Málaga, 22 de Noviembre de 2008

Informe Ejecutivo

TÍTULO: ACO-1.1-2008: Mejora de ACOhg-live usando componentes fuertemente conexas

RESUMEN: En este documento presentamos una mejora para el algoritmo ACOhg-live. La mejora consiste en usar la clasificación de las componentes fuertemente conexas del autómata que representa la negación de la fórmula LTL para aumentar la eficiencia y la eficacia de la búsqueda. Los resultados muestran que el coste computacional de la mejora es despreciable y las ventajas de la misma pueden ser importantes.

OBJETIVOS:

1. Presentar una mejora para el algoritmo ACOhg-live basado en componentes fuertemente conexas.
2. Comparar los resultados de ACOhg-live con y sin la mejora mencionada.

CONCLUSIONES:

1. El uso de la mejora basada en componentes fuertemente conexas aumenta la tasa de éxito y reduce los recursos computacionales requeridos para la búsqueda.
2. La longitud de las trazas de error puede aumentar ligeramente dependiendo del modelo.

RELACIÓN CON ENTREGABLES:

PRE: SOFTW-1.0-2008 (lectura necesaria)

PRE: ACO-1.0-2008 (lectura necesaria)

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-1.1-2008: Improving ACOhg-live by Analyzing Strongly Connected Components

ABSTRACT: In this deliverable we present an improvement on the ACOhg-live algorithm. This improvement consists of using the strongly connected components of the automaton representing the negation of the LTL formula in order to increase the efficiency and efficacy of the search. The results show that the computational cost of the improvement is negligible and the advantages of applying it can be important.

GOALS:

1. Present an improvement for ACOhg-live based on strongly connected components.
2. Compare the results of ACOhg-live with and without the improvement.

CONCLUSIONS:

1. The use of the SCC improvement increases the hit rate and decreases the computational resources required for the search.
2. The length of error paths could be slightly increased depending on the particular model.

RELATION WITH DELIVERABLES:

PRE: SOFTW-1.0-2008 (mandatory reading)

PRE: ACO-1.0-2008 (mandatory reading)

Improving ACOhg-live by Analyzing Strongly Connected Components

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 SOFTW-1.0-2008 the motivation for using metaheuristic algorithms and ACO in particular for this problem. In deliverable ACO-1.0-2008 we present a proposal, ACOhg-live, for checking liveness properties. In this deliverable we present an improvement to ACOhg-live that takes into account the structure of the automaton representing the property to check. In particular, this improvement is based on the strongly connected components of the property automaton.

The deliverable is organized as follows. The next section background information on properties and automata. It also introduces the terminology we will use through the document. Section 3 describes the improvement we propose for ACOhg-live. In Section 4 we present some experimental results comparing ACOhg-live with and without the improvement. Finally, Section 5 outlines the conclusions. In this deliverable we do not describe ACOhg and ACOhg-live. For a description of both algorithms the reader should see deliverables SOFTW-1.0-2008 and ACO-1.0-2008.

2 Background

In this section we give some details on the way in which properties are checked in explicit state model checking. In particular, we will focus on the model checker HSF-SPIN [5], an experimental model checker by Edelkamp, Lluch-Lafuente and Leue based on the popular model checker SPIN[9]. First, we formally define the concept of *property* of a concurrent system. After that, we detail how the properties can be represented using automata and we define the concept of *strongly connected components*, which allows us to implement improvements on the search algorithms.

