Applying Particle Swarm Optimization to Software Testing

Andreas Windisch
DaimlerChrysler AG
Research and Technology
Alt-Moabit 96a, D-10559
Berlin, Germany
Phone: +49 30 39982 463

Stefan Wappler
Technical University of Berlin
DaimlerChrysler AITI
Ernst-Reuter-Platz 7, D-10587
Berlin, Germany
Phone: +49 30 39982 358

Joachim Wegener
DaimlerChrysler AG
Research and Technology
Alt-Moabit 96a, D-10559
Berlin, Germany
Phone: +49 30 39982 232

ABSTRACT

Evolutionary structural testing is an approach to automatically generating test cases that achieve high structural code coverage. It typically uses genetic algorithms (GAs) to search for relevant test cases.

In recent investigations particle swarm optimization (PSO), an alternative search technique, often outperformed GAs when applied to various problems. This raises the question of how PSO competes with GAs in the context of evolutionary structural testing.

In order to contribute to an answer to this question, we performed experiments with 25 small artificial test objects and 13 more complex industrial test objects taken from various development projects. The results show that PSO outperforms GAs for most code elements to be covered in terms of effectiveness and efficiency.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging — *Test coverage of code, Testing tools*

General Terms

Verification

Keywords

evolutionary testing, genetic algorithm, particle swarm optimization, automatic test case generation

1. INTRODUCTION

Evolutionary structural testing [8, 12, 14, 15], a search-based approach to automatically generating relevant unit test cases, is a well-studied technique that has been shown to be successful for numerous academic test objects as well as for various industrial ones. Because of the complex search spaces that most test objects imply and that are often hard to understand due to their high dimensionality, genetic algorithms have been regarded as being especially adequate search strategies since they are able to deal with multi-modality, non-linearity, and discontinuity quite well.

However, genetic algorithms have started getting competition from other heuristic search techniques, such as the particle swarm optimization. Various works (e.g. [1, 4, 5, 9, 7]) show that particle swarm optimization is equally well suited or even better than genetic algorithms for solving a number of test problems[1]. At the same time, a particle swarm algorithm is much simpler, easier to implement and has a fewer number of parameters that the user has to adjust than a genetic algorithm. The performance and the above-mentioned characteristics of the particle swarm optimization favor its application as a search engine for evolutionary structural testing.

This paper reports on our results from an empirical comparison of a genetic algorithm and a particle swarm algorithm applied to evolutionary structural testing. We selected 25 artificial test objects that cover a broad variety of search space characteristics (e.g. varying number of local optima), and 13 industrial test objects taken from various development project. The results indicate that particle swarm optimization is well-suited as a search engine for evolutionary structural testing and tends to outperform genetic algorithms in terms of code coverage achieved by the delivered test cases and the number of needed evaluations.

The paper is structured as follows: section 2 introduces evolutionary structural testing, section 3 describes the used genetic algorithm and particle swarm optimization in more detail, section 3.3 deals with previous comparisons of PSO and GA in general, section 4 includes the results of the experiments, and section 5 summarizes the paper and gives suggestions for future work.

2. EVOLUTIONARY STRUCTURAL TESTING

Since it is apparently impossible to exercise a software unit with all possible combinations of input data, a subset of test data must be found that is considered to be relevant or adequate, respectively. Therefore the generation of relevant test cases for a given software unit is a critical task that directly affects the quality of the overall testing of this unit. A test case is a set of input data with which the unit is to be executed, and the expected outcome. Often, the structural properties of the software unit are considered to answer the

[1]The term *test problem* does not relate to software testing but rather to well-defined optimization problems used to compare the performances of different optimization techniques.

©ACM, 2007. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version will be published in GECCO '07: Proceedings of the 2007 conference on Genetic and evolutionary computation
GECCO'07, July 7–11, 2007, London, England, United Kingdom.
.

question as to what relevant test cases are. For instance, a set of test cases that lead to the execution of each statement of the software unit under test are said to be adequate with respect to statement coverage. Branch coverage, another coverage criterion requiring all branches of the control flow graph of the unit under test to be traversed, is demanded by various industrial testing standards and guidelines.

