Diseño Inteligente de Redes Inalámbricas de Comunicación
Conserjería de Innovación, Ciencia y Empresa
P07-TIC-03044
http://diricom.lcc.uma.es

Málaga, 22 de Noviembre de 2008

Informe Ejecutivo

Título: SOFTW-1.0-2008: Optimización basada en colonias de hormigas aplicada a model checking

Resumen: Model checking es una técnica formal completamente automática para comprobar propiedades en programas. La mayoría de los model checkers usados en la literatura usan algoritmos exactos deterministas para comprobar estas propiedades. Estos algoritmos requieren normalmente una gran cantidad de recursos computacionales si el modelo es grande. En este entregable presentamos la aplicación de un nuevo tipo de ACO (optimización basada en colonias de hormigas), ACOhg, para refutar propiedades de seguridad en sistemas concurrentes. Los ACOs son algoritmos estocásticos pertenecientes a la clase de las metaheurísticas y su funcionamiento está inspirado en el comportamiento de las hormigas reales cuando buscan comida. Los ACO tradicionales no pueden abordar el problema que proponemos y por ese motivo usamos ACOhg. Los resultados muestran que ACOhg encuentra trazas de error óptimas o casi óptimas con una cantidad reducida de recursos, superando algoritmos que son el estado del arte en model checking.

Objetivos:

1. Aplicar ACOhg al problema de búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes.
2. Comparar los resultados de ACOhg con los algoritmos del estado del arte en model checking.

Conclusiones:

1. ACOhg es capaz de encontrar trazas de error cortas con una pequeña cantidad de memoria.
2. ACOhg tiene un comportamiento similar en todos los modelos usados en los experimentos mientras que los algoritmos del estado del arte tienen un comportamiento desigual en los distintos modelos.
3. ACOhg es capaz de encontrar errores en modelos para los que la mayoría de los algoritmos del estado del arte fallan.

Relación con entregables:

DIRICOM

Diseño Inteligente de Redes Inalámbricas de Comunicación
Conserjería de Innovación, Ciencia y Empresa
P07-TIC-03044
http://diricom.lcc.uma.es

Málaga, November 22nd, 2008

Executive Summary

TITLE: SOFTW-1.0-2008: Ant Colony Optimization Applied to Model Checking

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. In this deliverable, we present the application of a new variant of Ant Colony Optimization (ACO), 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.

GOALS:

1. Apply ACOhg to the search of safety property violations in concurrent systems.
2. Compare the results of ACOhg against the state-of-the-art algorithms in model checking.

CONCLUSIONS:

1. ACOhg is able to get short error trails (good quality solutions) with a low amount of memory.
2. ACOhg has a similar behaviour for all the models used in the experimentation while the exact state-of-the-art algorithms have an erratic behaviour in the models.
3. ACOhg is able to find errors in models for which most of the state-of-the-art algorithms fail.

RELATION WITH DELIVERABLES:

Ant Colony Optimization Applied to Model Checking

DIRICOM

November 2008

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 [16], 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. Java PathFinder (JPF) is another model checker that works on Java bytecodes. Out of the box, JPF can only check deadlocks and unhandled exceptions. However, it is possible to extend the model checker by specifying user-defined properties.

The amount of states to explore in model checking 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 [18] and symmetry reduction [17]. 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 [4] 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 present the application of a new model of Ant Colony Optimization (ACO) [7] for finding safety errors in concurrent systems. ACO algorithms belong to the metaheuristic class of algorithms [3], 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.

The document 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. In Section 2 the ACO model used for the experiments, ACOhg, is described. In Section 4 we present some experimental results comparing the ACOhg-based algorithms against traditional exact algorithms for explicit state model checking. Finally, Section 5 outlines the conclusions of this deliverable.

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) [14]. 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. On the contrary, most of the canonical metaheuristic algorithms [3], 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 [19]. Safety properties of a model can be checked by searching for a single accepting state in the state graph. 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.