2.1 Properties

Let S be the set of *states* of a program (concurrent system), S^ω the set of infinite sequences of program states, and S^* the set of finite sequences of program states. The elements of S^ω are called *executions* and the elements of S^* are *partial executions*. However, (partial) executions are not necessarily real (partial) executions of the program. The real executions of the program form a subset of S^ω. A *property* P is a set of executions, $P \subseteq S^\omega$. We say that an execution $\sigma \in S^\omega$ *satisfies* the property P if $\sigma \in P$, and σ *violates* the property if $\sigma \notin P$. In the former case we use the notation $\sigma \vdash P$, and the latter case is denoted with $\sigma \nvdash P$. A property P is a *safety property* if for all executions σ that violate the property there exists a prefix σ_i (partial execution) such that all the extensions of σ_i violate the property. Formally,

$$\forall \sigma \in S^\omega : \sigma \nvdash \mathcal{P} \Rightarrow (\exists i \geq 0 : \forall \beta \in S^\omega : \sigma_i \beta \nvdash \mathcal{P}) \enspace , \tag{1}$$

where σ_i is the partial execution composed of the first i states of σ. Some examples of safety properties are the absence of deadlocks and the fulfilment of invariants. On the other hand, a property P is a *liveness property* if for all the partial executions α there exists at least one extension that satisfies the property, that is,

$$\forall \alpha \in S^* : \exists \beta \in S^\omega, \alpha\beta \vdash \mathcal{P} \enspace . \tag{2}$$

One example of liveness property is the absence of starvation. The only property that is a safety and liveness property at the same time is the trivial property $P = S^\omega$. It can be proved that any given property can be expressed as an intersection of a safety and a liveness property [1].

The properties of a concurrent system are usually specified using a temporal logic, like Linear Temporal Logic (LTL) or computation Tree logic (CTL). We use the former in this work. In this case, atomic propositions are defined on the base of the variables and program counters of the processes of the system. The property is expressed using an

LTL formula composed of the atomic propositions. For example, an LTL property can be $\Box p$ (read "henceforth p"), where $p \equiv x > 3$. This property specifies an invariant: for a concurrent system to fulfil this property the variable x must always be greater than 3. This property consists of all the executions σ in which the variable x is greater than 3 for all the states of σ.

2.2 Property Automaton and Checking

The way in which LTL properties are checked in HSF-SPIN is by means of an automaton that captures the violations of the property, that is, the automaton accepts the executions in which the property is violated. This automaton is called *never claim* in SPIN and HSF-SPIN. This never claim can be automatically computed from the negation of the LTL formula [7]. In order to find a violation of a given LTL property, HSF-SPIN (and SPIN) explores the synchronous product of the concurrent model and the never claim, also called Büchi automaton. As an illustration, in Fig. 1 we show the automaton of a simple concurrent system (left box), the never claim used to check the LTL formula $\Box(p \rightarrow \Diamond q)$ (which means that an occurrence of p is always followed by an occurrence of q, not necessarily in the next state), and the synchronous product of these two automata.

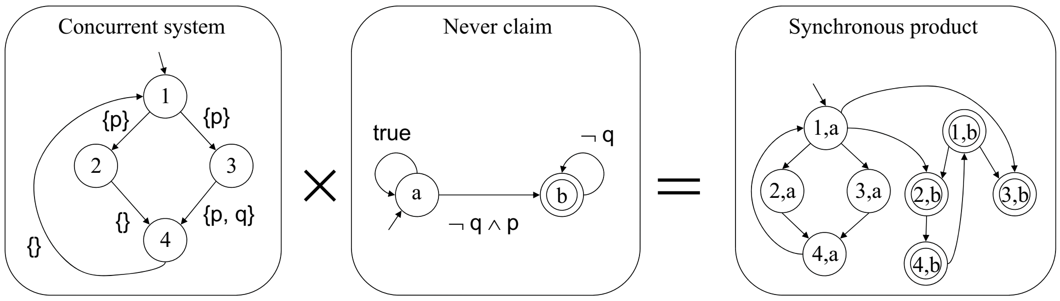

Figure 1: Synchronous product of a simple concurrent system and a never claim.

