Ant Colony Optimization in Model Checking

Francisco Chicano and Enrique Alba

University of Málaga, Spain,
{chicano, eat}@lcc.uma.es

Abstract. Most of model checkers found in the literature use exact deterministic algorithms to check the properties. The memory required for the verification with these algorithms usually grows in an exponential way with the size of the system to verify. 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. In this work we summarize our observations after the application of Ant Colony Optimization to find property violations in concurrent systems using a explicit state model checker. The experimental studies show that ACO finds optimal or near optimal error trails in faulty concurrent systems with a reduced amount of resources, outperforming in most cases the results of algorithms that are widely used in model checking, like Nested Depth First Search. This fact makes ACO suitable for checking properties in large faulty concurrent programs, in which traditional techniques fail to find counterexamples because of the model size.

1 Introduction

Model checking [7] is a fully automatic technique that allows to check if a given concurrent system satisfies a property like, for example, the absence of deadlocks, the absence of starvation, the fulfilment of an invariant, etc. The use of this technique is a must when developing software that controls critical systems, such as an airplane or a spacecraft. However, the memory required for the verification usually grows in an exponential way with the size of the system to verify. This fact is known as the *state explosion problem* and limits the size of the system that a model checker can verify.

When the search for errors with a low amount of computational resources (memory and time) is a priority (e.g., in the first stages of the development), 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 [3]. They are search algorithms used in optimization problems that can find good quality solutions in a reasonable time. In this work we summarize the approaches used for the application of one metaheuristic, Ant Colony Optimization (ACO), to the problem of finding property violations in concurrent systems. We also show the results of some experimental studies analyzing the performance of the different approaches.

The paper is organized as follows. The next section presents the background information. Section 3 describes our algorithmic proposals. In Section 4 we present some experimental studies to analyze the performance of our proposals. We also compare our proposals against the most popular algorithms utilized in model checking. Finally, Section 5 outlines the conclusions and future work.

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 [9], an experimental model checker by Edelkamp, Lluch-Lafuente and Leue based on the popular model checker SPIN [12]. First, we formally define the concept of *property* of a concurrent system and we detail how the properties are checked. Then, we define the concepts of *strongly connected components* (SCC), *partial order reduction* (POR) and the use of heuristic information.

2.1 Properties and Checking

Let S be the set of possible *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 set of real executions of the program, denoted by M, is a subset of S^ω, that is, $M \subseteq S^\omega$. A *property* P is also 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,

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

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,

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

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 [2].

In explicit state model checking the concurrent system M and the property P are represented by finite state ω-automata, $\mathcal{A}(M)$ and $\mathcal{A}(P)$ respectively, that accept those executions they contain. In HSF-SPIN (and SPIN) the automaton $\overline{\mathcal{A}(P)}$, which captures the violations of the property, is called *never claim*. In

order to find a violation of a given property, HSF-SPIN explores the intersection (or synchronous product) of the concurrent model and the *never claim*, $\mathcal{A}(M) \cap \mathcal{A}(P)$, also called Büchi automaton. HSF-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 property and, thus, the whole property. During the search, it is also possible to find a state in which the end state of the *never claim* is reached (if any). This means that an execution has been found that violates the safety component of the property and the partial execution $\alpha \in S^*$ that leads the model to that state violates the property.

Nested Depth First Search algorithm (NDFS) [11] is the most popular algorithm for performing the search. However, if the property is a safety one (the liveness component is *true*) the problem of finding a property violation is reduced to find a partial execution $\alpha \in S^*$, 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.

2.2 Strongly Connected Components

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* (and in the concurrent system). For improving the search first we need to compute the *strongly connected components* (SCCs) of the *never claim*. Then, we classify the SCCs into three categories depending on the accepting cycles they include. By an N-SCC, we denote an SCC in which no cycle is accepting. A P-SCC is an SCC in which there exists at least one accepting cycle and at least one non-accepting cycle. Finally, a F-SCC is an SCC in which all the cycles are accepting [10].

All the cycles found in the Büchi automaton have an associated cycle in the *never claim*, and, according to the definition of SCC, this cycle is included 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.

2.3 Partial Order Reduction

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 systems imposes an arbitrary ordering between concurrent events. When the automaton of the concurrent 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 automaton).

We use here a POR proposal based on *ample sets*. The main idea of ample sets is to explore only a subset of the enabled transitions of each state 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.

2.4 Using Heuristic Information

In order to guide the search to the accepting states, a heuristic value is associated to each state of the transition graph of the model. Different kinds of heuristic functions have been defined in the past to better guide exhaustive algorithms. Formula-based heuristics, for example, are based on the expression of the LTL formula checked [9]. Using the logic expression that must be false in an accepting state, these heuristics estimate the number of transitions required to get such an accepting state from the current one. Given a logic formula φ, the heuristic function for that formula H_φ is defined using its subformulae. In this work we use a formula-based heuristic that is defined in [9].

There is another group of heuristic functions called state-based heuristics that can be used when the objective state is known. From this group we can highlight the distance of finite state machines H_{fsm}, in which the heuristic value is computed as the sum of the minimum number of transitions required to reach the objective state from the current one in the local automaton of each process.

3 Algorithmic proposals

In order to find property violations in concurrent systems we proposed in the past an algorithm that we call ACOhg, a new variant of ACO [1]. This algorithm can be used when the property to check is a safety property. In the case of liveness properties we use a different algorithm, called ACOhg-live, that contains ACOhg as a component. We describe ACOhg in the next section and ACOhg-live in Section 3.2.

3.1 ACOhg algorithm

The objective of ACOhg is to find a path from the initial node to one objective node from a set O in a very large exploration graph. We denote with f a function that maps the paths of the graph into real numbers. This function must be designed to reach minimum values when the shortest path to an objective node is found. ACOhg minimizes this objective function. In Algorithm 1 we show the pseudocode of ACOhg.

The algorithm works as follows. At the beginning, the variables are initialized (lines 1-5). All the pheromone trails are initialized with the same value: a random number between τ_0^{min} and τ_0^{max}. In the `init` set (initial nodes for the ants construction), a starting path with only the initial node is inserted (line 1). This way, all the ants of the first stage begin the construction of their path at the initial node.

After the initialization, the algorithm enters in a loop that is executed until a given maximum number of steps (*msteps*) set by the user is performed (line 6). In a loop, each ant builds a path starting in the final node of a previous path

Algorithm 1 ACOhg

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

(line 9). This path is randomly selected from the `init` set. For the construction of the path, the ants enter a loop (lines 10-14) in which each ant k stochastically selects the next node according to the pheromone (τ_{ij}) and the heuristic value (η_{ij}) associated to each arc (a^k_*, j) with $j \in T(a^k_*)$ (line 11). The expression used is the standard random proportional rule used in ACOs [8].

After the movement of an ant from a node to the next one the pheromone trail associated to the arc traversed is updated as in Ant Colony Systems (ACS) [8] using the expression $\tau_{ij} \leftarrow (1-\xi)\tau_{ij}$ (line 13) where ξ, with $0 < \xi < 1$, controls the evaporation of the pheromone during the construction phase. This mechanism increases the exploration of the algorithm, since it reduces the probability that an ant follows the path of a previous ant in the same step. The construction process is iterated until the ant reaches the maximum length λ_{ant}, it finds an objective node, or all the successors of the last node of the current path, $T(a^k_*)$, have been visited by the ant during the construction phase. This last condition prevents the ants from constructing cycles in their paths.

After the construction phase, the ant is used to update the `next_init` set (line 15), which will be the `init` set in the next stage. In `next_init`, only starting paths are allowed and all the paths must have different last nodes. This rule is ensured by `selectBestPaths`. The cardinality of `next_init` is bounded by a given parameter ι. When this limit is reached and a new path must be included in the set, the starting path with higher objective value is removed from the set.

When all the ants have built their paths, a pheromone update phase is performed. First, all the pheromone trails are reduced according to the expression $\tau_{ij} \leftarrow (1-\rho)\tau_{ij}$ (line 20), where ρ is the *pheromone evaporation rate* and it holds that $0 < \rho \leq 1$. Then, the pheromone trails associated to the arcs traversed by the best-so-far ant (a^{best}) are increased using the expression $\tau_{ij} \leftarrow \tau_{ij} + 1/f(a^{best})$, $\forall (i,j) \in a^{best}$ (line 21). This way, the best path found is awarded with an extra amount of pheromone and the ants will follow that path with higher probability in the next step. We use here the mechanism introduced in Max-Min Ant Systems ($\mathcal{MM}$AS) [8] for keeping the value of pheromone trails in a given interval $[\tau_{min}, \tau_{max}]$ in order to maintain the probability of selecting one node above a given threshold. The values of the trail limits are $\tau_{max} = 1/\rho f(a^{best})$ and $\tau_{min} = \tau_{max}/a$ where the parameter a controls the size of the interval.

Finally, with a frequency of σ_s steps, a new stage starts. The `init` set is replaced by `next_init` and all the pheromone trails are removed from memory (lines 22-27). In addition to the pheromone trails, the arcs to which the removed pheromone trails are associated are also discarded (unless they also belong to a path in *next_init*). This removing step allows the algorithm to reduce the amount of memory required to a minimum value. This minimum amount of memory is the one utilized for storing the best paths found in one stage (the *next_init* set).

3.2 ACOhg-live

In this section we present ACOhg-live, an algorithm based on ACOhg for searching for general property violations in concurrent systems. In Algorithm 2 we show a high level object oriented pseudocode of ACOhg-live. We assume that `acohg1` and `acohg2` are two instances of a class implementing ACOhg.

The search that ACOhg-live performs is composed of two different phases. In the first one, ACOhg is used for finding accepting states in the Büchi automaton (line 2 in Algorithm 2). In this phase, the search of ACOhg starts in the initial node of the graph q and the set of objective nodes O is empty. That is, although the algorithm searches for accepting states, there is no preference on a specific set of them. If the algorithm finds accepting states, 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. That is, the initial node of the search and the only objective node are the same: the accepting state. 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 it will not be 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).

