Málaga, 22 de Noviembre de 2008

Informe Ejecutivo

TÍTULO: ACO-1.0-2008: Optimización basada en colonias de hormigas para la búsqueda de violaciones de propiedades de viveza en software concurrente

RESUMEN: *Model checking* es una técnica formal automática para comprobar propiedades del software. Estas propiedades normalmente se expresan usando fórmulas en lógicas temporales como LTL o CTL. La mayoría de los *model checkers* usan algoritmos exhaustivos deterministas para encontrar ejecuciones del software que violen alguna propiedad (contraejemplo) o para demostrar que el programa cumple las propiedades especificadas. Estos algoritmos requieren una gran cantidad de memoria cuando el modelo es muy grande y en ocasiones no pueden finalizar la comprobación. En trabajos anteriores hemos aplicado la optimización basada en colonias de hormigas (ACO) al problema de búsqueda de violaciones de propiedades de seguridad en software concurrente. En este documento presentamos una extensión de dichos trabajos al considerar propiedades de viveza. Las propiedades de viveza requieren para su comprobación una búsqueda más sofisticada que la usada en el caso de propiedades de seguridad. Hemos desarrollado un algoritmo basado en ACO para realizar esta búsqueda. Aquí presentamos el algoritmo y algunos resultados de su aplicación en programas concurrentes.

OBJETIVOS:

1. Presentar ACOhg-live, un algoritmo basado en ACO para buscar violaciones de propiedades de viveza en programas concurrentes.
2. Mostrar algunos resultados de ACOhg-live obtenidos tras aplicarlo a programas concurrentes con errores de viveza.
3. Comparar los resultados de ACOhg-live con los de Nested-DFS, el algoritmo tradicionalmente usado para este problema en la literatura.

CONCLUSIONES:

1. Los resultados muestran que ACOhg-live es capaz de superar a Nested-DFS en eficacia y eficiencia.
2. ACOhg-live requiere una cantidad pequeña de memoria y es capaz de encontrar errores en modelos en los que Nested-DFS falla debido a la falta de memoria.
3. ACOhg-live debería usarse en las primeras etapas del desarrollo del software y después de cada modificación de mantenimiento.
4. ACOhg-live puede también usarse al final del ciclo de desarrollo software para asegurar con alta probabilidad que el software satisface una determinada propiedad deseable.
5. En los sistemas críticos se debe usar un algoritmo exhaustivo (si es posible) en las etapas finales para verificar que el software realmente satisface las propiedades deseadas.
6. La principal desventaja de ACOhg-live desde el punto de vista de su aplicabilidad es la gran cantidad de parámetros del algoritmo.

RELACIÓN CON ENTREGABLES: PRE: SOFTW-1.0-2008 (lectura obligatoria)

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: ACO-1.0-2008: Ant Colony Optimization for Searching for Liveness Property Violations in Concurrent Software

ABSTRACT: Model Checking is a well-known and fully automatic technique for checking software properties, usually given as temporal logic formulae (LTL or CTL) on the program variables. Most of model checkers found in the literature use exact deterministic algorithms to check the properties. These algorithms usually require huge amounts of memory if the checked model is large. In previous work we used Ant Colony Optimization (ACO) for searching for safety property violations in concurrent software. In this deliverable we present an extension of the previous work dealing with liveness properties. The search for liveness property violations is more complex than the one used in the case of safety properties. We have developed an algorithm based on ACO for accomplishing this complex search. In this document we present the algorithm, called ACOhg-live, and some results after its application to concurrent software.

GOALS:

1. Present ACOhg-live, an ACO-based algorithm for searching for liveness property violations in concurrent software.
2. Show some results of ACOhg-live after its application to concurrent software.
3. Compare the ACOhg-live results against the ones of traditional algorithms used for this problem in the literature.

CONCLUSIONS:

1. The results show that ACOhg-live is able to outperform Nested-DFS in efficacy and efficiency.
2. ACOhg-live requires a very low amount of memory and it is able to find errors even in models in which Nested-DFS fails in practice due to memory requirements.
3. ACOhg-live should be used in the first/middle stages of the software development and after any maintenance modification made on the concurrent system.
4. ACOhg-live can also be used at the end of the software life cycle to assure with high probability that the software satisfies a given desirable property.
5. In critical systems an exhaustive algorithm must be used (if possible) in the final stages to verify that the software really satisfies the liveness property.
6. The main drawback of ACOhg-live from the point of view of the applicability is the large amount of parameters of the algorithm.