HSF-SPIN (and SPIN) searches in the Büchi automaton for an execution $\sigma = \alpha\beta^{\omega}$ composed of a partial execution $\alpha \in S^*$ and a cycle of states $\beta \in S^*$ containing an accepting state. If such an execution is found it violates the liveness component of the LTL formula and, thus, the whole LTL formula. During the search it can also be found a state in which the end state of the never claim is reached. This means that an execution has been found that violates the safety component of the LTL formula and the partial execution $\alpha \in S^*$ that leads the model to that state violates the LTL formula[1]. In HSF-SPIN and SPIN model checkers the search can be done using the Nested Depth First Search algorithm (NDFS) [8]. However, if the LTL formula represents a safety property (the liveness component is *true*) the problem of finding a property violation is reduced to find a partial execution $\alpha \in S^*$ violating the LTL formula, i.e., it is not required to find an additional cycle containing the accepting state. In this case classical graph exploration algorithms such as Breadth First Search (BFS), or Depth First Search (DFS) can be used for finding property violations. These classical algorithms cannot be used when we are searching for liveness property violations because they are not designed to find the cycle of states β above mentioned.

In order to improve the search for property violations it is possible to take into account the structure of the never claim. The idea is based on the fact that a cycle of states in the Büchi automaton entails a cycle in the never claim. For improving the search first we need to compute the *strongly connected components* (SCCs) of the never claim. A strongly connected subgraph $G = (V, A)$ of a directed graph is that in which for all pairs of different nodes $u, v \in V$ there exist two paths: one from u to v and another one from v to u. The strongly connected components of a directed graph are its maximal strongly connected subgraphs. Once we have the SCCs of the never claim we have to classify them into three categories depending on the accepting cycles they include. We denote with N-SCC those SCCs in which no cycle is accepting. A P-SCC is that in which there exists at least one accepting cycle and at least one non-accepting cycle. Finally, F-SCC are the SCCs in which all the cycles are accepting [6]. All the cycles found in the Büchi automaton have an associated cycle in one SCC of the never claim. Furthermore, if the cycle is accepting (which is the objective of the search) this SCC is necessarily a P-SCC or an F-SCC. The classification of the SCCs of the never claim can be used to improve the search for property violations. In particular, the accepting states in an N-SCC can be ignored, and the cycles found inside an F-SCC can be considered as accepting.

3 ACOhg-live improvement

We can make the search for liveness errors more efficient if we take into account the classification of the never claim SCCs. The improvements are localized in two places of ACOhg-live. During the first phase, in which accepting states are searched for, those accepting states that belong to an N-SCC in the never claim are ignored. The reason

[1]A deeper explanation of the foundations of the automata-based model checking can be found in [3] and [9].

is that an accepting state in an N-SCC cannot be part of an accepting cycle. This way, we reduce the number of accepting states to be explored in the second phase.

The second improvement is localized in the computation of the successors of a state (line 10 in Algorithm 1) in both, the first and the second phase of ACOhg-live. When the successors are computed, ACOhg checks if they are included in the path that the ant has traversed up to the moment. If they are, the state is not considered as the next node to visit since the ant would build a cycle. The improvement consists in checking if this cycle is in an F-SCC. This can be easily checked by finding if the state that closes the cycle is in an F-SCC of the never claim. If it is, then an accepting cycle has been found and the global search stops.

Algorithm 1 ACOhg

```
1: init = {initial_node};
2: next_init = $\emptyset$;
3: $\tau$ = initializePheromone();
4: step = 1;
5: stage = 1;
6: while step $\leq$ msteps do
7:   for k=1 to colsize do {Ant operations}
8:     $a^k = \emptyset$;
9:     $a^k_1$ = selectInitNodeRandomly (init);
10:    while $|a^k| < \lambda_{ant} \wedge T(a^k_*) - a^k \neq \emptyset \wedge a^k_* \notin O$ do
11:      node = selectSuccessor ($a^k_*$, $T(a^k_*)$, $\tau$,$\eta$);
12:      $a^k = a^k$ + node;
13:      $\tau$ = localPheromoneUpdate($\tau$,$\xi$,node);
14:    end while
15:    next_init = selectBestPaths(init, next_init, $a^k$);
16:    if $f(a^k) < f(a^{best})$ then
17:      $a^{best} = a^k$;
18:    end if
19:  end for
20:  $\tau$ = pheromoneEvaporation($\tau$, $\rho$);
21:  $\tau$ = pheromoneUpdate($\tau$, $a^{best}$);
22:  if step $\equiv$ 0 mod $\sigma_s$ then
23:    init = next_init;
24:    next_init = $\emptyset$;
25:    stage = stage+1;
26:    $\tau$ = pheromoneReset();
27:  end if
28:  step = step + 1;
29: end while
```

