2011 35th IEEE Annual Computer Software and Applications Conference

Ant Colony Optimization for Deadlock Detection in Concurrent Systems

Gianpiero Francesca, Antonella Santone
Dip. di Ingegneria
University of Sannio
Benevento, Italy
Email: {gianpiero.francesca@gmail.com, santone@unisannio.it}

Gigliola Vaglini
Dip. di Ing. dell'Informazione
University of Pisa
Pisa, Italy
Email: g.vaglini@iet.unipi.it

Maria Luisa Villani
ENEA
Roma, Italy
Email: marialuisa.villani@enea.it

***Abstract*—Ensuring deadlock freedom is one of the most critical requirements in the design and validation of concurrent systems. The biggest challenge toward the development of effective deadlock detection schemes remains the state-space explosion problem when model checking is used for proving the correctness of a system with respect to a desired behavior. In this paper we propose the use of the Ant Colony Optimization (ACO) to reduce the state explosion problem arising when finding deadlocks in complex networks described using Calculus of Communicating Systems (CCS). Moreover, ACO is used to provide minimal counterexamples. In fact, although one of the strongest advantages of model checking is the generation of counterexamples when verification fails, traditional model checkers may return very long counterexamples. We present an implementation of our technique and encouraging experimental results on several benchmarks. These results are then compared with other heuristic-based search strategies, retaining the advantages of our approach.**

***Keywords*-Formal methods; CCS; Ant Colony Optimization; Heuristic Searches.**

I. Introduction

Ensuring deadlock freedom is one of the most critical requirements in the design and validation of concurrent systems. Model checking [14] is an automatic technique to verify compliance of the system implementation with respect to its defined requirements. It applies to a formal description of the system behavior as a finite automata, and to a temporal-logic formula representing the requirement to be verified. Examples are, respectively, the Calculus of Communicating Systems (CCS) [30], which provides a Labelled Transition System (LTS) semantics of processes and is especially powerful to describe concurrency, complex networks and interaction networks, and the mu-calculus logic [33].

Model checking is also used to verify complex systems, i.e. systems composed of interconnected parts that as a whole exhibit one or more properties not obvious from the properties of the individual parts. Model checking has a number of advantages over traditional approaches that are based on simulation and testing and deductive reasoning. The biggest challenge toward the development of effective deadlock detection schemes remains the state explosion problem when model checking is used for proving the correctness of a system with respect to a desired behavior. This problem occurs in systems with many components that can interact with each other or in systems with data structures that can assume many different values.

Numerous approaches have been introduced for alleviating state-space explosion, for example symbolic model checking [29], on-the-fly [28], compositional reasoning [12], [32] local model checking [34], partial order [21], and abstraction [13]. Although the size of the systems that could be verified has been increased by these techniques, many realistic systems are still too large to be handled.

In recent years, great interest has been shown in combining techniques from the artificial intelligence area with model checking techniques, and this yields another promising future direction of research in the model checking field.

In this paper we consider concurrent systems described using the Calculus of Communicating Systems (CCS) [30] and propose the use of the Ant Colony Optimization (ACO) to reduce the state explosion problem. ACO algorithms are stochastic techniques belonging to the class of metaheuristic algorithms and inspired by the foraging behavior of real ants. Although one of the strongest advantages of model checking is the generation of counterexamples when verification fails, traditional model checkers (based on depth-first search algorithms) may return very long counterexamples. Ant Colony Optimization, beside reducing the state explosion problem, can be used to find short counterexamples. In fact, ACO algorithms are able to find optimal or near optimal solutions using a reasonable amount of resources. For this reason, they can be suitable for searching states (in particular deadlocks) in the graph of large system models, for which traditional exploration algorithms fail. A short counterexample is preferred to a long one, when searching deadlocks, since that path will be examined in order to determine the cause of the deadlock, and long error paths can prevent an easy comprehension of the fault.

A tool implementing our technique has been developed and used to verify a sample of CCS processes: the results of these

0730-3157/11 $26.00 © 2011 IEEE
DOI 10.1109/COMPSAC.2011.22

experiments are discussed in Section IV and comparisons with other informed search strategies are given. We show that the technique presented can indeed improve the search, especially when combined with a heuristic, in terms of nodes generation and solution length.

II. Overview of CCS

Let us now quickly recall the main concepts about the Calculus of Communicating Systems (CCS) [30]. The syntax of *processes* is the following:

$$p ::= nil \mid x \mid \alpha.p \mid p+p \mid \ p|p \ \mid p\backslash L \mid p[f]$$

where α ranges over a finite set of actions $\mathcal{A} = \{\tau, a, \overline{a}, b, \overline{b}, ...\}$. The action $\tau \in \mathcal{A}$ is called the *internal action*. The set of *visible actions*, $\mathcal{V}$, ranged over by $l, l' \ldots$, is defined as $\mathcal{A} - \{\tau\}$. The set L, in the processes of the form $p\backslash L$, is a set of actions such that $L \subseteq \mathcal{V}$; while the relabeling function f, in the processes of the form $p[f]$, is a total function, $f : \mathcal{A} \to \mathcal{A}$, such that the constraint $f(\tau) = \tau$ is respected. Each action $l \in \mathcal{V}$ (resp. $\overline{l} \in \mathcal{V}$) has a *complementary action* $\overline{l}$ (resp. l). Given $L \subseteq \mathcal{V}$, with $\overline{L}$ we denote the set $\{\overline{l}, l \mid l \in L\}$. The constant x ranges over a set of *constant* names: each constant x is defined by a constant definition $x \stackrel{\text{def}}{=} p$, where p is called the *body* of x. We denote the set of processes by $\mathcal{E}$.

The process nil can perform no actions. The process $\alpha.p$ can perform the action α and then become the process p. The process $p+q$ can behave either as p or as q. The operator $|$ expresses parallel composition: if the process p can perform α and become p', then $p|q$ can perform α and become $p'|q$, and similarly for q. Furthermore, if p can perform a visible action l and become p', and q can perform $\overline{l}$ and become q', then $p|q$ can perform τ and become $p'|q'$. The operator $\backslash$ expresses the restriction of actions. If p can perform α and become p', then $p\backslash L$ can perform α to become $p'\backslash L$ only if $\alpha \notin \overline{L}$. The operator $[f]$ expresses the relabeling of actions. If p can perform α and become p', then $p[f]$ can perform $f(\alpha)$ and become $p'[f]$. Each relabeling function f has the property that $f(\tau) = \tau$. Finally, a constant x behaves as p if $x \stackrel{\text{def}}{=} p$.