RELATION WITH DELIVERABLES: PRE: SOFTW-1.0-2008 (mandatory reading)

Ant Colony Optimization for Searching for Liveness Property Violations in Concurrent Software

DIRICOM

November 2008

1 Introduction

Model checking [10] is a well-known and fully automatic formal method that can check properties of software systems. In model checking, all the possible program states are analyzed (in an explicit or implicit way) in order to prove (or refute) that the program satisfies a given property such as absence of deadlocks or starvation. Some other more general properties can be specified using a temporal logic like Linear Temporal Logic (LTL) [9] or Computation Tree Logic (CTL) [8]. When the property is specified using LTL, model checkers transform the model and the negation of the LTL formula into automata in order to perform their intersection. The intersection automaton (called Büchi automaton) is exhaustively 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 fulfils the property and the verification ends with success.

The amount of states of the transition graph or the Büchi automaton associated to a concurrent model is very large even in the case of small systems, and it increases exponentially with the size of the model. This fact is known as *the state explosion problem* and limits the size of the model that an explicit state model checker can verify. This limit is reached when the model checker is not able to explore more states due to the absence of free computer memory. Therefore, techniques of bounded (low) complexity as metaheuristics will be needed for medium/large size programs working in real world scenarios.

In this document we present an algorithm based on a new kind of Ant Colony Optimization (ACO) called ACOhg for searching for liveness property violations in concurrent systems. This document belongs to the research line opened in [2] and extends the approach presented for safety property violations in [3] and [6] to liveness properties.

The deliverable is organized as follows. The next section presents the details of the problem and reformulates it as an optimization problem. Section 3 describes our algorithmic proposal, ACOhg-live, for tackling the problem. In Section 4 we present some experimental results obtained with ACOhg-live. In the same section we compare the results of ACOhg-live against the traditional algorithm utilized for checking liveness properties in explicit state model checking: Nested-DFS. Finally, Section 5 outlines the conclusions.

2 The Optimization Problem Addressed

The properties that can be specified with LTL formulae can be classified into two groups: *safety* and *liveness* properties [16]. Safety properties are those in which a finite execution can be a counterexample, while liveness properties can only be violated by infinite executions (for a formal definition of safety and liveness see [4]). Safety properties can be checked by searching for a single accepting state in the intersection Büchi automaton of the concurrent model and the negation of the LTL formula. That is, when safety properties are checked, it is not required to find an additional cycle containing the accepting state. This means that safety property verification can be transformed into the search for one objective node (one accepting state) in a graph (intersection Büchi automaton) and general graph exploration algorithms like DFS and BFS can be applied to the problem. Furthermore, in [12] and [13] the authors utilize heuristic information for guiding the search. They assign a heuristic value to each state that depends on the safety property to verify. After that, they utilize classical algorithms for graph exploration such as A*, Weighted A* (WA*), Iterative Deepening A* (IDA*), and Best First Search (BF). The results show that, by using heuristic search, the length of the counterexamples can be shortened (they can find optimal error trails using A* and BFS) and the amount of memory required to obtain an error trail is reduced, allowing the exploration of larger models.

The utilization of heuristic information for guiding the search for errors in model checking is known as *heuristic* or *directed model checking*. The heuristics are designed to lead the exploration first to the region of the state space in which an error is likely to be found. This way, the time and memory required to find an error in faulty concurrent systems is reduced in average. Different kinds of heuristic functions have been defined in the past. We use here two kinds of heuristics: formula-based and state-based heuristics. Formula-based heuristics are based on the expression

of the LTL formula checked. These heuristics estimate the number of transitions required to get an accepting state from the current one. In our algorithm we use a formula-based heuristic that is defined in [12]. State-based heuristics can be used when the objective sate is known. These heuristics estimate the number of transitions required to get the desired state. We use in our algorithm the distance of finite state machines, that is the sum of the minimum number of transitions required to reach the objective state from the current one in the local automata of each process.

The search for liveness errors requires to find an accepting state and a cycle involving the accepting state. In this case, the traditional algorithm applied is Nested-DFS [14], which is an exhaustive deterministic algorithm. When the search for errors with a low amount of computational resources (memory and time) is a priority (for example, in the first stages of the implementation of a program), non-exhaustive algorithms using heuristic information can be used. A well-known class of non-exhaustive algorithms for solving complex problems is the class of metaheuristic algorithms [5]. They are search algorithms used in optimization problems that can find good quality solutions in a reasonable time. The search for accepting states in the Büchi automaton can be translated into an optimization problem and, thus, metaheuristic algorithms can be applied to the search for errors.

