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-2.0-2008: ACO combinado con reducción de orden parcial para model checking

Resumen: En este documento analizamos la combinación de ACOhg con la reducción de orden parcial (POR) para resolver el problema de encontrar violaciones de propiedades de seguridad en sistemas concurrentes usando un enfoque basado en model checking. ACOhg es un algoritmo basado en colonias de hormigas capaz de trabajar con grafos de gran extensión. Por otro lado, la reducción de orden parcial es una técnica bien conocida en el dominio de model checking para reducir el espacio de búsqueda. Presentamos un estudio experimental en el que se compara ACOhg con y sin reducción de orden parcial. Incluimos además en la comparación el algoritmo A* como algoritmo base, ya que es popular en el dominio del model checking guiado. Los resultados muestran que la combinación de ambos reduce el tiempo de cómputo necesario para encontrar los errores.

Objetivos:

1. Combinar ACOhg con la reducción de orden parcial en model checking.
2. Analizar la combinación para comprobar los beneficios de la misma.

Conclusiones:

1. La memoria que requiere ACOhg con POR es menor que la que requiere ACOhg sin POR.
2. El número de estados expandidos sigue la misma tendencia que la longitud de las trazas de error debido a la dependencia lineal entre ambos en ACOhg.
3. La longitud de las trazas de error es menor en ACOhg con POR que sin POR en seis de los nueve modelos analizados.
4. El tiempo de cómputo requerido por ACOhg con POR es hasta 6.8 veces menor que el tiempo requerido por ACOhg sin POR.
5. A* no puede encontrar errores en la mayoría de los modelos usados en los experimentos.

Relación con entregables: PRE: SOFTW-1.0-2008 (lectura necesaria)

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-2.0-2008: ACO Combined with Partial Order Reduction for Model Checking

ABSTRACT: In this deliverable we analyze the combination of ACOhg plus partial order reduction applied to the problem of finding safety property violations in concurrent models using a model checking approach. ACOhg is an ant colony optimization algorithm that is able to deal with huge graphs. On the other hand, partial order reduction (POR) is a well-known technique in the model checking field to reduce the state space. We present an experimental study in which we compare ACOhg with POR against ACOhg without POR. We also include A* in the comparison as a base algorithm, since it is well-known in the field of heuristic model checking. The results show that the combination of ACOhg and POR is computationally beneficial for the search.

GOALS:

1. Combine ACOhg with partial order reduction in model checking.
2. Analize the combination to study its benefits.

CONCLUSIONS:

1. The memory required by ACOhg with POR is always smaller than the one required by ACOhg without POR.
2. The number of expanded states has the same trend as the length of the error paths due to the linear dependence between them for the ACOhg algorithms.
3. The length of the error trails is smaller for ACOhg with POR than ACOhg without POR in six of the nine analyzed models.
4. The CPU time required by ACOhg with POR is up to 6.8 times lower than the time required by ACOhg without POR.
5. A* cannot find errors in most of the models used for the experiments

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

ACO Combined with Partial Order Reduction for Model Checking

DIRICOM

November 2008

1 Introduction

Model checking [3] 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) or Computation Tree Logic (CTL).

We introduced in deliverable ACO-1.0-2008 the motivation for using metaheuristic algorithms and ACO in particular for this problem. In this deliverable we focus on the combination of ACOhg with partial order reduction (POR). and we present some results comparing the algorithm with and without POR.

The deliverable is organized as follows. The next section presents background information on partial order reduction. In Section 3 we present some experimental results comparing ACOhg with and without POR in a benchmark of nine concurrent models. We also include the results of *A algorithm. Finally, Section 4 outlines the conclusions. In this deliverable we do not describe ACOhg . For a description of both algorithms the reader should see deliverable ACO-1.0-2008.

2 Background

Partial order reduction (POR) is a method that exploits the commutativity of asynchronous systems in order to reduce the size of the state space. The interleaving model in concurrent models imposes an arbitrary ordering between concurrent events. When the Büchi automaton of the system is built, the events are interleaved in all possible ways. The ordering between independent concurrent instructions is meaningless. Hence, we can consider just one ordering for checking one given property since the other orderings are equivalent. This fact can be used to construct a reduced state graph hopefully much easier to explore compared to the full state graph (original Büchi automaton).