Given a set $\mathcal{D}$ of constant definitions, the standard *operational semantics* [30] is given by a relation $\longrightarrow_{\mathcal{D}} \subseteq \mathcal{E} \times \mathcal{A} \times \mathcal{E}$. $\longrightarrow_{\mathcal{D}}$ ($\longrightarrow$ for short) is the least relation defined by the rules in Table I.

A *(labeled) transition system* is a quadruple $T = (S, \mathcal{A}, \longrightarrow, p)$, where S is a set of states, $\mathcal{A}$ is a set of transition labels (actions), $p \in S$ is the initial state, and $\longrightarrow \subseteq S \times \mathcal{A} \times S$ is the transition relation. If $(p, \alpha, q) \in \longrightarrow$, we write $p \stackrel{\alpha}{\longrightarrow} q$. If $\delta \in \mathcal{A}^*$ and $\delta = \alpha_1 \ldots \alpha_n, n \geq 1$, we write $p \stackrel{\delta}{\longrightarrow} q$ to mean $p \stackrel{\alpha_1}{\longrightarrow} \cdots \stackrel{\alpha_n}{\longrightarrow} q$. Moreover $p \stackrel{\lambda}{\longrightarrow} p$, where λ is the empty sequence. Given $p \in S$, with $\mathcal{R}_{\longrightarrow}(p) = \{q \mid p \stackrel{\delta}{\longrightarrow} q\}$ we denote the set of the states reachable from p by $\longrightarrow$. Given a CCS process p, the *standard transition system* for p is defined as $\mathcal{S}(p) = (\mathcal{R}_{\longrightarrow}(p), \mathcal{A}, \longrightarrow, p)$. In the following, $p \not\longrightarrow$ denotes that no p', α exist such that $p \stackrel{\alpha}{\longrightarrow} p'$,while $p \stackrel{\alpha}{\not\longrightarrow}$ denotes that no p' exists such that $p \stackrel{\alpha}{\longrightarrow} p'$. Now the notion of deadlock is defined.

Definition 2.1: Let p be a CCS process.

- p is $deadlocked$ if $p \not\longrightarrow$;
- p is deadlock sensitive if and only if $q \in \mathcal{R}_{\longrightarrow}(p)$ exists such that $q \not\longrightarrow$;
- p is deadlock free if and only if it is not deadlock sensitive.

Note that p is deadlock sensitive if and only if $\mathcal{S}(p)$ contains at least a deadlocked state corresponding to a deadlocked process.

The difficulty of building complex systems has sparked a large research effort to develop formal approaches to the design and analysis of these systems. To reason formally about real-world systems, tool support is necessary; consequently, a number of tools embodying various analysis methods have been developed. They provide a support for automatically answering the verification question: does a system sys satisfy a property φ? To implement such a tool, the verification question must be formulated more carefully by fixing the following:

1) a precise notation for defining systems;
2) a precise notation for defining properties; and
3) an algorithm to check if a system satisfies a property.

To cope with the first problem, several specification languages have been developed as, for example, CCS. The second problem can be solved using a temporal logic as, for example, the mu-calculus logic [33]. For the last problem, several algorithms exist. The most used verification methodology is model checking [14]: the property that the system must satisfy, expressed in some temporal logic, is checked on a finite structure representing the behavior of the system. If we specify the system by means of a process algebra program, like CCS, this structure is a labeled transition system, i.e., an automaton whose transitions are labeled by event names. The transition system represents all possible executions of the program. To check a property, model checking explores every possible state that the system may reach during execution. If the system does not satisfy the property, a description of the execution sequence leading to the state is reported to the user that can be used to pinpoint the source of the error. Many bugs such as deadlock and critical section violations may be found using this approach.

One of the most popular verification environment is the Concurrency Workbench of New Century (CWB-NC) [15], which includes several different specification languages, among which CCS. In the CWB-NC the verification of temporal logic formulae is based on model checking.

Act	$\dfrac{}{\alpha.p \xrightarrow{\alpha} p}$		**Sum**	$\dfrac{p \xrightarrow{\alpha} p'}{p+q \xrightarrow{\alpha} p'}$ *and symmetric*	
Con	$\dfrac{p \xrightarrow{\alpha} p'}{x \xrightarrow{\alpha} p'}\ x \stackrel{\text{def}}{=} p \in \mathcal{D}$		**Par**	$\dfrac{p \xrightarrow{\alpha} p'}{p\|q \xrightarrow{\alpha} p'\|q}$ *and symmetric*	
Com	$\dfrac{p \xrightarrow{l} p',\ q \xrightarrow{\bar{l}} q'}{p\|q \xrightarrow{\tau} p'\|q'}$	**Rel**	$\dfrac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}$	**Res**	$\dfrac{p \xrightarrow{\alpha} p'}{p\backslash L \xrightarrow{\alpha} p'\backslash L}\ \alpha \notin \overline{L}$

Table I
STANDARD OPERATIONAL SEMANTICS OF CCS

III. HEURISTIC SEARCHES AND ACO

Blind search algorithms, such as Breadth-First (BFS) and depth-first (DFS) searches, are simple and can in principle find solutions to any state space problem. However, they usually result inefficient and impractical in case of large search spaces, as those of most real problems. On the other hand, heuristic search algorithms can be applied to improve the search efficiency.

In this section we briefly recall some notions on heuristic search algorithms, in particular A*. The reader can refer to [31] for further details.

A. The A* algorithm.

A well-known heuristic search algorithm for solving state-space search problems is A* [31]. A* is an algorithm for finding a path in a graph that leads to a goal node.