The algorithm can also stop its search due to another reason: an end state has been found. That is, when an end state is found either in the first or the second phase of the search the algorithm stops and returns the path from the initial

Algorithm 2 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;
```

state to that end state. If this happens, an execution of the concurrent system has been found that violates the safety component of the checked property.

When the property to check is the absence of deadlocks only the first phase of the search is required. In this case, ACOhg-live searches for deadlock states (states with no successors) instead of accepting states. When a deadlock state is found the algorithm stops returning the path from the initial state to that deadlock state. The second phase of the search, the objective of which is to find an accepting cycle, is never run in this situation.

Now we are going to give the details of the ACOhg algorithms used inside ACOhg-live. First of all, we use a node-based pheromone model, that is, the pheromone trails are associated to the nodes instead of the arcs. This means that all the values τ_{xj} associated to the arcs which head is node j are in fact the same value and is associated to node j. The heuristic values η_{ij} are defined after the heuristic function H using the expression $\eta_{ij} = 1/(1 + H(j))$. This way, η_{ij} increases when $H(j)$ decreases (high preference to explore node j).

Finally, the objective function f to be minimized is defined as

$$f(a^k) = \begin{cases} |\pi + a^k| \text{ if } a^k_* \in O \\ |\pi + a^k| + H(a^k_*) + p_p + p_c \frac{\lambda_{ant} - |a^k|}{\lambda_{ant} - 1} \text{ if } a^k_* \notin O \end{cases} , \tag{3}$$

where π is the starting path in `init` whose last node is the first one of a^k, p_p, and p_c are penalty values that are added when the ant does not end in an objective node and when a^k contains a cycle, respectively. The last term in the second row of Eq. (3) makes the penalty higher in shorter cycles (see [4] for more details).

4 Experimental studies

In this section we present some experimental studies aimed at analyzing the performance of our ACO proposals for the problem of finding property violations in concurrent systems. In the following section we present the Promela models used in the experimentation. Then we show the results of four different analyses: two of them related to the violation of safety properties and the other two related to liveness properties. In all the cases, 100 independent runs of the ACOhg algorithms are performed and the average and the standard deviation are shown.

4.1 Models

In the empirical studies we used nine Promela models, some of them scalable. In Table 1 we present the models with some information about them. They can be found in `oplink.lcc.uma.es` together with the HSF-SPIN and ACOhg source code. In the table we also show the safety and liveness properties that we check in the models.

Table 1. Promela models used in the experiments

Model	LoC	Processes	Safety property	Liveness property
phi$j$	57	$j+1$	deadlock	$\Box(p \rightarrow \Diamond q)$
giopi, j	740	$i+3(j+1)$	deadlock	$\Box(p \rightarrow \Diamond q)$
marriers$j$	142	$j+1$	deadlock	
leader$j$	178	$j+1$	assertion	
needham	260	4	LTL formula	
pots	453	8	deadlock	
alter	64	2		$\Box(p \rightarrow \Diamond q) \wedge \Box(r \rightarrow \Diamond s)$
elev$j$	191	$j+3$		$\Box(p \rightarrow \Diamond q)$
sgc	1001	20		$\Diamond p$

4.2 Safety properties

In this section we compare the results obtained with ACOhg for safety properties against the ones obtained with exact algorithms previously found in the literature. These algorithms are Breadth First Search (BFS), Depth First Search (DFS), A*, and Best First Search (BF). BFS and DFS do not use heuristic information while the other two do. In order to make a fair comparison we use two different ACOhg algorithms: one not using heuristic information (ACOhg-b) and another one using it (ACOhg-h). We show the results of all the algorithms in Table 2. In the table we can see the hit rate (number of executions that got an error trail), the length of the error trails found (number of states), the memory required (in Kilobytes), and the CPU time used (in milliseconds) by each algorithm. We highlight with a grey background the best results (maximum values for hit rate and minimum values for the rest of the measures). For ACOhg-b and ACOhg-h we omit here the standard deviation due to room problems. The parameters used in the ACOhg algorithms are the ones of [1].

In general terms, we can state that ACOhg-b is a robust algorithm that is able to find errors in all the proposed models with a low amount of memory. In addition, it combines the two good features of BFS and DFS: it obtains short error trails, like BFS, while at the same time requires a reduced CPU time, like DFS. Regarding the algorithms using heuristic information, we can state that ACOhg-h is the best trade-off between solution quality and memory required: it obtains almost optimal solutions with a reduced amount of memory.

4.3 Influence of POR

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. In Table 3 we present the results of applying ACOhg and ACOhg^{POR} to nine models: three instances of `giop`, `marriers`, and `leader`. The hit rate is always 100 %, and for this reason we omit it. In order to clarify that

Table 2. Results of ACOhg-b and ACOhg-h against the exhaustive algorithms.

Model	Measure	BFS	DFS	ACOhg-b	A*	BF	ACOhg-h
giop2,2	Hit rate	0/1	1/1	100/100	1/1	1/1	100/100
	Length	-	112.00	45.80	44.00	44.00	44.20
	Mem. (KB)	-	3945.00	4814.12	417792.00	2873.00	4482.12
	Time (ms)	-	30.00	113.60	46440.00	10.00	112.40
marriers4	Hit rate	0/1	0/1	57/100	0/1	1/1	84/100
	Length	-	-	92.18	-	108.00	86.65
	Mem. (KB)	-	-	5917.91	-	41980.00	5811.43
	Time (ms)	-	-	257.19	-	190.00	233.33
needham	Hit rate	1/1	1/1	100/100	1/1	1/1	100/100
	Length	5.00	11.00	6.39	5.00	10.00	6.12
	Mem. (KB)	23552.00	62464.00	5026.36	19456.00	4149.00	4865.40
	Time (ms)	1110.00	18880.00	262.00	810.00	20.00	229.50
phi16	Hit rate	0/1	0/1	100/100	1/1	1/1	100/100
	Length	-	-	31.44	17.00	81.00	23.08
	Mem. (KB)	-	-	10905.60	2881.00	10240.00	10680.32
	Time (ms)	-	-	289.40	10.00	40.00	243.80
pots	Hit rate	1/1	1/1	49/100	1/1	1/1	99/100
	Length	5.00	14.00	5.73	5.00	7.00	5.44
	Mem. (KB)	57344.00	12288.00	9304.67	57344.00	6389.00	6974.56
	Time (ms)	4190.00	140.00	441.63	6640.00	50.00	319.49

the reduced amount of memory required by the ACOhg algorithms is not due to the use of the heuristic information, we also show the results obtained with A* 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 ACOhg itself.

Table 3. Comparison among ACOhg, ACOhg^{POR} and A*.

Model	Measure	ACOhg		ACOhg^{POR}		A*
giop2,1	Length	42.30	1.71	42.10	0.99	42.00
	Mem. (KB)	3428.44	134.95	2979.48	98.33	27648.00
	Time (ms)	202.00	9.06	162.50	5.55	1000.00
giop4,1	Length	70.21	7.56	59.76	5.79	-
	Mem. (KB)	9523.67	331.76	7420.08	422.94	-
	Time (ms)	354.50	42.39	264.90	40.46	-
giop6,1	Length	67.59	13.43	61.74	3.16	-
	Mem. (KB)	11970.56	473.59	11591.68	477.67	-
	Time (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
	Time (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	-
	Time (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	-
	Time (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	-
	Time (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	-
	Time (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	-
	Time (ms)	49446.30	7557.40	8174.00	707.71	-

From the results in the table we conclude that the memory required by ACOhg^{POR} is always smaller than the one required by ACOhg. The length of the error paths is smaller for ACOhg^{POR} in six out of the nine models. Finally, the CPU time required by ACOhg^{POR} is up to 6.8 times lower (in `marriers10`) than the time required by ACOhg. Although it is not our objective to optimize the length of the error paths in this work, we can say that, in six out of the nine models, the length of the error paths obtained by ACOhg^{POR} is shorter than the one obtained by ACOhg. We finally also remind that other popular algorithm like A* cannot even be applied to most of these instances (only `giop2,1` and `leader6` can be tackled with A*), and thus we are investigating in a new frontier of high dimension models usually not found in literature.

4.4 Liveness Results

In the next experiment we compare the results obtained with ACOhg-live against the classical algorithm utilized for finding liveness errors in concurrent systems: Nested-DFS. This last algorithm is deterministic and for this reason we only perform one single run. In Table 4 we show the results of both algorithms. 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. For more details on the experiments see [5].

Table 4. Comparison between ACOhg-live and Nested-DFS

Model	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	+
`giop2,2`	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	+
`giop6,2`	Hit rate	100/100		0/1	+
	Length	58.77	7.21	•	•
	Mem. (KB)	5588.04	631.36	•	•
	Time (ms)	8733.50	3304.90	•	•
`giop10,2`	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	-

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 `giop6,2` and `giop10,2` because it requires more than the memory available in the machine used for the experiments (512 MB). 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). 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 `giop6,2` and `giop10,2` in which Nested-DFS requires more than 512 MB of memory while ACOhg-live obtains error paths with 38 MB at most. 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.

4.5 Influence of the SCC improvement

In this final study 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 5 we show the results. For more details on the experiments see [6].

Table 5. Influence of the SCC improvement

Model	Measure	ACohg-live$^-$		ACOhg-live$^+$		T	Model	ACohg-live$^-$		ACOhg-live$^+$		T
giop10,2	Hit rate	84/100		89/100		-	elev10	100/100		100/100		-
	Length	68.57	5.29	67.60	6.09	-		126.56	18.32	127.76	16.89	-
	Mem. (KB)	6375.90	542.50	5098.75	1580.90	+		2617.60	7.93	2617.04	9.72	-
	Time (ms)	7816.55	4779.41	935.84	1009.74	+		2577.30	2258.38	2372.90	1963.04	-
giop15,2	Hit rate	46/100		57/100		-	elev15	100/100		100/100		-