We use here a POR proposal based on *ample sets* [5]. Before giving more details on POR, we need to introduce some terminology. We call *transition* to a partial function $\gamma : S \rightarrow S$. Intuitively, a transition corresponds to one instruction in the program code of the concurrent model. The set of all the transitions that are defined for state s is denoted with $enabled(s)$. According to these definitions, the set of successors of s must be $T(s) = \{\gamma(s) | \gamma \in enabled(s)\}$. In short, we say that two transitions γ and δ are *independent* when they do not disable one another and executing them in either order results in the same state. That is, for all s if $\gamma, \delta \in enabled(s)$ it holds that:

1. $\gamma \in enabled(\delta(s))$ and $\delta \in enabled(\gamma(s))$
2. $\gamma(\delta(s)) = \delta(\gamma(s))$

Let $L : S \rightarrow 2^{AP}$ be a function that labels each state of the Büchi automaton B with a set of atomic propositions from AP. In the Büchi automaton of a concurrent system, this function assigns to each state s the set of propositions appearing in the LTL formula that are true in s. One transition γ is *invisible* with respect to a set of propositions $AP' \subseteq AP$ when its execution from any state does not change the value of the propositional variables in AP', that is, for each state s in which γ is defined, $L(s) \cap AP' = L(\gamma(s)) \cap AP'$.

The main idea of ample sets is to explore only a subset $ample(s) \subseteq enabled(s)$ of the enabled transitions of each state s such that the reduced state space is equivalent to the full state space. This reduction of the state space is performed on-the-fly while the graph is generated. In order to keep the equivalence between the complete and the reduced Büchi automaton, the reduced set of transitions must fulfil the following conditions [3]:

- **C0**: for each state s, $ample(s) = \emptyset$ if and only if $enabled(s) = \emptyset$.
- **C1**: for all state s and all path in the full state graph that starts at s, a transition γ that is dependent on a transition $\delta \in ample(s)$ cannot be executed without a transition in $ample(s)$ occurring previously.
- **C2**: for all state s, if $enabled(s) \neq ample(s)$ then all transition $\gamma \in ample(s)$ is invisible with respect to the atomic propositions of the LTL formula being verified.

- **C3**: a cycle is not allowed if it contains a state in which some transition γ is enabled but never included in $ample(s)$ for any state s of the cycle.

The first three conditions are not related to the particular search algorithm being used. However, the way of ensuring **C3** depends on the search algorithm. In [5] three alternatives for ensuring that **C3** is fulfilled were proposed. From them, the only one that can be applied to any possible exploration algorithm is the so-called $\mathbf{C3}_{static}$ and this is the one we use in our experiments. In order to fulfil condition $\mathbf{C3}_{static}$, the structure of the processes of the model is statically analyzed and at least one transition on each local cycle is marked as *sticky*. Condition $\mathbf{C3}_{static}$ requires that states s containing a sticky transition in $enabled(s)$ be fully expanded: $ample(s) = enabled(s)$. This condition is also called **c2s** in a later work by Bošnački *et al.* [1].

3 Experimental Section

In this section we are going to analyze how the combination of partial order reduction plus ACOhg can help in the search for safety property violations in concurrent models. For the experiments we implemented ACOhg inside the HSF-SPIN model checker.

We apply our algorithms to a benchmark of concurrent models codified in Promela. In fact, we have selected three scalable models used by Edelkamp et al. [5] in the past: `giop`, `leader`, and `marriers`. For a description of the models see [2].

We use two algorithms for the experiments: ACOhg and ACOhg^{POR}, the combination of ACOhg plus POR. The parameters used for the two algorithms are the ones shown in Table 1. 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 heuristic function h used is the one that assigns to each state s the number of active processes in the state (for `giop` and `marriers`) and a formula-based heuristic h_φ [4] (in the case of `leader`). The machine used in the experiments is a Pentium 4 at 2.8 GHz with 512 MB of RAM.

Table 1: Parameters for ACOhg and ACOhg^{POR}.