In addition to nodes, arcs and costs, A* uses one more kind of data: a number $\widehat{h}(n)$, that is a heuristic evaluation function, associated with each node. Specifically, $\widehat{h}(n)$ is an estimate of a lower bound on the cost of getting from that node to a goal node. The A* algorithm prefers to visit nodes that appear to be better, i.e., nodes with the minimum evaluation, so that goal nodes are found faster. More precisely, at each step A* generates and evaluates the successors of an unexplored node n with the lowest total estimate, $f(n) = g(n)+\widehat{h}(n)$, that is the sum of the distance from the start node to the node n, namely $g(n)$, plus the estimate from the node n to the goal node, i.e., $\widehat{h}(n)$. It stops when a goal node is chosen. A* starts with the initial node and generates all its children nodes. The estimates are usually based on logical or physical knowledge, which otherwise would not be represented in the graph. If the nodes represent cities and the arc costs are railroad miles, then $\widehat{h}(n)$ might be the airline distance from the city n to the goal city; if the nodes are puzzle positions, $\widehat{h}(n)$ might be the minimum number of moves before the puzzle is possibly solved. Table II outlines the A* algorithm for finding the minimal-cost solution path in a graph. For a more precise description of the algorithm the reader can refer to [31].

An important property holds: A* returns a minimal-cost solution path provided that the heuristic estimate function

1) Let $g(n)$ be the cost of a path from $start$ to node n and get $g(start) = 0$. Let OPEN be a list of nodes that initially contains only the $start$ node. Calculate the estimate $\widehat{h}(start)$.
2) Select the node n on the list OPEN such that the quantity $f(n) = (g(n)+\widehat{h}(n))$ is the smallest. In the presence of two or more nodes with the same $f(n)$, just select any of them. If n is a goal node, then the path to n is a preferred solution path and its cost is $g(n)$. If there are no OPEN nodes, there is no solution path in the graph.
3) Remove n from OPEN. Find all the successors of n. For each successor s, let $g(s) = g(n)+(\text{cost on arc form } n \text{ to } s)$. Calculate the estimate $\widehat{h}(s)$ and add s to OPEN if $\widehat{h}(s)$ is not ∞.
4) Go to step 2.

Table II
THE A* ALGORITHM.

$\widehat{h}$ satisfies the so-called *admissibility* condition, i.e., $\widehat{h}$ is optimistic, and f is a non-decreasing function. More formally:

Definition 3.1 (admissibility): A heuristic estimate function $\widehat{h}$ defined on the nodes of a graph G is admissible if for each node n in G,

$$\widehat{h}(n) \leq h(n)$$

where $h(n)$ is the actual cost of a preferred path from n to a goal node.

B. Ant Colony Optimization

Ant Colony Optimization algorithms were first introduced by Dorigo [18] and are a class of probabilistic techniques for solving combinatorial optimization problems which can be reduced to finding cost efficient paths on graphs.

As the name suggests, ACO algorithms have been inspired by the behavior of real ants. The idea behind is that, although the real ants are blind, they can construct the shortest paths from their nest to the food sources. The colonies accomplish this task by using a collective decision making strategy based on pheromone trails. Each ant is free to choose its path but it can also follow a pheromone trail left by other ants. During their walks, the ants leave pheromone trails on the paths, enlarging the probability of their choice. Thanks to this process, the shorter paths will be more desirable because the ants will walk through them faster, increasing their

pheromone trails sooner. This behavior has been successfully applied to several NP-complete optimization problems.

In ACO, there is a set of artificial ants (colony) that build the solutions using a stochastic constructive procedure. The main idea is to use an ant colony that explores the model graph of the problem, searching for solutions. When a solution is found, the pheromone process allows to find the shortest path from the source to the solution node. To help the algorithm's convergence, there are also two processes that do not appear in the natural ant system: daemons and pheromone evaporation. Daemons can control the search by erasing or increasing pheromone trails. The pheromone evaporation uniformly reduces the pheromone values in order to advantage the arcs that belong to the solution paths.

In Table III a general ACO pseudo-code is described and Figure 1 presents a flow chart describing how the ACO works to solve problems specifically.

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

Table III
PSEUDO-CODE OF THE ACO METAHEURISTIC

More formally, ants walk randomly on a graph $G = (C, L)$ called construction graph, where L is the set of connections (arcs) among the components (nodes) of C. Each connection $l_{ij} \in L$ has an associated pheromone trail τ_{ij}; each node $k \in C$ has an associated numerical value h_k, which by default is equal to one but can be replaced by an heuristic value. Both values are used to guide the search. In first step the algorithm places the ants on the source node, then at each step the ants go independently from a node to another looking for a solution; that behavior is the forward mode. When an ant finds a solution, it switches in backward mode: it goes back from the solution node to the source leaving the pheromone on the arcs. When it reaches the source node it switches in forward mode and restarts. The algorithm runs until a given stopping criterion is fulfilled, such as finding a solution or reaching a given number of steps. In the forward mode, the ant probabilistically chooses the next node according to the formula:

$$p_{i,j}^k = \begin{cases} \frac{(\tau_{i,j}^{\alpha}) \cdot (\frac{1}{h_j})^{\beta}}{\sum_{l \in N_i^k} (\tau_{i,l}^{\alpha}) \cdot (\frac{1}{h_j})^{\beta}} & \text{if } j \in N_i^k \\ 0 & \text{if } j \notin N_i^k \end{cases}$$

where N_i^k is the set of the nodes next to i, and the α and β are algorithm's parameters.

An important parameter is the pheromone quantity that the ants can leave on the edges. In most cases, this can be a constant quantity or be proportional to the solution goodness.

IV. THE METHOD AND EXPERIMENTAL RESULTS

In this section we discuss our experience of applying ACO to verify the deadlock freedom property in CCS processes. To this aim, we considered an heuristic function proposed in [22], whose formal definition, for completeness, is reported in the Appendix. The role of the heuristic function is to guide the ants in finding deadlocks and to guide the construction of a minimal-cost solution path. A tool implementing ACO with and without use of the heuristic function has been developed. We have used this tool to verify a sample of CCS processes, in order to evaluate the performances of our method, and compare them against some informed/non-informed strategies, such as BFS, A*, and the algorithm used by the CWB-NC.