ACO is a metaheuristic designed for searching short paths in graphs. This makes it very suitable for the problem at hand and for this reason our proposal is based on ACO. In order to guide the search we use some of the heuristic functions defined in [12]. In fact, we have extended their experimental model checker, HSF-SPIN, in order to include our ACOhg-live algorithm. In this way, we can use all the heuristic functions implemented in HSF-SPIN and, at the same time, all the existing work related to parsing Promela models and interpreting them.

2.1 Problem Formalization

The search for liveness errors can be translated into the search of a path in a graph (the intersection Büchi automaton) starting in the initial state and ending in an objective node (accepting state) and an additional cycle involving the accepting state. We formalize here the problem as follows.

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

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

The graph G used in the problem is derived from the intersection Büchi automaton B of the model and the negation of the LTL formula of the property. The set of nodes S in G is the set of states in B, the set of arcs T in G is the set of transitions in B, the initial node q in G is the initial state in B, and the set of objective nodes F in G is the set of accepting states in B. In the following, we will also use the words *state*, *transition*, and *accepting state* to refer to the elements in S, T, and F, respectively.

3 Algorithm

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

Algorithm 1 ACOhg-live

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

The search that ACOhg-live performs is composed of two different phases (see Fig. 1 for an illustration of the search in the two phases). In the first one, ACOhg is utilized for finding accepting states in the Büchi automaton (line 2 in Algorithm 1). In this phase, the search of ACOhg starts in the initial node of the graph and the algorithm searches for accepting states. If they are found, in a second phase a new search is performed using ACOhg again for each accepting state discovered (lines 3 to 8). In this second search the objective is to find a cycle involving the accepting state. The search starts in one accepting state and the algorithm searches for the same state in order to find a cycle. If a cycle is found ACOhg-live returns the complete accepting path (line 6). If no cycle is found for any of the accepting states ACOhg-live runs again the first phase after including the accepting states in a tabu list (line 9). This tabu list prevents the algorithm from searching again cycles containing the just explored accepting states. If one of the accepting states in the tabu list is reached, ACOhg expands the state as a normal state and it is not included in the list of accepting states to be explored in the second phase. ACOhg-live alternates between the two phases until no accepting state is found in the first one (line 10).

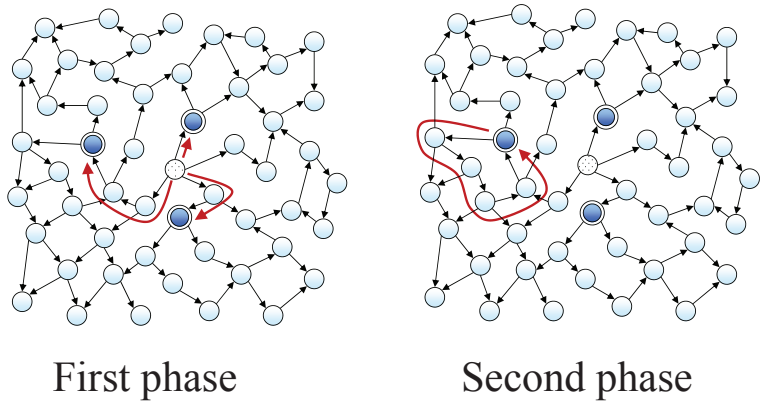

Figure 1: An illustration of the search that ACOhg-live performs in the first and second phase

The configuration of the ACOhg algorithms are, in general, different in the two phases since they tackle different objectives. We highlight this fact by using different variables for referring to both algorithms in Algorithm 1: `acohg1` and `acohg2`. For example, in the first phase a more exploratory search is required in order to find a diverse set of accepting states. In addition, the accepting states are not known and no state-based heuristic can be used; a formula-based heuristic must be used instead. On the other hand, in the second phase the search must be guided to search for one concrete state and, in this case, a state-based heuristic is more suitable.

4 Experiments

In this section we present some results obtained with our ACOhg-live algorithm. For the experiments we have selected three Promela models (two of them scalable) that are presented in the following section. After that, we discuss the algorithm parameters used in the experiments in Section 4.2. In Section 4.3 we compare the results obtained with ACOhg-live against Nested-DFS, the traditional algorithm utilized for model checking in explicit state model checkers.