Parameter	Value	Parameter	Value
$msteps$	1000	a	5
$colsize$	10	ρ	0.2
λ_{ant}	20	α	1.0
σ_s	4	β	2.0
ι	10	p_p	1000
ξ	0.5	p_c	1000

3.1 Results

In Table 2 we present the results of applying ACOhg and ACOhg^{POR} to nine models: three instances of each scalable model presented above (small, medium, and large). The information reported is the length of the error paths, the memory required (in Kilobytes), the number of expanded states during the search, and the CPU time required (in milliseconds). In order to clarify that the reduced amount of memory required by the ACOhg algorithms is not due to the use of the heuristic information (h), we also show the results obtained with A* (implemented in HSF-SPIN) for all the models using the same heuristic functions as the ACOhg algorithms. This clearly states that memory reduction is a very appealing attribute of the algorithm itself.

3.1.1 Computational Resources

The first observation is that ACOhg and ACOhg^{POR} both require less memory to find an error path than A* in all the models. This means that the success of ACOhg algorithms is not due to the heuristic information (A* also uses heuristic information). Heuristic information helps to find an error path using less resources since it guides the search, but the way in which ACOhg algorithms perform the search and their particular mechanisms for saving memory have a major impact in reducing the computational resources required.

A second observation is that ACOhg^{POR} requires less computational resources (memory and CPU time) than ACOhg in all the models. A statistical test (with significance level $\alpha = 0.05$) shows that all the differences in the memory and the CPU time required by ACOhg and ACOhg^{POR} in Table 2 are significant. Thus, we can state after these experiments that ACOhg^{POR} outperforms the performance of ACOhg, what confirms our expectations. In Figure 1 we can clearly see the advantage of ACOhg^{POR} against ACOhg with respect to the memory required (average of the 100 independent runs). The difference between both algorithms is still larger for the `leader`i and `marriers`i models. We have also drawn a line for each scalable model indicating how the amount of memory required increases with the parameter of the model. We can observe in `leader` and `marriers` that the growth is almost linear.

Table 2: Results (mean and standard deviation values) of ACOhg and ACOhg^{POR} (bold values represent best results). We also include the results obtained with A*.

Models	Measures	ACOhg		ACOhg^{POR}		A*
giop21	Length	42.30	1.71	**42.10**	0.99	42.00
	Mem. (KB)	3428.44	134.95	**2979.48**	98.33	27648.00
	Exp. states	1844.10	29.39	**1831.64**	26.96	26470.00
	CPU (ms)	202.00	9.06	**162.50**	5.55	1000.00
giop41	Length	70.21	7.56	**59.76**	5.79	-
	Mem. (KB)	9523.67	331.76	**7420.08**	422.94	-
	Exp. states	2663.91	325.19	**2347.94**	363.91	-
	CPU (ms)	354.50	42.39	**264.90**	40.46	-
giop61	Length	67.59	13.43	**61.74**	3.16	-
	Mem. (KB)	11970.56	473.59	**11591.68**	477.67	-
	Exp. states	2603.47	597.36	**2398.55**	378.58	-
	CPU (ms)	440.60	71.02	**391.70**	43.86	-
leader6	Length	**50.90**	4.52	56.36	3.04	37.00
	Mem. (KB)	16005.12	494.39	**3710.64**	410.29	132096.00
	Exp. states	**1894.28**	22.38	1955.23	82.64	21332.00
	CPU (ms)	494.00	21.12	**98.80**	8.16	1250.00
leader8	Length	**60.83**	4.66	74.11	4.51	-
	Mem. (KB)	24381.44	515.98	**4831.40**	114.10	-
	Exp. states	**2344.63**	320.90	2749.75	12.29	-
	CPU (ms)	1061.20	211.47	**198.90**	4.67	-
leader10	Length	**73.84**	4.79	80.86	6.36	-
	Mem. (KB)	30167.04	586.82	**7178.05**	2225.78	-
	Exp. states	**2764.42**	53.06	3114.22	315.07	-
	CPU (ms)	1910.70	45.02	**294.90**	66.96	-
marriers10	Length	307.11	34.87	**233.19**	21.91	-
	Mem. (KB)	34170.88	494.39	**18319.36**	804.93	-
	Exp. states	12667.11	1420.18	**9614.15**	1032.06	-
	CPU (ms)	8847.00	634.06	**1306.60**	126.56	-
marriers15	Length	540.41	60.88	**395.10**	40.07	-
	Mem. (KB)	51148.80	223.18	**26050.56**	1256.81	-
	Exp. states	22506.36	2526.52	**16458.42**	1671.93	-
	CPU (ms)	19740.50	1935.54	**3595.00**	316.59	-
marriers20	Length	793.62	80.45	**569.99**	54.63	-
	Mem. (KB)	68003.84	503.64	**33351.68**	1442.75	-
	Exp. states	33108.85	3364.88	**23747.43**	2309.39	-
	CPU (ms)	49446.30	7557.40	**8174.00**	707.71	-