The ACO algorithm was run with the following parameters settings

- h is the heuristic value (see Appendix);
- $\alpha = 1$; and $\beta = 1$;
- the initial value of the pheromone is equal to 1000;
- the amount of pheromone deposited is $1/length(S)$, where S is the set of arcs of the path from the source to a deadlock and circular paths are deleted;
- evaporation= 0.99999;
- the default number of ants is 10.

Also, 100 independent runs were executed on the same CCS process to get statistically significant values.

First we considered several instances of the dining philosophers example, as a well-known incorrect solution of that problem is described by a deadlock sensitive CCS process. In this solution, when a philosopher gets hungry, he can, without any control, pick up his left fork first, then his right one; if he can eat, then he puts the forks down in the same order that he had picked them up.

On these processes, the ACO algorithm was run both with and without the heuristic function in order to analyze its impact on the performance of the algorithm. Indeed, we discovered that the use of the heuristic is really worthwhile for big-size processes, such as the instances with $n > 8$ philosophers.

Figure 2 shows that, over the same number of iterations allowed in both cases, the heuristic much helps to reduce the exploration area, and Figure 3 provides evidence that the heuristic also helps to reach minimal length solutions.

Instead, Table IV shows a comparison of the heuristic-based ACO and the other strategies, on the number of generated nodes. The ACO algorithm used in those experiments had a colony size of 15 ants and performed 1000 iterations. It

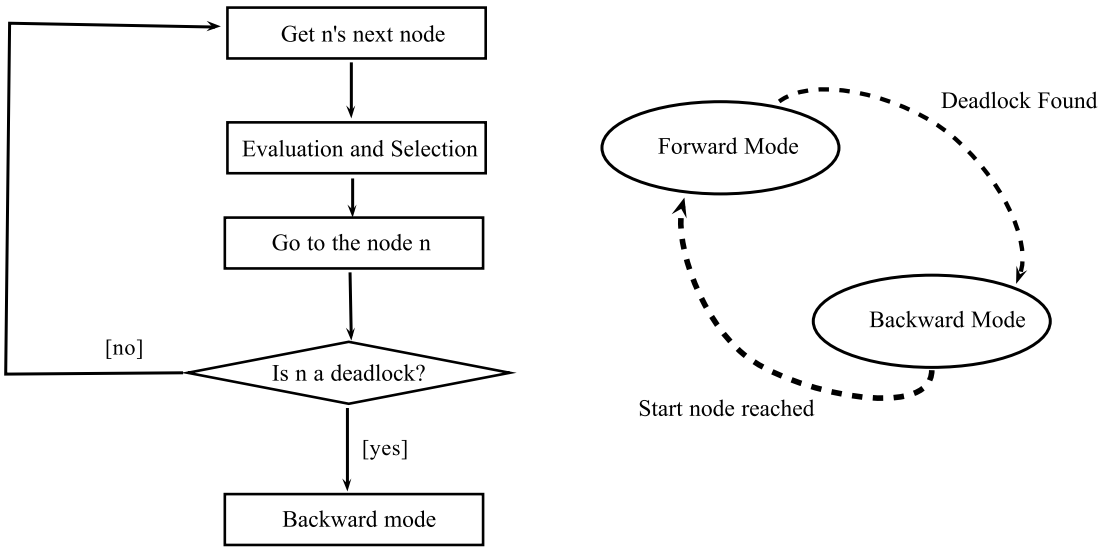


Figure 1. Flow chart of ACO

is interesting to note that the CWB-NC (resp. BFS) was not able to build the complete transition system with $n = 8$ (resp. $n = 9$); A* succeeds until $n = 12$, while only ACO exceed those limits.

n	ACO	A*	BFS	CWB-NC
2	10	7	12	18
3	42	16	48	79
4	96	33	185	342
5	184	81	741	1474
6	989	250	2964	6345
7	1561	1036	11952	27304
8	7749	2501	48376	–
9	12596	2862	–	–
10	19071	4731	–	–
11	40245	11287	–	–
12	45276	42131	–	–
13	48546	–	–	–
14	55446	–	–	–
15	59667	–	–	–

Table IV
RESULTS OF FOR THE DINING PHILOSOPHERS (GENERATED NODES)

In Figure 4 we have reported the length of the counterexamples for the dining philosophers when ACO and A* are used.

As Figure 4 reports, A* succeeds until 7 philosophers, while ACO succeeds until 12 philosophers.

We have also used the greedy strategy in our experiments noting that ACO returns better results than greedy mostly in finding the (near) minimal length path. Our aim is to find short counterexamples and to do so quickly. These two goals are in contrast so we need a trade-off between the advantage of the shortest counterexample and the cost to compute it. Thus, sometimes the optimality of the solution is not worth the loss in efficiency of the search. Therefore, the efficiency could be further improved because the number of generated nodes can be reduced by stopping the run when a solution is found. Table V shows the number of generated nodes in this case.

n		ACO	A*	BFS	CWB-NC
7	gen	416	1036	11952	27304
	$time$	2657	19531	619953	30937
9	gen	676	2862	–	–
	$time$	7766	97984	–	–
11	gen	1120	11287	–	–
	$time$	23222	774609	–	–
13	gen	2161	–	–	–
	$time$	75806	–	–	–

Table V
RESULTS OF FOR THE DINING PHILOSOPHERS (STOPPING WHEN A SOLUTION IS FOUND)

Improvements to the above method are introduced in order to ensure the correctness of deadlock-free concurrent systems. We use three kinds of daemon actions:

- *useless node elimination*: our heuristic is syntactically defined, thus can return infinity when a constant already expanded is encountered: this means that the state under consideration is safe. When a node, with heuristic value equal to infinity, is reached, the daemon deletes the pheromone trail.