	Length	81.26	3.64	78.30	6.49	+		182.02	9.75	180.04	16.83	-
	Mem. (KB)	9001.17	483.22	8538.54	1610.63	-		3163.56	10.98	3164.64	13.44	-
	Time (ms)	11725.65	7307.22	2016.84	1254.88	+		2683.00	3274.20	2812.10	3540.73	-
giop20,2	Hit rate	14/100		30/100		+	elev20	100/100		100/100		-
	Length	93.29	2.08	88.47	4.72	+		233.00	0.00	231.62	13.73	-
	Mem. (KB)	11132.71	894.26	10403.17	1920.50	-		3716.44	13.15	3716.92	11.29	-
	Time (ms)	11360.00	4564.72	2575.33	1103.46	+		3900.60	7141.02	3034.00	4709.07	-
phi20	Hit rate	98/100		97/100		-	alter	100/100		100/100		-
	Length	88.29	6.91	108.73	10.08	+		10.00	0.00	15.82	6.74	+
	Mem. (KB)	3398.63	34.05	3385.04	63.41	-		1929.00	0.00	1929.00	0.00	-
	Time (ms)	5162.04	645.64	851.75	1462.71	+		241.80	59.35	10.40	3.98	+
phi30	Hit rate	94/100		95/100		-	sgc	32/100		100/100		+
	Length	122.60	9.58	139.15	9.06	+		24.00	0.00	24.00	0.00	-
	Mem. (KB)	5146.62	44.70	5148.12	57.48	-		2699.00	23.13	2285.00	0.00	+
	Time (ms)	10980.64	2156.73	2701.79	3876.34	+		575191.88	62021.86	710.20	48.58	+
phi40	Hit rate	77/100		81/100		-						
	Length	154.74	9.74	166.83	9.44	+						
	Mem. (KB)	7573.68	66.50	7545.35	81.04	+						
	Time (ms)	20422.60	5795.93	5807.41	7588.17	+						

From the results in Table 5, we 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.

5 Conclusions and Future Work

In this paper we summarize our observations using ACO algorithms for the problem of searching for property violations in concurrent systems. The numerical results shown here are an excerpt from the research work performed during the last two years on this topic. From the results we conclude that ACO algorithms are promising for the model checking domain. They can find short error trails using a low amount of computational resources.

At present, we are investigating how other metaheuristic algorithms perform on this problem. We are also working on new heuristic functions that can guide the search in a better way. As future work, we plan to design and develop new models of parallel ACO algorithms in order to profit from the computational power of a cluster or a grid of computers.

6 Acknowledgements

This work has been partially funded by the Spanish Ministry of Science and Innovation and FEDER under contract TIN2008-06491-C04-01 (the M* project). It has also been partially funded by the Andalusian Government under contract P07-TIC-03044 (DIRICOM project).

References

1. Enrique Alba and Francisco Chicano. Finding safety errors with ACO. In *Proc. of GECCO*, pages 1066–1073, 2007.
2. Bowen Alpern and Fred B. Schneider. Defining liveness. *Inform. Proc. Letters*, 21:181–185, 1985.
3. C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.
4. 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.
5. Francisco Chicano and Enrique Alba. Finding liveness errors with ACO. In *Proceedings of the World Conference on Computational Intelligence*, pages 3002–3009, Hong Kong, China, 2008.
6. Francisco Chicano and Enrique Alba. Searching for liveness property violations in concurrent systems with ACO. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 1727–1734, Atlanta, USA, 2008.
7. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000.
8. Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. The MIT Press, 2004.
9. Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Protocol Verification with Heuristic Search. In *AAAI-Spring Symposium on Model-based Validation Intelligence*, pages 75–83, 2001.
10. Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *Intl. Jnl. of Soft. Tools for Tech. Transfer*, 5:247–267, 2004.
11. G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32, 1996.
12. Gerald J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.

Combinación de distribuciones de probabilidad con AHP

Joseba Esteban López , José Javier Dolado
Departamento de Lenguajes y Sistemas
Universidad del País Vasco U.P.V./E.H.U.
jose.esteban@ehu.es, dolado@si.ehu.es

No Institute Given

Resumen Dado el creciente aumento del uso de las redes Bayesianas en el área de la ingeniaría del software y que dichas redes trabajan con distribuciones de probabilidad tanto discretas como continuas, creemos interesante conocer como combinar varias de estas distribuciones. En este artículo se presenta el método matemático *combinación lineal de opiniones* para aunar distintas distribuciones de probabilidad. El único inconveniente que presenta este sencillo método, consiste en establecer los pesos de cada variable aleatoria. Combinando dicho método con el método de ayuda a toma de decisiones *Analytic Hierarchy Process*, podemos establecer los pesos de cada variable. En este artículo se exponen dos planteamientos en la combinación de distribuciones de probabilidad: combinar varias estimaciones de expertos para obtener una estimación más fiable, y estimar una variable global a partir de la estimación de las partes que lo componen.

1. Introducción

Las redes Bayesianas son cada vez más populares dentro de la ingeniería, la inteligencia artificial y la estadística. En la ingeniería del software se han utilizado en diferentes áreas como la estimación del esfuerzo y la calidad o en pruebas de software. Las redes Bayesianas son modelos gráficos probabilísticos utilizados en la toma de decisiones [1]. Dichas redes trabajan con variables aleatorias. Una variable aleatoria es una variable que puede tomar un valor numérico determinado por el resultado del experimento aleatorio. Es decir, sólo puede tomar un conjunto de valores concretos. Las variables aleatorias se representan mediante distribuciones de probabilidad. La distribución de probabilidad de una variable aleatoria establece el rango y la probabilidad de los valores que puede tomar la variable. Estas distribuciones de probabilidad pueden ser continuas o discretas determinando, de esta forma, el tipo de variable aleatoria. Se denomina variable continua a aquella que puede tomar cualquiera de los infinitos valores existentes dentro de un intervalo. En el caso de variable continua la distribución de probabilidad es la integral de la función de densidad. Por el contrario, se denomina distribución de variable discreta a aquella cuya función de probabilidad sólo toma valores positivos en un conjunto de valores finito o infinito numerable y se denomina función de masa de probabilidad.

Las redes Bayesianas se componen de dos partes. Una parte cualitativa que consiste en una estructura gráfica formada por nodos (variables aleatorias) y dependencias entre nodos. Y otra parte cuantitativa correspondiente a las tablas de probabilidad de los nodos. Estas tablas de probabilidad se pueden obtener a partir de bases de datos (en el caso de la ingeniería del software, de proyectos ya finalizados) o de una manera subjetiva, utilizando creencias de expertos en el domino [2] **Referencia Dolado 2007**.

En el caso de basarnos en el juicio experto para obtener las distribuciones de probabilidad de cada variable, es aconsejable apoyarse en varios expertos con el fin de minimizar posibles errores. Consultar a varios expertos puede verse como una versión subjetiva de aumentar la muestra en un experimento o de incrementar la información de base, es decir, hacerlo más fiable. Esto implica que los expertos deben resolver sus diferencias en cuanto a la definición de cada variable y ponerse de acuerdo en qué es lo que se quiere estimar. Conseguir llegar a un consenso entre los diferentes expertos puede suponer un gran esfuerzo, por lo que es recomendable que cada experto se familiarice con el el domino del problema con el fin de que todos tengan una idea similar del problema [3].

Una vez están de acuerdo, cada experto debe proporcionar su estimación en forma de distribución de probabilidad. Incluso estando conformes con las definiciones de las variables, es posible que los expertos estén en desacuerdo con las probabilidades de dichas variables. Estas desavenencias pueden deberse al empleo de diferentes métodos de obtención de información, las variaciones de los conjuntos de información que utilizan los expertos o los diferentes enfoques filosóficos. En cualquier caso, si los expertos nunca estuvieran en desacuerdo, no tendría cabida consultar a más de un experto. Una vez tenemos las diferentes opiniones de los expertos en forma de distribuciones de probabilidad, hay que combinarlas [3].

La combinación de distribuciones de probabilidad no es un ámbito exclusivo de unificar estimaciones de diferentes expertos sobre la misma variable. También puede darse la necesidad de aunar distintas distribuciones de probabilidad para obtener una distribución que globalice el resto de distribuciones. Por ejemplo, se puede establecer el esfuerzo de desarrollo de un software como la suma del esfuerzo de las diferentes funcionalidades que lo componen. En este caso consistiría en combinar la estimación de esfuerzo de cada funcionalidad que compone un software, para obtener una estimación del esfuerzo total del mismo. Las variables que se quieren combinar deben ser del mismo tipo (esfuerzo en nuestro ejemplo).

El enfoque axiomático, que comentamos en el apartado 2, permite combinar las distribuciones de probabilidad. Para poder utilizar este enfoque hay que establecer pesos a cada estimación. Con este objetivo, nosotros proponemos el método de ayuda a la toma de decisiones Analytic Hierarchy Process (A.H.P.) que planteamos en el tercer y cuarto apartado de este artículo. Por último exponemos las conclusiones y trabajo futuro.

2. Método matemático para la combinación de distribuciones de probabilidad: Enfoque axiomático

Los métodos de agregación matemáticos se componen de procesos analíticos que operan sobre varias distribuciones de probabilidad individuales para obtener una única distribución de probabilidad combinada. Estos métodos matemáticos van desde simples cálculos de sumas, como la media aritmética o geométrica de las probabilidades, hasta procedimientos basados en enfoques axiomáticos [3]. En esta sección exponemos una técnica axiomática para la combinación de distribuciones de probabilidad: Combinación lineal de opiniones (*linear opinion pool*) en el apartado 2.1.

2.1. Combinación lineal de opiniones (*linear opinion pool*)