This is a very promising result, since the number of states of the Büchi automaton usually grows in an exponential way when the parameter of the model increases linearly. This means that even with an exponential growth in the size of the Büchi automaton the memory required by ACOhg and ACOhg^{POR} grows in a linear way.

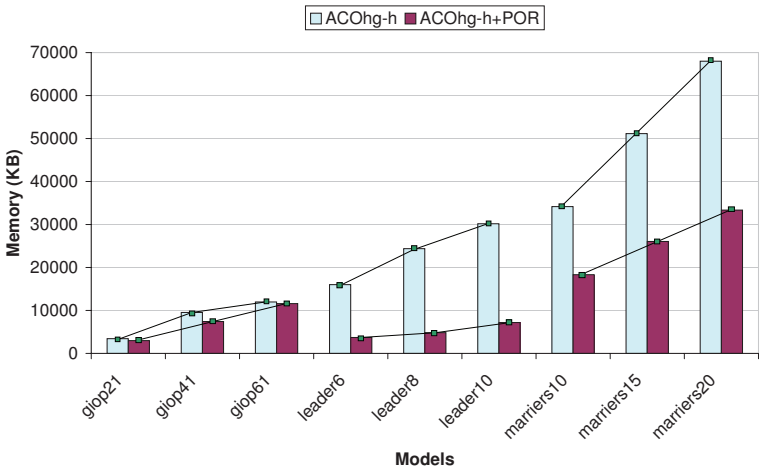

Figure 1: Memory required by ACOhg and ACOhg^{POR} for all the models.

Finally, we present in Figure 2 the evolution of the memory required by ACOhg and ACOhg^{POR} in the first 50 steps of their execution for `marriers20` (the figure presents the average of the 100 independent runs). In this figure we can clearly notice the effect of removing the pheromone trails (and the arcs associated to them) after one stage. When one stage finishes the required memory fall down and then it increases progressively in the new stage. This gives the shape of saw to Figure 2. We can also observe in this figure that the slope of the lines in ACOhg^{POR} is smaller. This indicates that the number of different paths traversed by the ants have been reduced due to the use of partial order reduction. In the first eight steps (two stages) of ACOhg^{POR} the memory is almost constant. This means that in the reduced state space there is only one possible starting path of length $2\lambda_{ant}$.

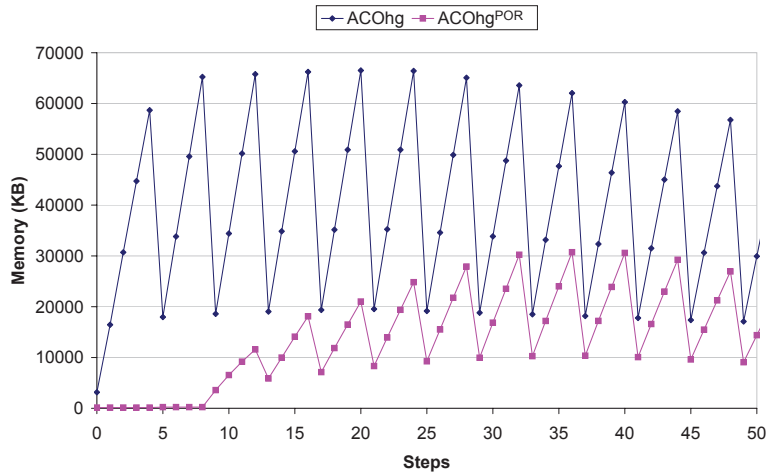

Figure 2: Evolution of the average memory required by ACOhg and ACOhg^{POR} in the first 50 steps for `marriers20`.