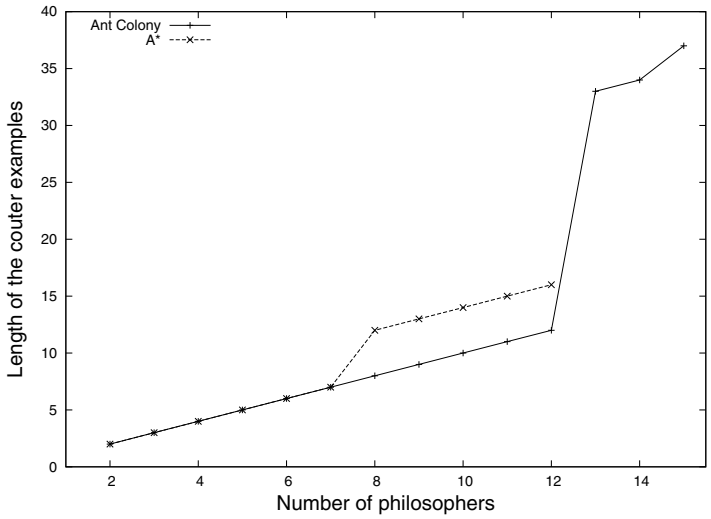


Figure 4. Length of the counterexamples

- *loop elimination*: whenever a loop is generated, the daemon deletes the pheromone trail breaking the loop.
- *dead-end node*: whenever an ant finds a node such that all the outgoing edges have no pheromone trails, the daemon erases ingoing edge's pheromone trail.

These improvements help to find a deadlock, but are especially useful to ensure the correctness of a deadlock-free system. In fact, a deadlock-free system is modelled with an automaton where no sink node exists. Using the actions of the daemons, eventually ants will stick in the start node proving the deadlock freedom of the system.

For a more complete evaluation of applying ACO to deadlock detection, we selected from the literature a sample of well known deadlock sensitive systems[1]:

- Solitaire Game (SG) [1]: a CCS specification of a solitaire game, developed by Luca Aceto.
- Context Management Application Service Element (CM-ASE) [16]: a model of the Application Layer of the Aeronautical Telecommunications Network, developed by Gurov and Kapron.
- Mail System (MS) [10]: a specification of a mail system, devised by Gordon Brebner. The communication software is a multiprocess implementation where one process handles each communication protocol used in the system.
- GRID [9]: two processes on a grid 5×5 of relay stations which allow them to communicate.
- MUTUAL: a system handling the requests of a resource shared by 10 processes. It presents two alternative choices between a server based on a *round robin scheduling* and a server based on *mutual exclusion*.

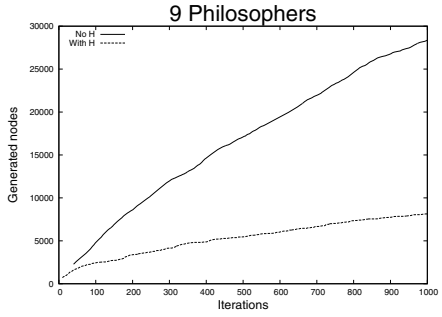
[1]Deadlock is present by design in some processes, while in others it has been induced.

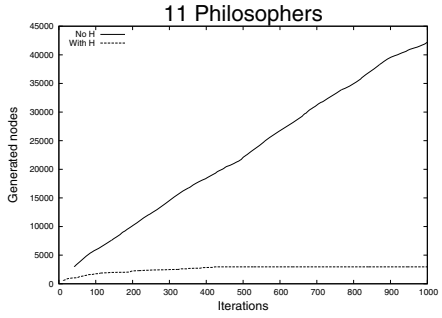
- Philips Bounded Retransmission Protocol (BRP) [24], [25], [26]: the Bounded Retransmission Protocol used by the Philips Company in one of its products.
- Solid State Interlocking (SSI) [11]: this system describes the British Rail's Solid State Interlocking (SSI) which is devoted "to adjust, at the request of the signal operator, the setting of signal and points in the railway to permit the safe passage of trains."

The results of all runs are reported in Table VI: *time* is in msec, *gen* indicates the number of generated states and *length* represents the best path length leading to a deadlock. Experiments were executed under windows on an Intel Pentium dual CPU with a 2 GHz processor and 2GB of RAM. In all the experiments we set the number of the ants to 10 and we performed 1000 iterations. In Table VI we compare only ACO, A* and BFS; clearly we have not considered the CWB-NC, since it uses a DFS search. It is worth to note that our approach really improves the search in terms of nodes generation and the time employed. Only in a few cases we loose the best path.

V. Conclusion and Related Work

In the literature, the use of ACO is limited to finding near-minimal counterexamples. In this paper, ACO is also used to reduce the state space explosion problem, when finding deadlocks in concurrent systems described with CCS. This method can also be applied to programming languages, with the only constraint to transform the program in a CCS process or, equivalently, in a transition system. For example, in [23] we have shown an example of transformation for multi-threaded JAVA program. Moreover, for CCS, several methodologies to reduce the state explosion problem have been defined, based on compositional reasoning [17], partial order [20], abstraction [8]. All these methods can be





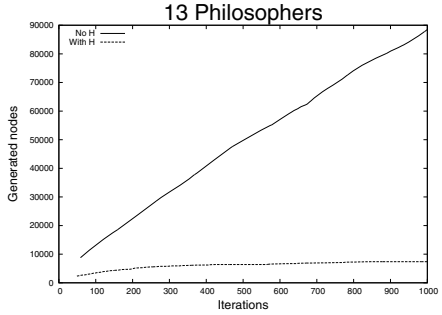
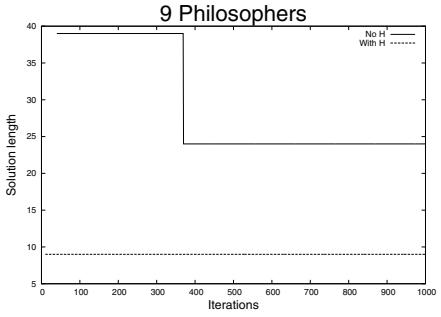
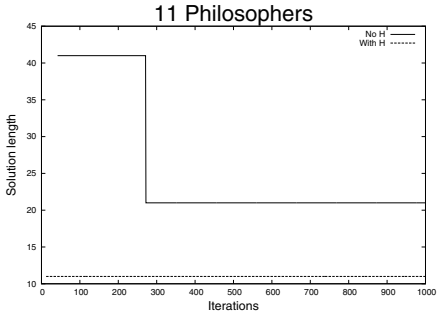


Figure 2. Generated nodes per iteration





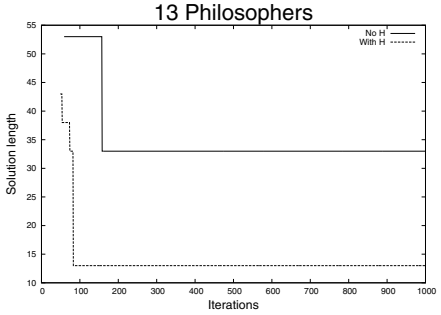


Figure 3. Length of the best solution found at each iteration

combined with our ACO-based method.