Este método sencillo de combinación de probabilidades se remonta a la época de Laplace y consiste en aplicar la formula 1, donde n es el número de expertos, $p_i(\Theta)$ representa la distribución del experto i-ésimo para la variable Θ, $p(\Theta)$ representa la distribución de probabilidad combinada, y w_i representa los pesos. Estos pesos han de sumar uno. Por simplificar, p representa la función de masa de en el caso de una distribución de probabilidad conjunta, y la función de densidad en el caso continuo [3].

$$p(\Theta) = \sum_{i=1}^{n} w_i p_i(\Theta) \tag{1}$$

En el caso de querer unificar distribuciones de probabilidad para obtener las distribución de probabilidad de una variable global, n sería el número de variables no globales a combinar, $p_i(\Theta)$ representaría la distribución de la variable i-ésima del mismo tipo de variables a combinar, y $p(\Theta)$ representa la distribución de probabilidad combinada.

Esta técnica es claramente una combinación lineal ponderada de las probabilidades de los expertos. Este método de combinación es fácil de calcular y de entender, además de satisfacer varios axiomas. Por ejemplo, satisface la propiedad de unanimidad (*unanimity*) que establece que si todos los expertos están de acuerdo en una probabilidad, entonces también estarán de acuerdo en la probabilidad combinada. Este método de combinación es el único que cumple la propiedad de marginación: supongamos de Θ es un vector de probabilidades y que nos interesa sólo un elemento de dicho vector, Θ_j, la propiedad de marginación establece que las probabilidades combinadas son las mismas tanto si combinamos las distribuciones marginales de Θ_j, como si combinamos las distribuciones de probabilidad conjunta Θ y después calculamos la distribución marginal de Θ_j [3].

2.2. Establecimiento de pesos en la combinación lineal de opiniones

Los pesos w_i pueden ser utilizados para representar, de algún modo, la calidad de los diferentes expertos o bien el grado de influencia de cada variable

a combinar sobre la variable global. En el caso de querer combinar diferentes estimaciones de distintos expertos sobre una misma variable, los pesos se establecerán de acuerdo a la calidad del experto en el domino como estimador. En el caso de necesitar aunar varias estimaciones para obtener una estimación global, los pesos determinan la influencia relativa de cada variable en la variable global. En el caso del ejemplo anterior de combinar el esfuerzo de cada funcionalidad, no tiene el mismo peso sobre el esfuerzo total una funcionalidad que gestione el acceso de los usuarios a la aplicación, que una funcionalidad que englobe la lógica de negocio de, por ejemplo, una aplicación de alquiler de vehículos vía web, la cual resultará bastante más costosa en términos de esfuerzo.

Si se considera a todos los expertos por igual o que todas las variables influyen de igual manera en una variable global, los pesos tendrán todos el mismo valor $1/n$, siendo n el número de expertos o la cantidad de variables respectivamente. En este caso, este método de combinación de distribuciones de probabilidad se convierte en una media aritmética. Se puede entender que un experto es "mejor"que otro (por ejemplo debido a que dispone de mejor información), esto implicará que el peso asociado a dicho experto tendrá un valor superior al del resto. La determinación de los pesos es una cuestión subjetiva y se pueden dar múltiples interpretaciones a los pesos.

Quizá la mayor complejidad que presenta el método de combinación lineal de opiniones sea el establecimiento de los pesos. Con este fin proponemos la utilización del método de ayuda a la toma de decisiones Analytic Hierarchy Process (A.H.P.) que exponemos en los siguientes apartados.

3. Analytic Hierarchy Process (A.H.P.)

Analytic Hierarchy Process (A.H.P.) es un método de estimación de ayuda a la toma de decisiones basado en múltiples criterios de decisión. AHP fue propuesto por Thomas L. Saaty en la década de los 80 [4]. Desde entonces se ha convertido en una de las técnicas más utilizadas para la toma de decisiones multiatributo. AHP se basa en juicios subjetivos realizados por los expertos. Los expertos aportan su conocimiento subjetivo, consistente en comparaciones entre las principales tareas que constituyen un proyecto software. Los expertos estiman, más que un valor exacto, una medida relativa. Basándose en esta idea, el experto, evaluando la proporción entre cada par de tareas definidas en la aplicación software, consigue una mayor exactitud en sus evaluaciones.

3.1. Algoritmo AHP

El algoritmo del método de estimación Analytic Hierarchy Process consta de cinco pasos que exponen a lo largo de este apartado.

En primer lugar se define el problema. Para esto hay que dividirlo en tres partes: objetivo, criterios y alternativas. El objetivo es la decisión que se ha de tomar. Los criterios representan los factores que afectan a la preferencia o deseabilidad de una alternativa. Pueden estar compuestos por otros criterios o

Definición	Explicación	Valor Relativo	Valor Recíproco
Mismo tamaño	Las dos entidades tienen aproximadamente el mismo tamaño	1	1.00
Ligeramente mayor (menor)	La experiencia o el juicio reconoce una entidad como algo más grande (más pequeño)	3	0.33
Mayor (menor)	La experiencia o el juicio reconoce una entidad como definitivamente más grande (más pequeño)	5	0.20
Mucho mayor (menor)	El dominio de una entidad sobre otra es evidente; un diferencia muy fuerte de tamaño	7	0.14
Extremadamente mayor (menor)	La diferencia entre las entidades comparadas es de un orden de magnitud	9	0.11
Valores intermedios entre puntos adyacentes de la escala	Cuando el compromiso es necesario	2, 4, 6, 8	0.5, 0.25, 0.16, 0.12

Cuadro 1. Escala verbal propuesta por Thomas L. Saaty

subcriterios. Las alternativas son las posibles opciones o acciones de las que se dispone y de las cuales se intenta elegir una. Una alternativa puede ser cualquier entidad relevante en un grupo de interés, como casos de uso, módulos software, objetos etc., es decir, cualquier entidad de la que se pueda conocer las magnitudes que se necesitan a la hora de tomar una decisión. Una vez hecho esto, se debe construir la jerarquía, de la que AHP toma el nombre.

Aunque no se trate de una parte esencial de la metodología de AHP, establecer una escala verbal o *verbal scale* agiliza el proceso de estimación y no hace peligrar la exactitud de la estimación. La escala verbal ayuda a entender cómo de menor es el término "menor que"ó cómo de mayor es el término "mayor que". Se compone de cuatro atributos: definición o etiqueta, explicación, valor relativo y valor recíproco. La escala verbal establece un consenso que evita que los expertos o los participantes en la estimación pierdan tiempo discutiendo sobre el grado de diferencia entre las alternativas comparadas. Thomas L. Saaty [5]nos propone una escala compuesta por nueve valores y sus recíprocos, como se puede ver en 1.

Con la jerarquía y la escala verbal ya bien definidas, se pasa a obtener la matriz de juicios o *judgement matrix*. Es en esta etapa donde entra en juego el juicio experto. El experto, basándose en la escala verbal, debe hacer comparaciones por parejas en cada nivel de la jerarquía, e ir anotándolos en la matriz. La matriz de juicios es de tamaño nxn, siendo n el número de alternativas de las que se dispone. Cada celda de la matriz de juicios contiene un valor a_{ij}, que representa el tamaño relativo de la entidad i respecto del de la entidad j. Los elementos de la matriz se definen como se muestra en (3.1). Si la entidad i es a_{ij} veces mayor (o menor) que la entidad j, entonces la entidad j es $1/a_{ij}$ veces menor (o mayor) que la entidad i. Teniendo en cuenta esta premisa y que la diagonal de la matriz sólo tiene como valor la unidad, no haría falta calcular todos los valores de la matriz. Sólo es necesario calcular una mitad de la matriz, ya sea la parte superior a la diagonal o la inferior.

$$A^{n\times n} = \begin{cases} a_{ij} = \frac{s_i}{s_j} & \text{Cómo de grande o pequeña es la entidad} i \text{respecto de la entidad} j \\ a_{ij} = 1 & \text{Las entidades} i y j \text{son de la misma proporción} \\ a_{ij} = \frac{1}{a_{ji}} & \text{Inversamente proporcionales} \end{cases} \tag{2}$$

A la hora de hacer las comparaciones por parejas, es necesaria la colaboración del experto y al menos una entidad de referencia o *reference task* de la que se conozca su magnitud real. Las proporciones de la entidad de referencia son las primeras que se han de situar en la matriz. Es importante que la proporción de esta entidad no ocupe los valores extremos de la escala verbal, sino que se sitúe más o menos hacia la mitad de la escala. De esta manera se minimizan los posibles prejuicios introducidos en la matriz de juicios. Otra posibilidad, con el mismo objetivo, radica en introducir más de una entidad de referencia repartidas uniformemente en la escala verbal.

Se debe elaborar una matriz de juicios por cada criterio a tener en cuenta en la toma de decisión. A continuación se calcula la escala de proporción o *ratio scale*. La escala de proporción es un vector r en el que cada posición del vector contiene un valor proporcional a la entidad i en relación al criterio elegido. Des esta forma tendremos un vector de proporción por cada criterio elegido. Asimismo, se debe componer una matriz de juicios comparando los criterios elegidos en la toma de decisión y, a continuación, calcular la escala de proporción entre los criterios. Este vector nos permite ponderar cada criterio y así conocer qué criterios afectan más a la decisión. También se calcula un índice de inconsistencia o *inconsistency index* por cada matriz. Este índice nos proporciona una medida de cómo de lejos está nuestra estimación de la consistencia perfecta. Una matriz de juicios perfectamente consistente es aquella en la que todos sus elementos satisfacen $a_{ij} \times a_{jk} = a_{ik}\ \forall i, j, k$.

Como procedimiento para calcular la escala de proporción y el índice de inconsistencia, proponemos la utilización del modelo propuesto por Eduardo Miranda en [5], por sencillez y buenos resultados. Hay que calcular la media geométrica v_i de cada fila de la matriz de juicios definida para un determinado criterio. El valor de v_i viene dado por (3). La escala de proporción, denominada vector de valores propios o *eigenvalue*, r, consiste en un vector en el que cada valor se calcula aplicando (4).

$$v_i = \sqrt[n]{\prod_{j=1}^{n} a_{ij}} \tag{3}$$