4.1 Promela Models

We have selected three Promela models implementing faulty concurrent systems previously reported in the literature [13]. All these models violate a liveness property that is specified in LTL. In Table 1 we present the models with some information about them (lines of code, scalability, number of processes, and the LTL formulae they violate). They can be found with the source code of ACOhg and HSF-SPIN in `http://oplink.lcc.uma.es/software`. See [7] for a description of the models.

Table 1: Promela models used in the experiments

Model	LoC	Scalable	Processes	LTL formula
alter	64	no	2	$\Box(p \rightarrow \Diamond q) \wedge \Box(r \rightarrow \Diamond s)$
giopij	740	yes	$i + 3(j + 1)$	$\Box(p \rightarrow \Diamond q)$
phi$j$	57	yes	$j + 1$	$\Box(p \rightarrow \Diamond q)$

4.2 Parameters of the Algorithm

The parameters used in the experiments for the ACOhg algorithms in the two phases of ACOhg-live are shown in Table 2. These parameters are not set in an arbitrary way: they are the result of a previous study aimed at finding the best configuration for them. One portion of this study was published in [1]. As mentioned in Section 3, in the first phase we use an explorative configuration ($\xi = 0.7$, $\lambda_{ant} = 20$) while in the second phase the configuration is adjusted to search in the region near the accepting state found (intensification).

Table 2: Parameters for the ACOhg algorithms in ACOhg-live

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

With respect to the heuristic information, we use the formula-based heuristic defined in [11] in the first phase of the search when the objective is to find accepting states. In the second phase we use the distance of finite state machines.

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

4.3 ACOhg-live vs. Nested-DFS

In this section we compare the results obtained with ACOhg-live against the ones obtained with Nested-DFS. This last algorithm is deterministic and for this reason we only perform one single run. In Table 3 we show the results of both algorithms (for ACOhg-live we show the average and the standard deviation). For comparing the hit rate we use the Westlake-Schuirmann test. However, for the other measures we utilize the one sample Wilcoxon sign rank test because we compare one sample (the results of ACOhg-live) with one single value (the result of Nested-DFS).

The first observation concerning the hit rate is that ACOhg-live is the only one that is able to find error paths in all the models. Nested-DFS is not able to find error paths in `giop6` and `giop10` because it requires more than the memory available in the machine used for the experiments (512 MB). This result is very important since Nested-DFS is a standard *de facto* in the formal methods community for checking liveness properties. Our first conclusion is that ACOhg-live is able to find error paths running in regular machines while Nested-DFS is not. If we focus on the remaining models we observe a similar hit rate in both algorithms (in Nested-DFS the hit rate is always 100% since it is a deterministic algorithm).

With respect to the length of the error paths we observe that ACOhg-live obtains shorter error executions than Nested-DFS in all the models (with statistical significance). The difference increases with the size of the models. For example, for `alter` the average length obtained with ACOhg-live (30.68) is half the length obtained with Nested-DFS (64.00) and for `phi8` (that is a larger model) the length obtained with ACOhg-live (51.36) is approximately one sixtieth of the length obtained with Nested-DFS (3405.00). The biggest differences can be observed in `phi14` and `phi20` in which ACOhg-live finds error paths that are more than one hundred times shorter than the ones found by Nested-DFS. Furthermore, we limited the exploration depth of Nested-DFS to 10000 in order to avoid stack overflow problems. If we allowed Nested-DFS to explore deeper regions we would obtain longer error paths with Nested-DFS. In fact, we run Nested-DFS using a depth limit of 80000 in `phi20` and we got an error path of 80001 states. This means that the lengths of the error paths that are shown in Table 3 for Nested-DFS in `phi14` and `phi20` are in fact a lower bound of the real length that Nested-DFS would obtain in theory. In conclusion, ACOhg-live obtains error paths that are by far shorter than the ones obtained with Nested-DFS. This is a very important result since short error paths are preferred in order for the programmers to understand faster what is wrong in the concurrent model.