As a future work we intend to perform an empirical analysis in order to investigate the effectiveness of our approach in relation with the structure of CCS processes. With this kind of investigation we could be able to define a sort of taxonomy that would be of great value when the search strategy more appropriate for the problem at hand has to be chosen.

The application of ACO to formal verification has not yet been extensively explored. Only Alba and Chicano [2], [3], [4] have proposed ACO for formal verification in the SPIN [27] environment. That approach is applied in [2] to any safety property and works on the product automaton between the formula negation and the system automaton; our approach is dedicated to a particular safety property and is applied on the system automaton directly. Moreover, they use the heuristic functions defined by Edelkamp et al. in [19], namely, two estimation functions, the first counts the number of active (or non-blocked) processes, and the other is a formula based heuristic where the deadlock formula is inferred from the user designated dangerous states. We use an estimation that is made from the structure of the process, hence more informative. The technique proposed in [2] is not able to ensure the correctness when no error (for example a deadlocked state) is found since they establish a maximum length for the paths that the ants build. Our work overcomes this problem since a syntactically defined heuristic and particular daemon actions are used.

According to the experiments carried out, optimal counterexamples have been retrieved in most cases in our approach. Nevertheless, we can observe that we have, in percentage, better results both in terms of memory reduction and in time of CPU, compared with techniques such as BFS and A*.

From the point of view of the length of counterexample instead, in some cases we do not reach the minimal counterexample path, but the result is quite close to it.

		ACO	A*	BFS
SG	gen	53	666	111
	$time$	515	1969	1906
	$length$	7	7	7
CM-ASE	gen	47	82	91
	$time$	1735	5735	19594
	$length$	6	6	6
MS	gen	63	265	276
	$time$	813	9828	22656
	$length$	12	12	12
MPMC	gen	423	3517	3361
	$time$	6766	308937	1452984
	$length$	20	18	18
GRID	gen	563	2548	–
	$time$	16299	315891	–
	$length$	11	6	6
MUTUAL	gen	670	6656	17221
	$time$	3985	2515003	2598094
	$length$	14	14	14
BRP	gen	449	101	99
	$time$	23050	14107	11351
	$length$	22	21	21
SSI	gen	124	101	99
	$time$	688	4328	5358
	$length$	6	6	6

Table VI
RESULTS FOR SOME DEADLOCKED SYSTEMS

A more accurate comparison with the results of the Alba and Chicano's approach cannot be made, since they start from a Promela description of the concurrent system under verification and speak of memory occupation; instead, we start from the transition system corresponding to a CCS process and speak of system states.

REFERENCES

[1] L. Aceto. The Solitaire. URL: http://www.cs.auc.dk/∼luca/DAT4/solitaire.cwb.

[2] E.Alba, F. Chicano. Ant Colony Optimization for Model Checking. In *Proc. of the 11th International Conference on Computer Aided Systems Theory (EUROCAST 2007)* Lecture Notes in Computer Science 4739, 2007. 523-530.

[3] E. Alba, F. Chicano. Finding Safety Errors with ACO. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO07)*, pp. 1066-1073, London, UK.

[4] E. Alba, F. Chicano. ACOhg: Dealing with Huge Graphs. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO07)*, pp. 10-17, London, UK.

[5] G. Anastasi, A. Bartoli, N. De Francesco. Efficient Verification of a Multicast Protocol for Mobile Computing. *The Computer Journal, 44(1)*, 2001. 21-30.

[6] G. Anastasi, F. Spadoni, A. Bartoli. Group Multicast in Distributed Mobile Systems with Unreliable Wireless Network. In *Proc. of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS '99)* IEEE Computer Society, 14, 1999. 14-23.

[7] A. Arnold, D. Begay, P. Crubille. Construction and analysis of transition systems with MEC. *World Scientific, Chapter 6*, 1994.

[8] R. Barbuti, N. De Francesco, A. Santone, G. Vaglini. Reduced Models for Efficient CCS Verification *Formal Methods in System Design, 26(3)*, 2005. 319-350.

[9] J. Bradfield, C. Stirling. Verifying Temporal Properties of Processes. In *Proc. of CONCUR '90: Theories of Concurrency - Unification and Extension* Springer, 1990. 115-125.

[10] G. J. Brebner. A CCS-based Investigation of Deadlock in a Multi-process Electronic Mail System. *Formal Aspect of Computing, 5(5)*, 1993. 467-478.

[11] G. Bruns. A Case Study in Safety-Critical Design. In *Proc. of the Fourth International Workshop on Computer Aided Verification* Springer, 1992. 220-233.

[12] E.M. Clarke, D.E. Long, K.L. McMillan. Compositional Model Checking. *Proc. of the Fourth Annual IEEE Symposium on Logic in Computer Science*, 1989. 353-362.

[13] E.M. Clarke, O. Grumberg, D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems, 16(5)*, 1994. 1512-1542.

[14] E.M. Clarke, O. Grumberg, D. Peled. Model Checking. *MIT press*, 2000.

[15] R. Cleaveland, S. Sims. The NCSU Concurrency Workbench. In *Proc. of the Eighth International Conference on Computer-Aided Verification (CAV'96)*, Lecture Notes in Computer Science 1102, 1996. 394–397.

[16] The Context Management Application Server Element. URL: http://www.cs.uvic.ca/∼bmkapron/ccs.html

[17] M. Dam, D. Gurov Compositional Verification of CCS Processes In *Proc. of the Third International Andrei Ershov Memorial Conference (PSI'99)*, Lecture Notes in Computer Science 1755, 2000. 247-256.