$$r = [r_1,\ r_2,\ \ldots\ , r_n]\ con\ r_i = \frac{v_i}{\sum_{j=1}^{n} v_j} \tag{4}$$

El índice de inconsistencia se puede calcular de la siguiente forma (5).

$$CI = \frac{\sqrt{\sum_{i=1}^{n} \sum_{j>i}^{n} \left(ln\ a_{ij} \ - \ ln\ \frac{v_i}{v_j} \right)^2}}{\frac{(n-1)\times(n-2)}{2}} \tag{5}$$

A partir del vector de valores propios o *eigenvalue* y el valor real de la entidad de referencia, se puede calcular el valor absoluto de cada entidad. Para ello se debe aplicar la expresión (6). Aplicando estos cálculos con el resto de criterios, previamente ordenados según el porcentaje de contribución sobre el proyecto en su totalidad, se puede calcular el valor de cada alternativa.

$$Valor_i = \frac{r_i}{r_{refrencia}} \times Valor_{refrencia} \tag{6}$$

En el siguiente apartado, se muestran las características más relevantes de AHP . También se indican los principales problemas que presenta este método de estimación.

3.2. Características de AHP

Analytic Hierarchy Process es uno de los métodos de estimación más sencillos cuya mayor dificultad radica en identificar los atributos y su contribución relativa. Además, proporciona una visión del proyecto software jerarquizada, estructurada y sistemática. Asimismo, AHP es poco propenso a errores, permitiendo estimaciones precisas con hasta un 40 % de comparaciones erróneas. A pesar de este dato no se puede afirmar que AHP sea mejor que la estimación experta, ya que está basado, precisamente, en comparaciones hechas por expertos entre pares de tareas.

AHP aporta una notable ventaja para los expertos, ya que resulta más sencillo hacer comparaciones por parejas (entre los pares de tareas) que estimar cada tarea de una en una. Por otro lado, el número de comparaciones que deben realizar puede suponer un problema. Esto se debe a que la relación de las comparaciones a realizar y el número de tareas es de orden cuadrático. Si nuestro proyecto se compusiera de n tareas, el número de comparaciones que se deberían realizar sería de $n(n-1)/2$. En el supuesto de manejar 30 tareas, se deberán realizar 435 comparaciones. Para evitar esto, aún a riesgo de aumentar la homogeneidad de la matriz de juicios, se pueden agrupar las tareas similares en grupos más reducidos.

En las fases iniciales del desarrollo de un proyecto software suele darse una carencia de datos de referencia. El método de estimación AHP resulta muy útil en estas etapas iniciales, ya que como mínimo necesita un único dato de referencia: la tarea de referencia. Esta característica nos posibilita hacer estimaciones bastante precisas en fases tempranas del desarrollo de un proyecto software. Es importante que el porcentaje de contribución de la tarea de referencia al proyecto global sea lo más preciso posible. Esto aumentará la exactitud de las estimaciones del resto de las tareas. En caso contrario, podrá derivar en errores. La exactitud de

las predicciones también se verá mejorada con el uso de más de una tarea de referencia.

Con el método de estimación AHP puede darse el fenómeno del Rank Reversal o alteración del rango. Este fenómeno consiste en un cambio en el ranking relativo de las tareas, al introducir una nueva tarea o eliminar una de ellas. Los autores Ying-Ming Wang y Taha M.S. Elhag en [6], exponen la existencia de varias propuestas enfocadas a evitar el fenómeno de la alteración en el rango. Estas propuestas tienen en cuenta si la alternativa introducida o eliminada del conjunto de alternativas seleccionadas agrega o no información. En ese mismo artículo se expone la propuesta de los autores, en la que se preserva el ranking de las alternativas sin necesidad de variar los pesos de las alternativas ni el número de criterios. Para esto, los autores proponen la normalización del vector de valores propios o *eigenvalue*.

Un aspecto a tener en cuenta utilizando el método de estimación AHP radica en la escala elegida, tanto por su proporción como por el número de puntos que la constituyen. En AHP el éxito en las estimaciones reside en la exactitud de las comparaciones entre las tareas. Estas comparaciones, son consecuencia directa de la escala de evaluación elegida, así como del número de puntos de la escala. La escala propuesta por Thomas L. Saaty en [4], propone el uso de una escala que varía entre 1/9 y 9, lo que supone un total de 17 puntos en la escala. En [7] los autores aconsejan el uso de una escala con un número de etiquetas más bajo, ya que esto nos facilitará el manejo de la escala. Eduardo Miranda en [5] también propone una escala diferente. Ésta consta de menos puntos que la propuesta por Saaty, y según el autor, es más fácil de utilizar por los expertos.

4. AHP como ponderador

El modo en el que AHP realiza las comparaciones entre las diferentes tareas nos ofrece un buen recurso para establecer los pesos. El procedimiento es más sencillo que el propio algoritmo de AHP. Básicamente es suficiente con construir la matriz de comparaciones y a continuación calcular el vector de valores propios. Se trata de comparar las variables o los expertos unos con otros utilizando la matriz de comparaciones. Estas comparaciones nos sirven para capturar el juicio experto, como ya se ha comentado. El experto deberá responder a una pregunta que permita establecer las diferentes influencias de las variables o de los expertos. La pregunta sería de la forma *¿cuánta más (menos) importancia tiene la variable v_i comparado con la variable v_j sobre la variable global?* o *¿cuánto más (menos) relevante es la opinión del experto p_i comparado con el experto p_j en este dominio?*.

Las preguntas están formuladas de manera que se compara la influencia de cada variable respecto a la variable global. Una vez realizada las comparaciones con ayuda de la escala verbal, obtenemos el vector de valores propios o de proporción. Este vector establece una proporción entre las diferentes variables comparadas acorde al criterio establecido en la pregunta formulada. La influencia de una variable sobre otra global se puede interpretar como el peso de dicha

variable sobre la variable global. De esta forma, el vector de valores propios se puede interpretar del mismo modo como un vector de pesos. Por tanto, el vector de valores propios nos proporciona directamente los pesos de cada variable.

A continuación presentamos dos ejemplos donde podemos ver la aplicación del método.Veremos un ejemplo de cómo combinar diferentes opiniones de distintos expertos, y otro para mostrar cómo aunar estimaciones de diferentes variables en una más global.

4.1. Combinar diferentes opiniones de expertos

En esta sección aplicamos el método de combinación lineal de opiniones de varios expertos sobre una misma variable. El objetivo es obtener una estimación más confiable que si esta fuera llevada a cabo por un sólo experto. En este ejemplo contamos con tres expertos. Para poder aplicar el método de combinación necesitamos establecer el peso de cada experto. Con este fin, en nuestro ejemplo, se tendrá en cuenta la experiencia de cada miembro y la información de que disponga, tanto en términos de cantidad como de calidad:

- E_1: 10 años de experiencia, dispone se mucha información fiable
- E_2: 5 años de experiencia, dispone se poca información y es poco fiable
- E_3: 15 años de experiencia, dispone se mucha información poco fiable

A los expertos (E_i) se les asocia un peso utilizando el método A.H.P. como se ha explicado anteriormente. Las comparaciones se realizan acorde a la escala verbal propuesta por Saaty que podemos ver en la tabla 1. Las comparaciones entre los diferentes expertos las debe realizar otro experto con el fin de que sea lo más objetiva posible (ver figura 1).

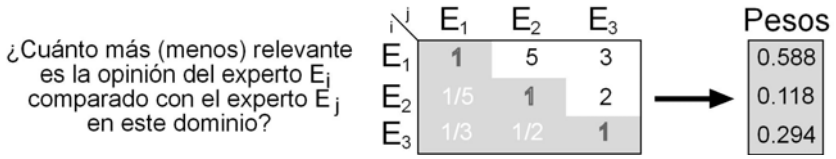

Figura 1. Matriz de juicios y vector de pesos

Los expertos serán los encargados de realizar las estimaciones de una variable aleatoria. En este ejemplo la variable aleatoria que utilizaremos será un factor comúnmente utilizado en el área de estimación de esfuerzo software: la *complejidad del software*. Lo primero que tienen que hacer los expertos es ponerse de acuerdo en lo que se entiende por *complejidad de software* y los valores que puede tomar. Suponemos que en nuestro ejemplo ya se han puesto de acuerdo y han establecido que la variable será discreta y contará con tres estados: *Baja*, *Media* y *Alta*. En la figura 2 podemos ver las distribuciones de probabilidad

de la variable *complejidad* realizada por los tres expertos con los que cuenta el ejemplo.

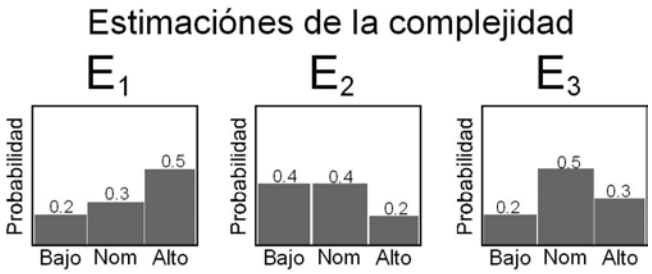

Figura 2. Estimaciones de los expertos sobre la variable *Complejidad*

A continuación lo único que nos queda por hacer es aplicar el método de combinación lineal de opiniones (ver formula 7) y obtener la distribución de probabilidad combinada que podemos ver en la figura 3.

$$p(\Theta) = \sum_{i=1}^{n} w_i p_i(\Theta)$$

$$p(Complejidad_{Baja}) = 0{,}588 * o{,}2 + 0{,}118 * 0{,}4 + 0{,}294 * 0{,}2 = 0{,}2236$$
$$p(Complejidad_{Media}) = 0{,}588 * o{,}3 + 0{,}118 * 0{,}4 + 0{,}294 * 0{,}5 = 0{,}3706$$
$$p(Complejidad_{Alta}) = 0{,}588 * o{,}5 + 0{,}118 * 0{,}2 + 0{,}294 * 0{,}3 = 0{,}4058 \quad (7)$$

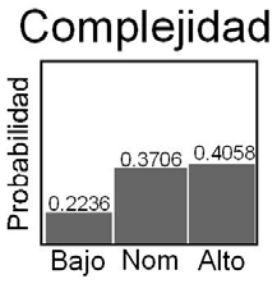

Figura 3. Distribución de probabilidad combinada de la *Complejidad*

4.2. Combinar diferentes variables para establecer una variable globalizadora

En este caso nos planteamos estimar una variable aleatoria a partir de la estimación de cada una de sus partes. Como ejemplo utilizaremos el esfuerzo de

desarrollo de un proyecto software y supondremos que el esfuerzo de desarrollo de un software se establece a partir del esfuerzo de desarrollo de cada una de sus partes (esto no es del todo cierto ya que depende de más factores que afectan al esfuerzo total, pero nos sirve como ejemplo). Pongamos como ejemplo un proyecto software que se compone de tres funcionalidades:

- F_1: Núcleo básico - módulos de seguridad, backup y gestión de usuarios
- F_2: Docu - módulo encargado de gestionar toda la documentación
- F_3: Reservas - módulo encargado de la gestión de reservas de automóviles