Genetic Algorithms (GAs) have also been applied to the problem of refuting safety properties in concurrent systems. In an early proposal, Alba and Troya [1] 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 order to guide the search, a heuristic value is associated to every automaton state. The computation of this value can be based on the property to verify [9] or in the objective node (if known beforehand) [17]. 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. 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}.

3 ACOhg

ACOhg (Ant Colony Optimization for Huge Graphs) is a new kind of ACO variant 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.

We use a node-based pheromone model, that is, the pheromone trails are associated to the nodes instead of the arcs. The algorithm works as follows. At the beginning, the variables are initialized (lines 1-5). All the pheromone trails are initialized with the same value: a random number between 0.1 and 10. In the `init` set (of initial nodes for the ants construction), a starting path with only the initial node is inserted (line 1). This way, all the ants of the first stage begin the construction of their path at the initial node.

After the initialization, the algorithm enters in a loop that is executed until a given maximum number of steps 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 construction of

Algorithm 1 ACOhg

```
1: init = {initial_node};
2: next_init = ∅;
3: τ = initializePheromone();
4: step = 1;
5: stage = 1;
6: while step ≤ msteps do
7:    for k=1 to colsize do {Ant operations}
8:       a^k = ∅;
9:       a^k_1 = selectInitNodeRandomly (init);
10:      while |a^k| < λ_ant ∧ T(a^k_*) − a^k ≠ ∅ ∧ a^k_* ∉ O do
11:         node = selectSuccessor (a^k_*, T(a^k_*), τ,η);
12:         a^k = a^k + node;
13:         τ = localPheromoneUpdate(τ,ξ,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:   τ = pheromoneEvaporation(τ, ρ);
21:   τ = pheromoneUpdate(τ, a^best);
22:   if step ≡ 0 mod σ_s then
23:      init = next_init;
24:      next_init = ∅;
25:      stage = stage+1;
26:      τ = pheromoneReset();
27:   end if
28:   step = step + 1;
29: end while
```

the path, the ants enter a loop (lines 10-14) in which each ant k stochastically selects the next node according to the traditional stochastic rule used in ACO [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 $\eta_j = 1/(1+H(j))$. This way, η_j increases when $H(j)$ decreases (short distance to the objective node). After the movement of an ant from a node to the next one the pheromone trail associated to the new node is updated as in Ant Colony Systems (ACS) using the expression $\tau_j \leftarrow (1-\xi)\tau_j$ (line 13). 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 (this is ensured by `selectBestPaths()`). A path a^k is inserted in this set if its last node is not one of the last nodes of a starting path π already included in the set. If this does not hold, the new path a^k replaces the starting path π of `next_init` only if $f(a^k) < f(\pi)$, where f is the objective function to be minimized that we present below. Before the inclusion, the path must be concatenated with the corresponding starting path of `init`, that is, the starting path π with $\pi_* = a^k_1$ (this path exists and it is unique). This way, only starting paths are stored in the `next_init` set. The cardinality of `next_init` is bounded by a given parameter ι. When this limit is reached and a new path must be included in the set, the starting path with higher objective value is removed from the set.

When all the ants have built their paths, a pheromone update phase is performed. First, all the pheromone trails are reduced (evaporation) according to the expression $\tau_j \leftarrow (1-\rho)\tau_j$ (line 20). Then, the pheromone trails associated to the nodes traversed by the best-so-far ant (a^{best}) are increased (line 21) using the expression $\tau_j \leftarrow \tau_j + 1/f(a^{best})$, $\forall j \in a^{best}$. We use here the mechanism introduced in Max-Min Ant Systems ($\mathcal{MM}$AS) for keeping the value of pheromone trails in a given interval $[\tau_{min}, \tau_{max}]$ in order to maintain the probability of selecting one node above a given threshold. The values of the trail limits are $\tau_{max} = 1/\rho f(a^{best})$ and $\tau_{min} = \tau_{max}/a$ where the parameter a controls the size of the interval.