[18] M. Dorigo, T. Stuetzle. Ant Colony Optimization. *MIT Press*, 2004.

[19] S. Edelkamp, A. Lluch-Lafuente, S. Leue. Directed Explicit Model Checking with HSF-SPIN. In *Proc. of the 8th International SPIN Workshop on Model Checking Software*, Lecture Notes in Computer Science 2057, 2001. 57-79.

[20] R. Gerth, R. Kuiper, D. Peled, W. Penczek A partial order approach to branching time logic model checking. *Information and Computation, 150(2)*, 1999. 132-152.

[21] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems. Lecture Notes in Computer Science 1032, 1996.

[22] S. Gradara, A. Santone, M.L. Villani. DELFIN+: An Efficient Deadlock Detection Tool for CCS Processes. *Journal of Computer and System Sciences, 72(8)*, 2006. 1397-1412.

[23] S. Gradara, A. Santone, M.L.Villani. Using Heuristic Search for Finding Deadlocks in Concurrent Systems. *Information and Computation, 202(2)*, 2005. 191-226.

[24] J. F. Groote, J. van de Pol. A Bounded Retransmission Protocol for Large Data Packets. *Algebraic Methodology and Software Technology, 1996*. 536-550.

[25] K. Havelund, N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *Industrial Benefit and Advances in Formal Methods (FME '96)*, Springer-Verlag, 1996. 662-681.

[26] L. Helmink, M. P. A. Sellink, F. W. Vaandrager. Proof-checking a data link protocol. In *Proc. of the International Workshop on Types for Proofs and Programs (TYPES '93)*, Lecture Notes in Computer Science, 806, 1994. 127-165.

[27] G.J. Holzmann, Design and Validation of Computer Protocols. *Prentice Hall*, 1991.

[28] C. Jard, T. Jéron. Bounded-memory Algorithms for Verification on-the-fly. In *Proc. of the Third International Conference on Computer-Aided Verification (CAV'91)*, Lecture Notes in Computer Science 575, 1991. 192-201.

[29] K. McMillan. Symbolic Model Checking. *Kluwer Academic Publishers*, 1993.

[30] R. Milner. Communication and Concurrency. *Prentice-Hall*, 1989.

[31] J. Pearl. Heuristics: Intelligent Search Strategies for Computer Problem Solving. *Addison-Wesley*.

[32] A. Santone. Automatic Verification of Concurrent Systems using a Formula-Based Compositional Approach. *Acta Informatica, 38(2)*, 2002. 531-564.

[33] C. Stirling. An Introduction to Modal and Temporal Logics for CCS. In *Concurrency: Theory, Language, and Architecture*, Lecture Notes in Computer Science 391, 1989.

[34] C. Stirling, D. Walker. Local Model Checking in the Modal Mu-Calculus. TCS, 89, 1991. 161-177.

APPENDIX

First, we give some definition and notation.

Given a process p, a constant x of p is said to be *guarded in* p if x is contained in a sub-process of p of the form $\alpha.q$, where q is a process. A process p is *guarded* if every constant of p is guarded in p, it is *unguarded* otherwise. In the following, $Unfold^x(p)$ is the process obtained replacing each unguarded constant x by its definition. For example, if $x \stackrel{\text{def}}{=} \overline{a}.x$, $Unfold^x((a.b.x|x)\backslash\{a\})$ is the process $(a.b.x|\overline{a}.x)\backslash\{a\}$.

Given a process p, $\mathcal{F}irst(p) = \{\alpha \in \mathcal{A} \mid \exists p' \text{ s.t. } p \xrightarrow{\alpha} p'\}$ denotes the set of all actions that p can immediately perform. It can be syntactically defined as the least solution of the recursive definition in Table V.

$$
\begin{array}{lcl}
\mathcal{F}irst(nil) & = & \emptyset \\
\mathcal{F}irst(\alpha.p) & = & \{\alpha\} \\
\mathcal{F}irst(p+q) & = & \mathcal{F}irst(p) \cup \mathcal{F}irst(q) \\
\mathcal{F}irst(p\backslash L) & = & \mathcal{F}irst(p) - \overline{L} \\
\mathcal{F}irst(x) & = & \mathcal{F}irst(p) \quad \text{if } x \stackrel{\text{def}}{=} p \\
\mathcal{F}irst(p[f]) & = & \{f(\alpha) \mid \alpha \in \mathcal{F}irst(p)\} \\
\mathcal{F}irst(p|q) & = & \\
\multicolumn{3}{l}{\quad \begin{cases} \mathcal{F}irst(p) \cup \mathcal{F}irst(q) \cup \{\tau\} & \text{if } \exists \alpha \in \mathcal{F}irst(p) \text{ and} \\ & \quad \exists \overline{\alpha} \in \mathcal{F}irst(q) \\ \mathcal{F}irst(p) \cup \mathcal{F}irst(q) & \text{otherwise} \end{cases}}
\end{array}
$$

Table VII
DEFINITION OF FIRST ACTIONS.

We now formally define the function $\widehat{h}$.

Definition 5.1: Let p be a CCS process, $\mathcal{L}$ a set of visible actions, and $\mathcal{C}$ a set of pairs $\{\langle x, \mathcal{L}' \rangle\}$, where x is a constant occurring in p and $\mathcal{L}' \subseteq \mathcal{V}$. First, we define the auxiliary function $\widehat{h}$ with three arguments, $\widehat{h}(p, \mathcal{L}, \mathcal{C})$, inductively on p, as in Table VIII. Then, $\widehat{h}(p)$ is defined as: $\widehat{h}(p) = \widehat{h}(p, \emptyset, \emptyset)$.

The heuristic function $\widehat{h}(p, \mathcal{L}, \mathcal{C})$ is parametric with respect to a *restriction environment* $\mathcal{L}$, ($\mathcal{L} \subseteq \mathcal{V}$), which keeps the set of actions on which some restriction holds. The function is initially applied to a process with $\mathcal{L} = \emptyset$. The current environment $\mathcal{L}$ is modified when the function is applied to $p\backslash L$ (*Rule R5*): in this case the actions in $\overline{L}$ are added to $\mathcal{L}$. Note that we expand the body of a constant x each time the environment under which that constant is evaluated has changed (*Rule R7*). Initially, $\mathcal{C} = \emptyset$.