Hay que tener en cuenta que, como en este ejemplo, las distribuciones de probabilidad que queremos combinar deben ser del mismo tipo, en nuestro caso el esfuerzo necesario para desarrollar cada una de las funcionalidades. Lo primero que debemos hacer es establecer los pesos. En nuestro ejemplo los pesos nos indican la influencia relativa de cada funcionalidad en el esfuerzo total. Para esto crearemos la matriz de comparaciones. Las filas y las columnas de la matriz de comparaciones estarán formadas por las diferentes alternativas, en nuestro caso las funcionalidades F_1, F_2 y F_3. A continuación, con ayuda de un experto en el dominio, realizaremos las comparaciones entre cada una de las funcionalidades respondiendo a la pregunta: *¿Cuánto más (menos) influencia tiene la funcionalidad F_i sobre el esfuerzo comparándolo con la funcionalidad F_j?*. Ver figura 4.

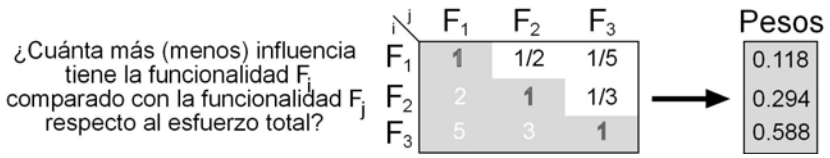

Figura 4. Matriz de juicios y vector de pesos

Para responder a estas comparaciones utilizaremos la escala verbal propuesta por Saaty que aparece en el cuadro 1. Podemos ver en la figura 4 las comparaciones realizadas por un experto así como el resultado del siguiente paso: calcular el vector de proporciones. Dado que este vector responde a la pregunta antes formulada que compara la influencia de cada funcionalidad sobre el esfuerzo total y sus valores suman uno, lo podemos interpretar como el peso de cada funcionalidad respecto al esfuerzo.

Una vez calculados los pesos procedemos a combinar las distribuciones de probabilidad de cada funcionalidad. Estas distribuciones representan una estimación del esfuerzo de desarrollo de cada funcionalidad. En nuestro ejemplo se trata de distribuciones de probabilidad discretas en tres estados: *Bajo*, *Nominal* y *Alto*. En la figura 5 podemos ver las distribuciones de probabilidad obtenidas a partir de un experto.

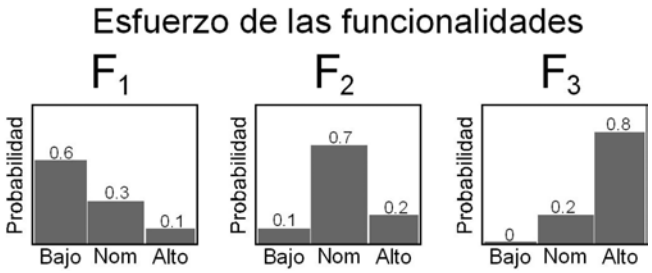

Figura 5. Distribuciones de probabilidad de las funcionalidades y el esfuerzo total

Los cálculos necesarios para establecer la distribución de probabilidad del esfuerzo total se muestran en 8. Se calcula por estados: se multiplica el peso de cada variable por la probabilidad del estado de dicha variable. Se suman y ya tenemos la probabilidad del estado de las variable global.

$$p(\Theta) = \sum_{i=1}^{n} w_i p_i(\Theta)$$

$$p(\Theta_{Bajo}) = 0{,}118 * o{,}6 + 0{,}294 * 0{,}1 + 0{,}588 * 0 = 0{,}1002$$
$$p(\Theta_{Nominal}) = 0{,}118 * o{,}3 + 0{,}294 * 0{,}7 + 0{,}588 * 0{,}2 = 0{,}3588$$
$$p(\Theta_{Alto}) = 0{,}118 * o{,}1 + 0{,}294 * 0{,}2 + 0{,}588 * 0{,}8 = 0{,}541 \quad (8)$$

Se trata de un método muy sencillo de calcular una vez tenemos los pesos de las variables. En la figura 6 podemos ver la distribución de probabilidad del esfuerzo total.

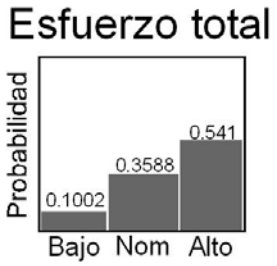

Figura 6. Distribuciones de probabilidad del esfuerzo total

5. Conclusiones y trabajo futuro

En este artículo se ha presentado una técnica de combinación de distribuciones de probabilidad. El método de combinación lineal de opiniones resulta ser

un método muy sencillo de aplicar y de entender, donde la mayor dificultad consiste en establecer los pesos de cada variable o experto. Para el establecimiento de dichos pesos consideramos que el método de ayuda a la toma de decisiones Analytic Hierarchy Process (A.H.P.)resulta apropiado para este fin, además de sencillo y fiable.

Se ha expuesto la combinación de ambos métodos desde dos puntos de vista bastante útiles dentro de la ingeniería del software. Por un lado se puede utilizar la combinación lineal de opiniones para combinar diferentes estimaciones de distintos expertos. Pero también se puede aplicar el mismo enfoque para establecer la estimación de una variable a partir de la combinación de la estimación de sus partes, por ejemplo entendiendo la estimación del esfuerzo de desarrollo de un software como la combinación de las estimaciones las funcionalidades que componen dicho software.

Agradecimientos: Este trabajo se ha desarrollado gracias a la financiación del proyecto TIN2004-06689-C03-01.

Referencias

1. Enrique Castillo, Jose M. Gutierrez, and Ali S. Hadi. *Expert Systems and Probabiistic Network Models*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
2. Esperanza Manso and Jose Javier Dolado. *Técnicas cuantitativas para la gestión en la Ingeniería del software*. Netbiblio, 2007.
3. Clemen R.T. and Winkler R.L. Combining probability distributions from experts in risk analysis. *Risk Analysis*, 19:187–203(17), April 1999.
4. T.L. Saaty. How to make a decision - the analytic hierarchy process. *INTERFACES*, 24(6):19–43, NOV-DEC 1994.
5. Eduardo Miranda. Improving subjective estimates using paired comparisons. *IEEE Software*, 18(1):87–91, feb 2001.
6. Ying-Ming Wang and Taha M. S. Elhag. An approach to avoiding rank reversal in ahp. *Decis. Support Syst.*, 42(3):1474–1480, 2006.
7. Sahrman Barker Martin Shepperd and Martin Aylett. The analytic hierarchy process and data-less prediction. *Empirical Software Engineering Research Group ESERG: TR98-04*, 1998.

On the Correlation between Static Measures and Code Coverage using Evolutionary Test Case Generation

Javier Ferrer, Francisco Chicano, y Enrique Alba

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
{ferrer,chicano,eat}@lcc.uma.es

Resumen. Evolutionary testing is a very popular domain in the field of search based software engineering that consists in automatically generating test cases for a given piece of code using evolutionary algorithms. One of the most important measures used to evaluate the quality of the generated test suites is code coverage. In this paper we want to analyze if there exists a correlation between some static measures computed on the test program and the code coverage when an evolutionary test case generator is used. In particular, we use Evolutionary Strategies (ES) as search engine of the test case generator. We have also developed a program generator that is able to create Java programs with the desired values of the static measures. The experimental study includes a benchmark of 3600 programs automatically generated to find correlations between the measures. The results of this study can be used in future work for the development of a tool that decides the test case generation method according to the static measures computed on a given program.

Palabras clave: Evolutionary testing, branch coverage, evolutionary algorithms, evolutionary strategy

1 Introduction

Automatic software testing is one of the most studied topics in the field of Search-Based Software Engineering (SBSE) [8]. From the first works [10] to nowadays many approaches have been proposed for solving the automatic test case generation problem. This great effort in building computer aided software testing tools is motivated by the cost and importance of the testing phase in the software development cycle. It is estimated that half the time spent on software project development, and more than half its cost, is devoted to testing the product [4]. This explains why the Software Industry and Academia are interested in automatic tools for testing.

Evolutionary algorithms (EAs) have been the most popular search algorithms for generating test cases [8]. In fact, the term *evolutionary testing* was coined to refer to this approach. In the paradigm of *structural testing* a lot of research has been performed using EAs and, in particular, different elements of the structure of a program have been studied in detail. Some examples are the presence of flags in conditions [2], the coverage of loops [5], the existence of internal states [19] and the presence of possible exceptions [16].

The objective of an automatic test case generator used for structural testing is to find a test case suite that is able to cover all the software elements. These elements can be instructions, branches, atomic conditions, and so on. The performance of an automatic test case generator is usually measured as the percentage of elements that the generated test suite is able to cover in the test program. This measure is called *coverage*. The coverage obtained depends not only on the test case generator, but also on the program being tested. Then, we can ask the following research questions:

- *RQ1*: Is there any static measure of the test program having a clear correlation with the coverage percentage?
- *RQ2*: Which are these measures and how they correlate with coverage?

As we said before, coverage depends also on the test case generator. Then, in order to completely answer the questions we should use all the possible automatic test case generators or, at least, a large number of them. We can also focus on one test case generator and answer to the previous questions on this generator. This is what we do in this paper. In particular, we study the influence on the coverage of a set of static software measures when we use an evolutionary test case generator. This study can be used to predict the value of a dynamic measure, coverage, from static measures. This way, it is possible to develop tools taking into account the static measures to decide which automatic test case generation method is more suitable for a given program.

The rest of the paper is organized as follows. In the next section we present the measures that we use in our study. Then, we detail the evolutionary test case generator used in Section 3. After that, Section 4 describes the experiments performed and discusses the results obtained. Finally, in Section 5 some conclusions and future work are outlined.

2 Measures

The measures used in this study are six: number of sentences, number of atomic conditions per condition, total number of conditions, nesting degree, coverage, and McCabe's cyclomatic complexity. The three first measures are easy to understand. The nesting degree is the maximum number of conditional statements that are nested one inside another. In the following paragraphs we describe in more detail the coverage and the McCabe's cyclomatic complexity.

In order to define a coverage measure, we first need to determine which kind of element is going to be "covered". Different coverage measures can be defined depending on the kind of element to cover. *Statement coverage*, for example, is defined as the percentage of statements that are executed. In this work we use *branch coverage*, which is the percentage of branches exercised in a program. This coverage measure is used in most of the related papers in the literature.