The advantages of these improvements depend on the structure of the LTL formula and the model to check. We can notice no advantages in some cases, especially when the number of N-SCC and F-SCC is small. However, the computational cost of the improvements is negligible, since it is possible to check the kind of SCC associated to a state in constant time.

4 Experiments

In this section we present some results obtained with our ACOhg-live algorithm. For the experiments we have selected five Promela models implementing faulty concurrent systems: `alter`, `giop`, `phi`, `elev` and `sgc`. All these models violate a liveness property that is specified in LTL. They can be found with the source code of ACOhg and HSF-SPIN in `http://oplink.lcc.uma.es/software`. See [2] for a description of the models.

The parameters used in the experiments for the ACOhg algorithms in the two phases of ACOhg-live are shown in Table 1. 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).

With respect to the heuristic information, we use the formula-based heuristic defined in [4] 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 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 4 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

Table 1: Parameters for 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

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.

We compare two versions of the ACOhg-live algorithm: one of them using the SCC improvement (called ACOhg-live$^+$ in the following) and the other one without that improvement (called ACOhg-live$^-$). With this experiment we want to analyze the influence on the results of the SCC improvement. All the properties checked in the experiments have at least one F-SCC in the never claim; none of them has a P-SCC; and all except `sgc` have exactly one N-SCC. In Table 2 we show the hit rate, the length of the error trails, the memory required (in Kilobytes), and the run time (in milliseconds) of the ACOhg-live algorithms. The average values of the 100 independent runs are shown in normal size while the standard deviation values are shown as subscript. We highlight with a grey background the best results (maximum values for hit rate and minimum values for the rest of the measures). We also show the results of a statistical test (with level of significance $\alpha = 0.05$) in order to check if there exist statistically significant differences (last column). A plus sign means that the difference is significant and a minus sign means that it is not. In the case of the hit rate we use a Westlake-Schuirmann test of equivalence of two independent proportions, for the rest of the measures we use a Kruskal-Wallis test [10].