For $p = nil$ (*Rule R1*) the function $\widehat{h}$ returns 0 as this is a deadlock by definition.

When applied to $\alpha.p$ (*Rule R2*), the function returns 0 if α is a restricted action (i.e., $\alpha \in \mathcal{L}$), otherwise we recursively apply the function to find, if any, an action in $\mathcal{L}$. Roughly speaking, if α is restricted by $\mathcal{L}$ then $\alpha.p$ could not be able to move; thus, we optimistically return 0.

When the choice of two processes is encountered (*Rule R3*), the minimum number of actions between the two components is returned.

Let us consider the rule for the parallel composition of processes (*Rule R4*). The motivation for this rule is to provide some information also in the presence of several communication actions. However, this information is given without too much care of all the possible situations that may occur, in order to make the search faster. The rationale behind the definition of *Rule R4* is to assume the best case

$$
\begin{array}{lll}
\mathbf{R1}. & \widehat{h}(nil,\mathcal{L},\mathcal{C}) & = \quad 0 \\
\mathbf{R2}. & \widehat{h}(\alpha.p,\mathcal{L},\mathcal{C}) & = \quad \left\{ \begin{array}{ll} 0 & \text{if } \alpha \in \mathcal{L} \\ 1+\widehat{h}(p,\mathcal{L},\mathcal{C}) & \text{otherwise} \end{array} \right. \\
\mathbf{R3}. & \widehat{h}(p_1+p_2,\mathcal{L},\mathcal{C}) & = \quad \min(\widehat{h}(p_1,\mathcal{L},\mathcal{C}),\widehat{h}(p_2,\mathcal{L},\mathcal{C})) \\
\mathbf{R4}. & \widehat{h}(p_1|\cdots|p_n,\mathcal{L},\mathcal{C},\mathcal{U}) & = \\
 & \left\{ \begin{array}{ll} \widehat{h}(\mathit{Unfold}^x(p_1|\cdots|p_n),\mathcal{L},\mathcal{C},\mathcal{U}\cup\{x\}) & \text{if there exists an unguarded constant } x \\ & \text{in } p_1|\cdots|p_n \text{ with } x \notin \mathcal{U} \\ 1+\widehat{h}(p_1|\cdots|q|\cdots|p_n,\mathcal{L},\mathcal{C},\mathcal{U}) & \text{if } p_i \text{ is guarded, } i\in[1..n], \text{ and there} \\ & \text{exists } i\in[1..n] \text{ s.t. } p_i=\alpha.q, \text{ and } \alpha\notin\mathcal{L} \\ \mathit{number}(p_1|\cdots|p_n) & \text{if } p_i \text{ is guarded and } \mathcal{F}\mathit{irst}(p_i)\subseteq\mathcal{L},\ \forall i\in[i..n] \\ \sum_{i=1}^{n}\widehat{h}(p_i,\mathcal{L},\mathcal{C},\mathcal{U}) & \text{otherwise} \end{array} \right. \\
\mathbf{R5}. & \widehat{h}(p\backslash L,\mathcal{L},\mathcal{C}) & = \quad \widehat{h}(p,\mathcal{L}\cup\overline{L},\mathcal{C}) \\
\mathbf{R6}. & \widehat{h}(p[f],\mathcal{L},\mathcal{C}) & = \quad \widehat{h}(p,f^{-1}(\mathcal{L}),\mathcal{C}) \\
\mathbf{R7}. & \widehat{h}(x,\mathcal{L},\mathcal{C}) & = \quad \left\{ \begin{array}{ll} \infty & \text{if } \langle x,\mathcal{L}\rangle\in\mathcal{C} \\ \widehat{h}(p,\mathcal{L},\mathcal{C}\cup\{\langle x,\mathcal{L}\rangle\}) & \text{if } \langle x,\mathcal{L}\rangle\notin\mathcal{C} \text{ and } x\stackrel{\text{def}}{=}p \end{array} \right.
\end{array}
$$

Table VIII
THE $\widehat{h}$ FUNCTION.

where all the communication actions can be performed. First, we unfold ($\mathit{Unfold}^x(p)$ is the process obtained replacing each unguarded constant x by its definition) once the unguarded constants occurring in the parallel composition. This can be done storing the unfolded constants in a new set $\mathcal{U}$. Consider now the case where all the parallel components are guarded. If:

- there exists an independent component of the parallel composition, i.e., a process that can perform a non-restricted action ($p_i = \alpha.q$ and $\alpha \notin \mathcal{L}$); then the estimated number of actions to a deadlocked state is 1 plus the value returned by a recursive application of the function.
- all the components can perform only restricted actions, then the estimated number of actions to a deadlocked state is $\mathit{number}(p_1|\cdots|p_n)$, i.e., the number of the first actions on which two different processes can communicate. More precisely,

$$
\begin{array}{l}
\mathit{number}(p_1|\cdots|p_n) = \\
\quad |\{a | a\in\mathcal{F}\mathit{irst}(p_i), \overline{a}\in\mathcal{F}\mathit{irst}(p_j),\ i<j\}|
\end{array}
$$

In all the other cases the sum of the number of actions by each parallel process p_i is returned.

When considering a relabelled process (*Rule R6*), we must take as set of actions the set $f^{-1}(\rho) = \{\alpha \mid f(\alpha)\in\rho\}$, since now the interesting actions are also those relabelled by f in actions in $\mathcal{L}$.

Finally, (*Rule R7*), we return ∞ when we encounter a constant already expanded under the same environment (more precisely, when we encounter a constant x such that $\langle x,\mathcal{L}\rangle \in C$). In this case, no action in $\mathcal{L}$ has been found, and this means that the state under consideration is safe: a state that can perform actions in $\mathcal{L}$ is more promising. Actually, when we reach a state whose $\widehat{h}$-value is ∞, we will never expand that state, because surely it will be deadlock free.