] E. Alba, editor. *Parallel Metaheuristics. A New Class of Algorithms.* Wiley, 2005.

[2] E. Alba and F. Chicano. Una versión de ACO para problemas con grafos de muy gran extensión. In *Actas del Quinto Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, MAEB 2007*, pages 741–748, Tenerife, Spain, February 2007.

[3] E. Alba and J. Troya. Genetic algorithms for protocol validation. In *Proc. of the PPSN IV International Conference (LNCS 1141)*, pages 870–879, Berlin, 1996. Springer.

[4] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[5] 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*, 13(4), April 1994.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, January 2000.

[7] M. Dorigo and T. Stützle. *Ant Colony Optimization.* The MIT Press, 2004.

[8] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *LNCS, 2057*, pages 57–79. Springer, 2001.

[9] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation Intelligence*, pages 75–83, 2001.

[10] 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*, 5:247–267, 2004.

[11] P. Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Proceedings of the 9th Conference on Computer Aided Verification, LNCS 1254*, pages 476–479, 1997.

[12] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *LNCS, 2280*, pages 266–280. Springer, 2002.

[13] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *International Journal on Software Tools for Technology Transfer*, 6(2):117–127, 2004.

[14] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):1–17, May 1997.

[15] G. J. Holzmann. *The SPIN Model Checker.* Addison-Wesley, 2004.

[16] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32. American Mathematical Society, 1996.

[17] M. Kamel and S. Leue. Vip: a visual editor and compiler for v-promela. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1785*, pages 471–486. Springer.

[18] 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.

[19] A. Lluch-Lafuente. Symmetry reduction and heuristic search for error detection in model checking. In *Workshop on Model Checking and Artificial Intelligence*, August 2003.

[20] 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, April 2002. Springer.

[21] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[22] D. G. McVitie and L. B. Wilson. The stable marriage problem. *Commun. ACM*, 14(7):486–490, 1971.

[23] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.