Table 2: Influence of the SCC improvement

Models	**Meas.**	**ACOhg-live$^-$**		**ACOhg-live$^+$**		**T**
`alter`	Hit	100/100		100/100		-
	Len.	10.00	$_{0.00}$	15.82	$_{6.74}$	+
	Mem.	1929.00	$_{0.00}$	1929.00	$_{0.00}$	-
	CPU	241.80	$_{59.35}$	10.40	$_{3.98}$	+
`giop10`	Hit	84/100		89/100		-
	Len.	68.57	$_{5.29}$	67.60	$_{6.09}$	-
	Mem.	6375.90	$_{542.50}$	5098.75	$_{1580.90}$	+
	CPU	7816.55	$_{4779.41}$	935.84	$_{1009.74}$	+
`giop15`	Hit	46/100		57/100		-
	Len.	81.26	$_{3.64}$	78.30	$_{6.49}$	+
	Mem.	9001.17	$_{483.22}$	8538.54	$_{1610.63}$	-
	CPU	11725.65	$_{7307.22}$	2016.84	$_{1254.88}$	+
`giop20`	Hit	14/100		30/100		+
	Len.	93.29	$_{2.08}$	88.47	$_{4.72}$	+
	Mem.	11132.71	$_{894.26}$	10403.17	$_{1920.50}$	-
	CPU	11360.00	$_{4564.72}$	2575.33	$_{1103.46}$	+
`phi20`	Hit	98/100		97/100		-
	Len.	88.29	$_{6.91}$	108.73	$_{10.08}$	+
	Mem.	3398.63	$_{34.05}$	3385.04	$_{63.41}$	-
	CPU	5162.04	$_{645.64}$	851.75	$_{1462.71}$	+
`phi30`	Hit	94/100		95/100		-
	Len.	122.60	$_{9.58}$	139.15	$_{9.06}$	+
	Mem.	5146.62	$_{44.70}$	5148.12	$_{57.48}$	-
	CPU	10980.64	$_{2156.73}$	2701.79	$_{3876.34}$	+
`phi40`	Hit	77/100		81/100		-
	Len.	154.74	$_{9.74}$	166.83	$_{9.44}$	+
	Mem.	7573.68	$_{66.50}$	7545.35	$_{81.04}$	+
	CPU	20422.60	$_{5795.93}$	5807.41	$_{7588.17}$	+
`elev10`	Hit	100/100		100/100		-
	Len.	126.56	$_{18.32}$	127.76	$_{16.89}$	-
	Mem.	2617.60	$_{7.93}$	2617.04	$_{9.72}$	-
	CPU	2577.30	$_{2258.38}$	2372.90	$_{1963.04}$	-
`elev15`	Hit	100/100		100/100		-
	Len.	182.02	$_{9.75}$	180.04	$_{16.83}$	-
	Mem.	3163.56	$_{10.98}$	3164.64	$_{13.44}$	-
	CPU	2683.00	$_{3274.20}$	2812.10	$_{3540.73}$	-
`elev20`	Hit	100/100		100/100		-
	Len.	233.00	$_{0.00}$	231.62	$_{13.73}$	-
	Mem.	3716.44	$_{13.15}$	3716.92	$_{11.29}$	-
	CPU	3900.60	$_{7141.02}$	3034.00	$_{4709.07}$	-
`sgc`	Hit	32/100		100/100		+
	Len.	24.00	$_{0.00}$	24.00	$_{0.00}$	-
	Mem.	2699.00	$_{23.13}$	2285.00	$_{0.00}$	+
	CPU	575191.88	$_{62021.86}$	710.20	$_{48.58}$	+

With respect to the hit rate both algorithms obtain the maximum value (100%) in the models `alter` and `elev`j.

In `giop`*j* ACOhg-live$^+$ obtains higher hit rate than ACOhg-live$^-$ and the difference increases with the model size. In `phi`*j* the hit rate is similar in both algorithms (not statistically significant). In `sgc`, ACOhg-live$^+$ obtains the maximum hit rate while ACOhg-live$^-$ is only able to find an error in 32% of the executions. Thus, the first conclusion of the experiment is that ACOhg-live$^+$ is able to obtain higher hit rates than ACOhg-live$^-$.

Regarding the quality of the solutions, we observe in Table 2 that the length of the error paths obtained with ACOhg-live$^-$ is shorter with statistical confidence than the length of the ones obtained with ACOhg-live$^+$ in `alter` and `phi`*j*. On the other hand, the lengths of the error paths obtained by ACOhg-live$^+$ are shorter than the ones obtained by ACOhg-live$^-$ with statistical confidence in `giop`15 and `giop`20. In the rest of the models there is no statistically significant difference.

Finally, concerning the computational resources (memory and time required for finding an error path) we can observe that ACOhg-live$^+$ outperforms the results of ACOhg-live$^-$. All the statistically significant differences in memory and CPU time support this claim. This is the expected result, since the cost of the SCC improvement in terms of memory and computation time is negligible and, in addition, the search is more effective.

5 Conclusions

In summary, we can conclude that the use of the SCC improvement increases the hit rate and decreases the computational resources required for the search. The length of error paths could be slightly increased depending on the particular model.

References

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

[2] 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.

[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. Directed Explicit Model Checking with HSF-SPIN. In *Lecture Notes in Computer Science, 2057*, pages 57–79. Springer, 2001.

[5] 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.

[6] 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.

[7] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of IFIP/WG6.1 Symposium on Protocol Specification, Testing, and Verification (PSTV95)*, pages 3–18, Warsaw, Poland, June 1995.

[8] 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.

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

[10] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 2007.