Cyclomatic complexity is a complexity measure of code related to the number of ways there are to traverse a piece of code. This determines the minimum number of inputs needed to test all the ways to execute the program. Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of sentences of a program, and a directed edge connects two nodes if the second sentence might be executed immediately after the first sentence. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program and is formally defined as follows:

$$v(G) = E - N + 2P; \tag{1}$$

where E is the number of edges of the graph, N is the number of nodes of the graph and P is the number of connected components.

In Figure 1, we show an example of control flow graph (G). It is assumed that each node can be reached by the entry node and each node can reach the exit node. The maximum number of linearly independent circuits in G is 9-6+2=5, and this is the cyclomatic complexity.

The correlation between the cyclomatic complexity and the number of software faults has been studied in some research articles [3, 7]. Most such studies find a strong positive correlation between the cyclomatic complexity and the defects: the higher the complexity

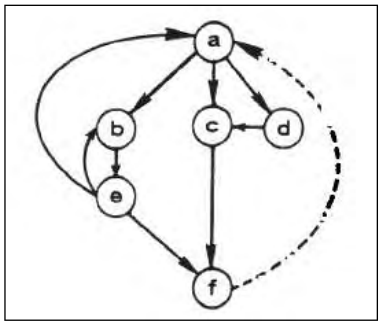

Fig. 1. The original graph of the McCabe's article

the larger the number of faults. For example, a 2008 study by metric-monitoring software supplier Energy [6], analyzed classes of open-source Java applications and divided them into two sets based on how commonly faults were found in them. They found strong correlation between cyclomatic complexity and their faultiness, with classes with a combined complexity of 11 having a probability of being fault-prone of just 0.28, rising to 0.98 for classes with a complexity of 74.

In addition to this correlation between complexity and errors, a connection has been found between complexity and difficulty to understand software. Nowadays, the subjective reliability of software is expressed in statements such as "I understand this program well enough to know that the tests I have executed are adequate to provide my desired level of confidence in the software". For that reason, we make a hard link between complexity and difficulty of discovering errors.

Since McCabe proposed the cyclomatic complexity, it has received several criticisms. Weyuker [18] concluded that one of the obvious intuitive weaknesses of the cyclomatic complexity is that it makes no provision for distinguishing between programs which perform very little computation and those which perform massive amounts of computation, provided that they have the same decision structure. Piwowarski [12] noticed that cyclomatic complexity is the same for N nested `if` statements and N sequential `if` statements.

In connection with our research questions, Weyuker's critic is not relevant, since coverage does not take into account the amount of computation made by a block of statements. However, Piworarski's critic is important in our research because the nesting degree of a program is inverse correlated with the branch coverage as we will show in the experimental section.

3 Test Case Generator

Our test case generator breaks down the global objective (to cover all the branches) into several partial objectives consisting of dealing with only one branch of the program. Then, each partial objective can be treated as a separate optimization problem in which the function to be minimized is a distance between the current test case and one satisfying the partial objective. In order to solve such minimization problem EAs are used. The main loop of the test data generator is shown in Fig. 2.

In a loop, the test case generator selects a partial objective (a branch) and uses the optimization algorithm to search for test cases exercising that branch. When a test case covers a branch, the test case is stored in a set associated to that branch. The structure composed of the sets associated to all the branches is called *coverage table*. After the optimization algorithm stops, the main loop starts again and the test case generator selects a different

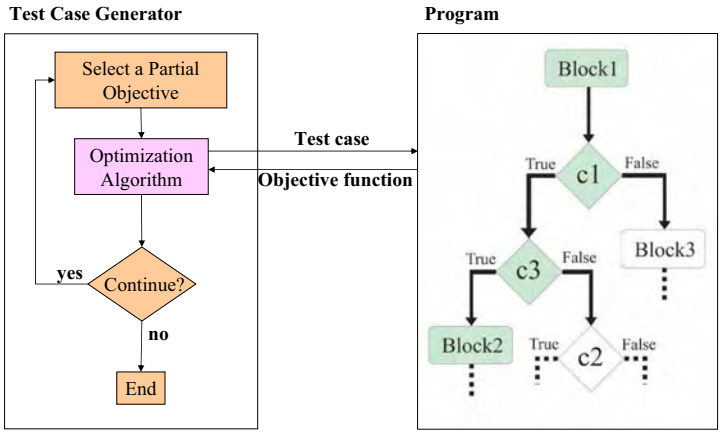

Fig. 2. The test case generation process

branch. This scheme is repeated until total branch coverage is obtained or a maximum number of consecutive failures of the optimization algorithm is reached. When this happens the test data generator exits the main loop and returns the sets of test cases associated to all the branches. In the rest of this section we describe two important issues related to the test case generator: the objective function to minimize and the optimization algorithm used.

3.1 Objective Function

Following on from the discussion in the previous section, we have to solve several minimization problems: one for each branch. Now we need to define an objective function (for each branch) to be minimized. This function will be used for evaluating each test case, and its definition depends on the desired branch and whether the program flow reaches the branching condition associated to the target branch or not. If the condition is reached we can define the objective function on the basis of the logical expression of the branching condition and the values of the program variables when the condition is reached. The resulting expression is called *branch distance* and can be defined recursively on the structure of the logical expression. That is, for an expression composed of other expressions joined by logical operators the branch distance is computed as an aggregation of the branch distance applied to the component logical expressions. For the Java logical operators `&` and `|` we define the branch distance as[1]:

$$bd(a\&b) = bd(a) + bd(b) \tag{2}$$

$$bd(a|b) = \min(bd(a), bd(b)) \tag{3}$$

where a and b are logical expressions.

In order to completely specify the branch distance we need to define its value in the base case of the recursion, that is, for atomic conditions. The particular expression used for the branch distance in this case depends on the operator of the atomic condition. The operands of the condition appear in the expression. A lot of research has been devoted in the past to the study of appropriate branch distances in software testing. An accurate branch distance taking into account the value of each atomic condition and the value of its operands can better guide the search. In procedural software testing these accurate functions

[1] These operators are the Java *and*, *or* logical operators without shortcut evaluation. For the sake of clarity we omit here the definition of the branch distance for other operators.

are well-known and popular in the literature. They are based on distance measures defined for relational operators like $<$, $>$, and so on [9]. We use these distance measures used in the literature.

When a test case does not reach the branching condition of the target branch we cannot use the branch distance as objective function. In this case, we identify the branching condition c whose value must first change in order to cover the target branch (critical branching condition) and we define the objective function as the branch distance of this branching condition plus the *approximation level*. The approximation level, denoted here with $ap(c, b)$, is defined as the number of branching nodes lying between the critical one (c) and the target branch (b) [17].

In this paper we also add a real valued penalty in the objective function to those test cases that do not reach the branching condition of the target branch. With this penalty, denoted by p, the objective value of any test case that does not reach the target branching condition is higher than any test case that reaches the target branching condition. The exact value of the penalty depends on the target branching condition and it is always an upper bound of the target branch distance. Finally, the expression for the objective function is as follows:

$$f_b(x) = \begin{cases} bd_b(x) & \text{if } b \text{ is reached by } x \\ bd_c(x) + ap(c, b) + p & \text{otherwise} \end{cases} \quad (4)$$

where c is the critical branching condition, and bd_b, bd_c are the branch distances of branching conditions b and c.

Nested branches pose a great challenge for the search. For example, if the condition associated to a branch is nested within three conditional statements, all the conditions of these statements must be true in order for the program flow to proceed onto the next one. Therefore, for the purposes of computing the objective function, it is not possible to compute the branch distance for the second and third nested conditions until the first one is true. This gradual release of information might cause efficiency problems for the search (what McMinn calls the *nesting problem* [11]), which forces us to concentrate on satisfying each predicate sequentially.

In order to alleviate the nesting problem, the test case generator selects as objective in each loop one branch whose associated condition has been previously reached by other test cases stored in the coverage table. Some of these test cases are inserted in the initial population of the EA used for solving the optimization problem. The percentage of individuals introduced in this way in the population is called the *replacement factor* and is denoted by Rf. At the beginning of the generation process some random test cases are generated in order to reach some branching conditions.

3.2 Optimization Algorithm

EAs [1] are metaheuristic search techniques loosely based on the principles of natural evolution, namely, adaptation and survival of the fittest. These techniques have been shown to be very effective in solving hard optimization tasks. They are based on a set of tentative solutions (individuals) called *population*. The problem knowledge is usually enclosed in an objective function, the so-called *fitness function*, which assigns a quality value to the individuals. In Fig. 3 we show the main loop of an EA.

Initially, the algorithm creates a population of μ individuals randomly or by using a seeding algorithm. At each step, the algorithm applies stochastic operators such as selection, recombination, and mutation (we call them variation operators in Fig. 3) in order to compute a set of λ descendant individuals P'(t). The objective of the selection operator is to select some individuals from the population to which the other operators will be applied. The