3.1.2 Expanded States

In this implementation of ACOhg and ACOhg^{POR}, the minimum number of steps required for finding an accepting state of the Büchi automaton is the length of the error path divided by λ_{ant} and multiplied by the number of steps per stage σ_s, since in each stage the depth of the exploration region is increased by λ_{ant}. This reasoning gives us an expression that relates the length of the error paths (len) with the minimum number of expanded states: $exp_{min} \approx colsize \cdot \sigma_s \cdot \lambda_{ant} \cdot \lceil len/\lambda_{ant} \rceil$. This expression is only valid for ACOhg and ACOhg^{POR}, it is not a general algorithm-independent formula. According to the previous expression, there is a quasi-linear relation between the length of the error paths and the number of expanded states, what explains why both measures are reduced at the same time in Table 2. Furthermore, the quotient exp_{min}/len must approximately be $colsize \cdot \sigma_s$. We can corroborate this prediction in Figure 3, where we plot the number of expanded states against the length of the error paths for all the independent runs and all the models in our experiments. The slope of the best fit line is 41.96, quite close to $colsize \cdot \sigma_s = 40$, the predicted value for the quotient exp_{min}/len.

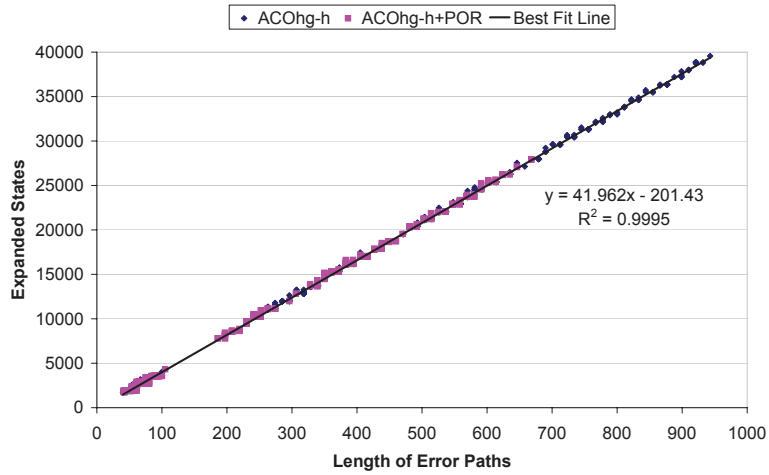

Figure 3: Linear dependence between the expanded states and the error paths length.

3.1.3 Length of Error Paths

We can observe in Table 2 that ACOhg^{POR} obtains shorter error paths than ACOhg in the `giop`ij and `marriers`i models. Furthermore, we can notice that ACOhg^{POR} greatly reduces the error paths obtained by ACOhg in the `marriers`i models. In the three `leader`i models the length of the error paths obtained by ACOhg^{POR} is only a few states longer than the one obtained by ACOhg. We must remind here that the objective in these experiments is not to minimize the length of the error paths, but to find an error path. In spite of this fact, the error paths obtained by ACOhg algorithms are near the optimum ones, at least in `giop21` and `leader6` (see the length of the error paths obtained by A*, which are optimal).

The length of the error paths is sometimes increased when ACOhg^{POR} is used. One reason for this is that, in general, the reduction in the construction graph performed by POR does not maintain the optimal paths and, thus,

the optimal error path in the reduced model can be longer than the one of the original model. This is a well-known effect of POR that can be clearly observed in the `leader`*i* models in Table 2.

4 Conclusions

In this deliverable we have combined ACOhg with partial order reduction for solving the problem of searching for safety properties violations in concurrent models. We used scalable models for the experiments in order to check if the use of POR is beneficial for all the model sizes. From the results shown in this document we conclude that ACOhg combined with POR reduces the computational effort. This kind of combination seems to be a promising research line in searching for errors in very large models and real software.

References

[1] Dragan Bošnački, Stefan Leue, and Alberto Lluch-Lafuente. Partial-order reduction for general state exploring algorithms. In *SPIN 2006*, volume 3925 of *Lecture Notes in Computer Science*, pages 271–287, 2006.

[2] 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.

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

[4] 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.

[5] 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.