Table 3: Comparison between ACOhg-live and Nested-DFS

Models	Measure	ACOhg-live	Nested-DFS	Test
alter	Hit rate	100/100	1/1	-
	Length	30.68 $_{10.72}$	64.00	+
	Mem. (KB)	1925.00 $_{0.00}$	1873.00	+
	Time (ms)	90.00 $_{13.86}$	0.00	+
giop2	Hit rate	100/100	1/1	-
	Length	43.76 $_{5.82}$	298.00	+
	Mem. (KB)	2953.76 $_{327.48}$	7865.00	+
	Time (ms)	747.50 $_{408.09}$	240.00	+
giop6	Hit rate	100/100	0/1	+
	Length	58.77 $_{7.21}$	•	•
	Mem. (KB)	5588.04 $_{631.36}$	•	•
	Time (ms)	8733.50 $_{3304.90}$	•	•
giop10	Hit rate	86/100	0/1	+
	Length	62.85 $_{7.03}$	•	•
	Mem. (KB)	9316.67 $_{700.44}$	•	•
	Time (ms)	43059.07 $_{21417.74}$	•	•
phi8	Hit rate	100/100	1/1	-
	Length	51.36 $_{6.95}$	3405.00	+
	Mem. (KB)	2014.32 $_{18.87}$	4005.00	+
	Time (ms)	2126.10 $_{479.64}$	40.00	+
phi14	Hit rate	99/100	1/1	-
	Length	76.05 $_{9.35}$	10001.00	+
	Mem. (KB)	2496.07 $_{41.81}$	59392.00	+
	Time (ms)	8070.30 $_{1530.12}$	2300.00	+
phi20	Hit rate	98/100	1/1	-
	Length	97.39 $_{10.14}$	10001.00	+
	Mem. (KB)	3244.67 $_{91.33}$	392192.00	+
	Time (ms)	18064.90 $_{5538.30}$	17460.00	-

If we focus on the computational resources we observe that ACOhg-live requires less memory than Nested-DFS to find the error paths with the only exception of `alter`. The biggest differences are those of `giop6` and `giop10` in which Nested-DFS requires more than 512 MB of memory while ACOhg-live obtains error paths with 38 MB at most. Memory is the main problem of the traditional model checking techniques and we can observe here that ACOhg-live is more tolerant to the state explosion problem. With respect to the time required for the search, Nested-DFS is faster than ACOhg-live. The mechanisms included in ACOhg-live in order to be able to find short error paths with high hit rate and low amount of memory extend the time required for the search. Anyway, the maximum difference with respect to the time is around six seconds (in `phi14`), which is not too much if we take into account that the error path obtained is much shorter.

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

5 Conclusions

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

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

First of all, it must be clear from the beginning that ACOhg-live can find short counterexamples in faulty concurrent systems, but it cannot be used for verifying that a concurrent system satisfies a given liveness property.

Thus, ACOhg-live should be used in the first/middle stages of the software development and after any maintenance modification made on the concurrent system. In these phases errors are expected to exist in the concurrent software. In spite of the previous considerations, ACOhg-live can also be used to assure with high probability that the software satisfies a given desirable property (perhaps obtained from the specification). In this case it can be used at the end of the software life cycle. Unlike this, in critical systems (like airplane controllers) an exhaustive algorithm must be used in the final stages to verify that the software really satisfies the liveness property.

The main drawback we have found from the point of view of the applicability of ACOhg-live is the large amount of parameters of the algorithm. These parameters make ACOhg-live more flexible, since it is possible to tackle models with different features changing the parameterization. However, software practitioners have no time to adjust the parameters of the algorithm and they want a robust algorithm that works well in most situations with minimum cost. In this sense, we understand that a parameterization study must be a priority in the following steps of this research. In fact, from the experiments performed for this and previous work we have outlined a set of rules for assigning values to the parameters (some of them are published in [1]).

References

[1] Enrique Alba and Francisco Chicano. ACOhg: Dealing with huge graphs. In *Proceedings of the Genetic and Evolutionary Conference*, pages 10–17, London, UK, July 2007. ACM Press.

[2] Enrique Alba and Francisco Chicano. Ant colony optimization for model checking. In *EUROCAST 2007*, volume 4739 of *Lecture Notes in Computer Science*, pages 523–530, Gran Canaria, Spain, February 2007.

[3] 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.

[4] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inform. Proc. Letters*, 21:181–185, 1985.

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

[6] Francisco Chicano and Enrique Alba. Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models. *Information Processing Letters*, 2007. (to appear).

[7] Francisco Chicano and Enrique Alba. Finding liveness errors with ACO. In *Proceedings of the Conference on Evolutionary Computation*, pages 3002–3009. IEEE Computer Society, 2008.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[9] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

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

[11] 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.

[12] 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.

[13] 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.

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