```
t := 0;
P(t) = Generate ();
Evaluate (P(t));
while not StopCriterion do
   P'(t) := VariationOps (P(t));
   Evaluate (P'(t));
   P(t+1) := Replace (P'(t),P(t));
   t := t+1;
endwhile;
```

Fig. 3. Pseudocode of an EA

recombination operator generates a new individual from several ones by combining their solution components. This operator is able to put together good solution components that are scattered in the population. On the other hand, the mutation operator modifies one single individual and is the source of new different solution components in the population. The individuals created are evaluated according to the fitness function. The last step of the loop is a replacement operation in which the individuals for the new population $P(t+1)$ are selected from the offspring $P'(t)$ and the old one $P(t)$. This process is repeated until a stop criterion is fulfilled, such as reaching a pre-programmed number of iterations of the algorithm or finding an individual with a preset target quality. In the following we focus on the details of the specific EAs used in this work to perform the test case generation.

We used an Evolutionary Strategy (ES) for the search of test cases for a given branch. In an ES [14] each individual is composed of a vector of real numbers representing the problem variables ($\mathbf{x}$), a vector of standard deviations (σ) and, optionally, a vector of angles (ω). These two last vectors are used as parameters for the main operator of this technique: the Gaussian mutation. They are evolved together with the problem variables themselves, thus allowing the algorithm to self-adapt the search to the landscape. For the recombination operator of an ES there are many alternatives: each of the three real vectors of an individual can be recombined in a different way. However, this operator is less important than the mutation. The mutation operator is governed by the three following equations:

$$\sigma_i' = \sigma_i \exp(\tau N(0,1) + \eta N_i(0,1)) \ , \tag{5}$$

$$\omega_i' = \omega_i + \varphi N_i(0,1) \ , \tag{6}$$

$$\mathbf{x}' = \mathbf{x} + \mathbf{N}(\mathbf{0}, C(\sigma', \omega')) \ , \tag{7}$$

where $C(\sigma', \omega')$ is the covariance matrix associated to σ' and ω', $N(0,1)$ is the standard univariate normal distribution, and $\mathbf{N}(\mathbf{0}, C)$ is the multivariate normal distribution with mean $\mathbf{0}$ and covariance matrix C. The subindex i in the standard normal distribution indicates that a new random number is generated anew for each component of the vector. The notation $N(0,1)$ is used for indicating that the same random number is used for all the components. The parameters τ, η, and φ are set to $(2n)^{-1/2}$, $(4n)^{-1/4}$, and $5\pi/180$, respectively, as suggested in [15]. With respect to the replacement operator, there is a special notation to indicate wether the old population is taken into account or not to form the new population. When only the new individuals are used, we have a (μ, λ)-ES; otherwise, we have a $(\mu + \lambda)$-ES.

Regarding the representation, each component of the vector solution $\mathbf{x}$ is rounded to the nearest integer and used as actual parameter of the method under test. There is no limit in the input domain, thus allowing the ES to explore the whole solution space. This contrasts with other techniques such as genetic algorithm with binary representation that can only explore a limited region of the search space.

In the experimental section we have used two ESs: a (1+5) ES without crossover, that we call (1+5)-ES, and a (25+5)-ES with uniform crossover, called (25+5)-ES$_c$. In the latter algorithm, random selection was used.

4 Experimental Section

In order to study the correlations between the static measures and the coverage, we first need a large number of test programs. For the study to be well-founded, we require a lot of programs having the same value for the static measures as well as programs having different values for the measures. It is not easy to find such a variety of programs in the related literature. Thus, we decided to automatically generate the programs. This way, it is possible to randomly generate programs with the desired values for the static measures and, most important, we can generate different programs with the same values for the static measures.

The automatic program generation raises a non-trivial question: are the generated programs realistic? That is, could them be found in real-world? Using automatic program generation it is not likely to find programs that are similar to the ones who a programmer would make. This is especially true if the program generation is not driven by a specification. However, this is not a drawback in our study, since we want to analyze the correlations between some static measures of the programs and code coverage. In this situation, "realistic programs" means programs that have similar values for the considered static measures as the ones found in real-world; and we can easily fulfil this requirement.

Our program generator takes into account the desired values for the number of atomic conditions per condition, the nesting degree, the number of sentences and the number of variables. With these parameters and other (less important) ones, the program generator creates a program with a defined control flow graph containing several conditions. The main features of the generated programs are:

- They deal with integer input parameters.
- Their conditions are joined by whichever logical operator.
- They are randomly generated.

Due to the randomness of the generation, the static measures could take values that are different from the ones specified in the configuration file of the program generator. For this reason, in a later phase, we used the free tool CyVis to measure the actual values for the static measures. CyVis [13] is a free software tool for metrics collection, analysis and visualization of Java based programs.

The methodology applied for the program generation is the following. First, we analyzed a set of Java source files from the JDK 1.5, in particular, the package `java.util`; and we computed the static measures on these files. In Table 1 we show a summary of this analysis. Next, we used the ranges of the most interesting values obtained in this previous analysis as a guide to generate Java source files having values in the same range for the static measures. This way, we generate programs that are realistic with respect to the static measures, making the following study meaningful. Finally, we generated a total of 3600 Java programs using our program generator and we applied our test case generator with (1+5)-ES and (25+5)-ES$_c$ to all of them 5 times. We need 5 independent runs of each algorithm because they are stochastic algorithms: one single run is not meaningful; instead, several runs are required and the average and the standard deviation are used for comparison purposes. The experimental study requires a total of $3600 \cdot 5 \cdot 2 = 36000$ independent runs of the test case generator.

4.1 Results

After the execution of all the independent runs for the two algorithms in the 3600 programs, in this section we analyze the linear correlation between the static measures and the coverage.

Tabla 1. Range of values obtained in java.util library

Parameter	Minimum	Maximum
Number of Sentences	10	294
Nesting Degree	1	7
McCabe Cyclomatic Complexity	1	80

We use the Pearson correlation coefficient to study the degree of linear correlation between two variables. This coefficient is usually represented by r, and is computed with the following formula:

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{(n-1)S_x S_y}$$

where x_i and y_i are the values of the samples, n is the number of cases, S_x and S_y are the standard deviations of each variable.

First, we study the correlation between the number of sentences and the branch coverage. We obtain a correlation of 13.4% for these two variables using the (25+5)-ES$_c$ algorithm and 18.8% with the (1+5)-ES algorithm[2]. In Figure 4 we plot the average coverage against the number of sentences for ES and all the programs. It can be observed that the number of sentences is not a significant parameter and it has no influence in the coverage measure. The results obtained with ES$_c$ are similar and we omit the corresponding graph.

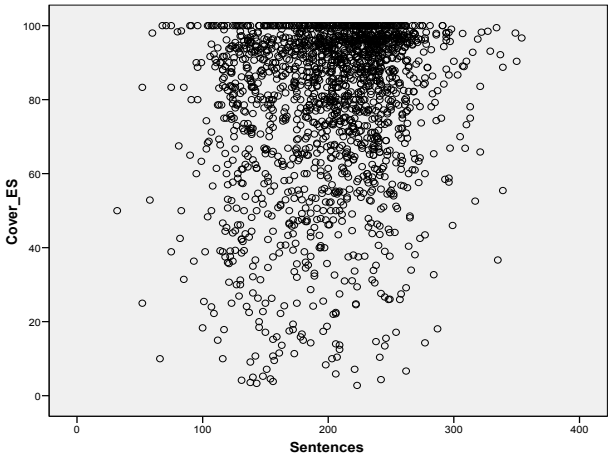

Fig. 4. Average branch coverage against the number of sentences for ES in all the programs

In second place, we study the correlation between the number of atomic conditions per condition and coverage. In Table 2 we show the average coverage obtained for all the programs with the same number of atomic conditions per condition when ES and ES$_c$ are used. From the results we conclude that there is no linear correlation between these two variables. The minimum values for coverage are reached with 1 and 7 atomic conditions per condition. This could seem counterintuitive, but a large condition with a sequence of

[2] In order to save room, in the following we use ES$_c$ and ES to refer to (25+5)-ES$_c$ and (1+5)-ES, respectively

logical operators, can be easily satisfied due to OR operators. Otherwise, a short condition composed of AND operators can be more difficult to satisfy.

Tabla 2. Correlation between the number of atomic conditions per condition and average coverage for both algorithms

At. Conds.	ES_c	ES
1	$83.59\%_{21.69}$	$77.74\%_{23.49}$
2	$82.85\%_{20.33}$	$78.54\%_{21.82}$
3	$85.15\%_{19.82}$	$81.39\%_{21.53}$
4	$\mathbf{88.04}\%_{16.42}$	$\mathbf{83.23}\%_{19.87}$
5	$85.06\%_{19.19}$	$80.50\%_{21.53}$
6	$84.16\%_{19.58}$	$79.12\%_{23.09}$
7	$81.24\%_{21.18}$	$76.26\%_{23.74}$
r	-0.50 %	0.30 %

Now we analyze the influence on coverage of total number of conditions of a program. In Figure 5, we can observe that programs with a small number of conditions reach a higher coverage than the programs with a large number of conditions. This could be interpreted as large programs with many conditions are more difficult to test. If there are a lot of conditions, this generates a lot of different paths in the control flow graph, this fact is also weighted in the cyclomatic complexity. The correlation coefficient is -16.4% for ES and -21.6% for ES_c.

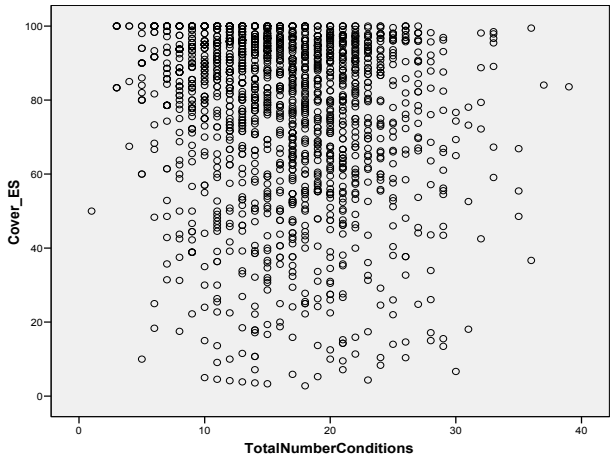

Fig. 5. Average branch coverage against the total number of conditions for ES in all the programs

Let us analyze the nesting degree. In Table 3, we summarize the coverage obtained with different nesting degree. If the nesting degree is increased, the branch coverage decreases and vice versa. It is clear that there is an inverse correlation between these variables. The correlation coefficients are -45.7% and -47% for ES and ES_c respectively, what confirms the observations. As we said in Section 3.1, nested branches pose a great challenge for the search.

Tabla 3. Correlation between nesting degree and average coverage for both algorithms

Nesting	ES_c	ES
1	**96.03**$\%_{5.74}$	**93.56**$\%_{8.15}$
2	94.07$\%_{8.85}$	89.97$\%_{11.29}$
3	90.02$\%_{13.67}$	85.29$\%_{16.71}$
4	85.06$\%_{16.43}$	80.09$\%_{19.18}$
5	77.83$\%_{22.53}$	72.02$\%_{23.90}$
6	73.02$\%_{24.80}$	67.47$\%_{24.66}$
7	64.45$\%_{25.96}$	58.41$\%_{27.16}$
r	-47,00 %	-45,70 %

Finally, we study the correlation between the McCabe cyclomatic complexity and coverage. In Figures 6 and 7, we plot the average coverage against the cyclomatic complexity for ES and ES_c in all the programs. In general we can observe that there is no clear correlation between both parameters. The correlation coefficients are -4.8% and -8.6% for ES and ES_c, respectively. These values are very low, and confirms the observations: McCabe's cyclomatic complexity and branch coverage are not correlated. This is somewhat surprising, we would expect a positive correlation between the complexity of a program and the difficulty to get an adequate test suite. However, this is not true: McCabe's cyclomatic complexity cannot be used as a measure of the difficulty to get an adequate test suite. We can go one step forward and try to explain this unexpected behaviour. Cyclomatic complexity is not correlated with branch coverage because complexity does not take into account the nesting degree, which is the parameter with a higher influence on branch coverage.

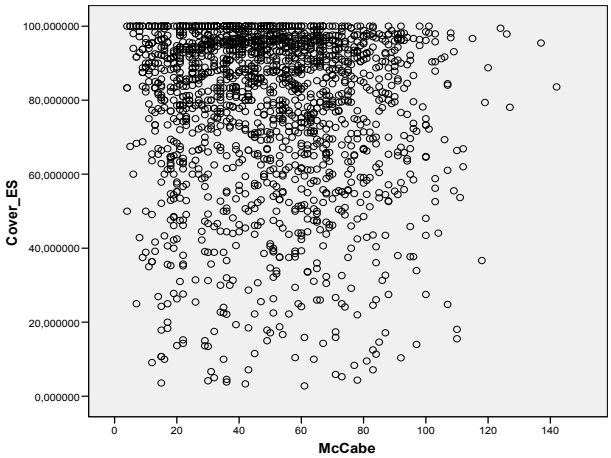

Fig. 6. Average branch coverage against the McCabe's cyclomatic complexity for ES in all the programs

5 Conclusions

In this work we have analyzed the correlation between the branch coverage obtained using automatically generated test suites and five static measures: number of sentences, number

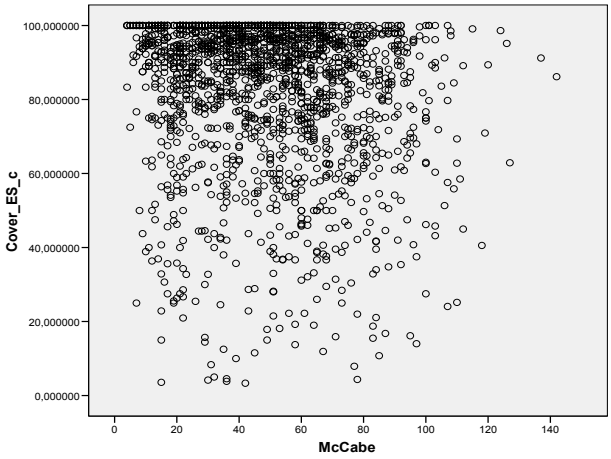

Fig. 7. Average branch coverage against the McCabe's cyclomatic complexity for ES_c in all the programs

of atomic conditions per condition, number of total conditions, nesting degree and McCabe's cyclomatic complexity. The results show that there is a small correlation between the branch coverage and the number of sentences, the number of conditions per conditions and the number of total conditions. On the other hand, the nesting degree is the measure that is more (inverse) correlated to the branch coverage: the higher the nesting degree the lower the coverage. Finally, there is no correlation between McCabe's cyclomatic complexity and branch coverage, that is, cyclomatic complexity does not reflect the difficulty of the automatic generation of test suites.

As future work we plan to advance in the analysis of static and dynamic measures of a program in order to propose a reliable measure of the difficulty of generating adequate test suites. In addition, we want to make an exhaustive study of the static measures in object-oriented programs, using a larger number of static measures. Finally, we would like to design new static measures able to reflect the real difficulty of automatic testing for specific test case generators.

Acknowledgements

This work has been partially funded by the Spanish Ministry of Science and Innovation and FEDER under contract TIN2008-06491-C04-01 (the M* project). It has also been partially funded by the Andalusian Government under contract P07-TIC-03044 (DIRICOM project).

References

1. T. Bäck, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation.* Oxford University Press, New York NY, 1997.
2. André Baresel, David Wendell Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, 2004.
3. Victor Basili and Barry Perricone. Software errors and complexity: an empirical investigation. *ACM commun*, 27(1):42–52, 1984.
4. Boris Beizer. *Software testing techniques.* Van Nostrand Reinhold Co., New York, NY, USA, 2nd edition, 1990.