Finally, with a frequency of σ_s steps, a new stage starts. The `init` set is replaced by `next_init` and all the pheromone trails are removed from memory (lines 22-27).

The objective function f to be minimized is defined as follows

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

where π is the starting path in `init` whose last node is the first one of a^k, p_p, and p_c are penalty values that are added when the ant does not end in an objective node and when a^k contains a cycle, respectively. The last term in the second row of Eq. (1) makes the penalty higher in shorter cycles (see [5] for more details).

4 Experimental Section

In this section we present some results obtained with the ACOhg algorithms. For the experiments we have selected five Promela models previously reported in the literature by Edelkamp et al. [10]. All these models violate a safety property. The models are `giop22`, `marriers4`, `needham` , `phi16` and `pots`. For a brief description of the models see [2].

The configuration parameters of ACOhg are shown in Table 1. 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.

Table 1: 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. However, we use two versions of ACOhg: one not using heuristic information and another one using it. The stopping criterion used in our ACOhg algorithms is to find an error trail or to reach a maximum number of allowed steps (100). We performed 100 independent runs in order to get high statistical confidence on the results. The machine used in the experiments is a Pentium 4 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.

In the following experiments we compare the results of ACOhg against exact algorithms previously found in the literature. These algorithms are BFS, DFS, A*, and BF. BFS and DFS do not use heuristic information while the other two do. In order to make a fair comparison we use two different ACOhg algorithms: one not using heuristic information (ACOhg-b) and another one using it (ACOhg-h). We compare ACOhg-b against BFS and DFS in Table 2, and ACOhg-h against A* and BF in Table 3. 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.

The first observation that we can make from the results of Table 2 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

Table 2: 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

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 3: 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 3. 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 3 and 2) 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 3 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 3. 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.

5 Conclusions

We have presented here the application of a new Ant Colony Optimization variant, 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. ACOhg algorithms are able to get short error trails (good quality solutions) with a low amount of memory. They are more robust than the state-of-the-art algorithms used in our experimental section. That is, ACOhg algorithms have a similar behaviour for all the models, while the behaviour of the exact state-of-the-art algorithms depends to a large extent on the model they solve.

References

[1] E. Alba and J.M. Troya. Genetic Algorithms for Protocol Validation. In *Proceedings of the PPSN IV International Conference*, pages 870–879, Berlin, 1996. Springer.

[2] Enrique Alba and Francisco Chicano. Finding safety errors with ACO. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, pages 1066–1073, London, UK, July 2007. ACM Press.

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

[4] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4), April 1994.

[5] Francisco Chicano and Enrique Alba. Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models. *Information Processing Letters*, 106(6):221–231, June 2008.

[6] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000.

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

[8] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed Explicit Model Checking with HSF-SPIN. In *Lecture Notes in Computer Science, 2057*, pages 57–79. Springer, 2001.

[9] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Protocol Verification with Heuristic Search. In *AAAI-Spring Symposium on Model-based Validation Intelligence*, pages 75–83, 2001.

[10] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal of Software Tools for Technology Transfer*, 5:247–267, 2004.

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

[12] Patrice Godefroid and Sarfraz Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. In *Lecture Notes in Computer Science, 2280*, pages 266–280. Springer, 2002.

[13] Patrice Godefroid and Sarfraz 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, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32. American Mathematical Society, 1996.

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

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

[17] Alberto Lluch Lafuente. Symmetry Reduction and Heuristic Search for Error Detection in Model Checking. In *Workshop on Model Checking and Artificial Intelligence*, August 2003.

[18] Alberto Lluch-Lafuente, Stefan Leue, and Stefan Edelkamp. Partial Order Reduction in Directed Model Checking. In *9th International SPIN Workshop on Model Checking Software*, Grenoble, April 2002. Springer.

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