In general, the process of test case generation is time-consuming and error-prone when done manually. Evolutionary structural testing (EST) [15, 8, 12, 14], an automatic test case generation technique, has been developed in order to provide relief. EST interprets the task of test case generation as an optimization problem and tries to solve it using a search technique, i.e. a genetic algorithm. A genetic algorithm is a meta-heuristic optimization technique, which is dealt with in more detail in section 3.1. It requires a *fitness function* to be provided, which is used to assess the ability of a particular solution to solve the given optimization problem. Depending on the selected coverage criterion, the source code under test is partitioned into single test goals that have to be optimized separately. In the context of EST, the construction of fitness functions using the two distance metrics *approximation level* and *branch distance* has proven of value [14]. The approximation level relates to the control flow graph of the unit under test. It corresponds to the number of critical branches that are in between the problem node and the target (where the target is the code element to be covered and the problem node is the node at which execution diverges down a branch that makes it impossible to reach the target). Branch distance relates to the condition assigned to the problem node. It expresses how "close" the evaluation of this condition is to delivering the boolean result necessary for reaching the target node.

3. APPLIED SEARCH TECHNIQUES

This section describes both search techniques to be compared, namely genetic search and particle swarm optimization. Section 3.1 and 3.2 characterize both strategies in detail whereas section 3.3 summarizes previous comparisons and their results.

3.1 Genetic Search

Genetic search, carried out by a genetic algorithm, is a meta-heuristic optimization technique that mimics the principles of the Darwinian theory of biological evolution. Its adequacy for solving non-linear, multi-modal, and discontinuous optimization problems has drawn the attention of many researchers and practitioners during the last decades.

A genetic algorithm works with a set of potential solutions for the given optimization problem. Through multiple modifications applied iteratively to the current set of solutions, better solutions and finally an optimum solution is supposed to be found. These modifications include *crossover* and *mutation.* While crossover generates new offspring solutions by combining multiple existing solutions, mutation randomly changes parts of an existing solution to yield a new one. The selection of candidate solutions that should undergo crossover and mutation is based on their *fitness.* The fitness of a solution is its ability to solve the optimization problem at hand; it is calculated using the *fitness function.* This function is problem-specific; its suitability essentially contributes to the success and performance of the optimization process. Figure 1 shows the main workflow of a genetic algorithm. The evaluation of the new candidate solutions comprises the calculation of the fitness values. The algorithm terminates if either an optimum solution is found or other predefined termination criteria apply, e.g. a maximum number of iterations has been reached.

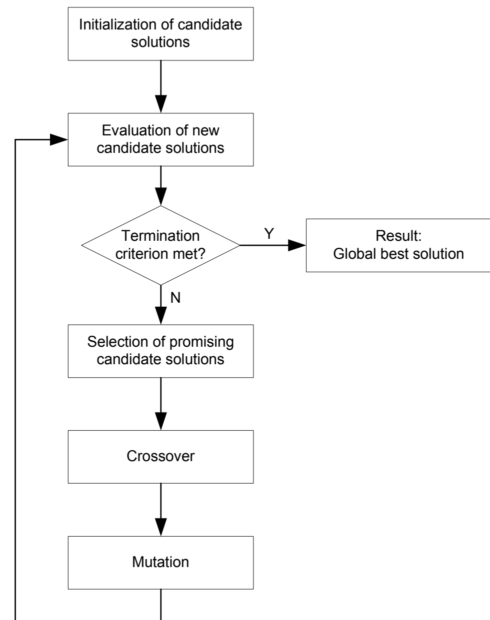

Figure 1: Workflow of a genetic algorithm

3.2 Particle Swarm Optimization

In comparison with genetic search, the particle swarm optimization is a relatively recent optimization technique of the swarm intelligence paradigm. It was first introduced in 1995 by Kennedy and Eberhart [10, 2]. Inspired by social metaphors of behavior and swarm theory, simple methods were developed for efficiently optimizing non-linear mathematical functions. PSO simulates swarms such as herds of animals, flocks of birds or schools of fish.

Similar to genetic search, the system is initialized with a population of random solutions, called particles. Each particle maintains its own current position, its present velocity and its personal best position explored so far. The swarm is also aware of the global best position achieved by all its members. The iterative appliance of update rules leads to a stochastic manipulation of velocities and flying courses. During the process of optimization the particles explore the D-dimensional space, whereas their trajectories can probably depend both on their personal experiences, on those of their neighbors and the whole swarm, respectively. This leads to further explorations of regions that turned out to be profitable. The best previous position of particle i is denoted by $pbest_i$, the best previous position of the entire population is called $gbest$.

Figure 2 shows the general workflow of a PSO-algorithm. The termination criterion can be either a specific fitness value, the achievement of a maximum number of iterations or the general convergence of the swarm itself.

Since its first presentation, many improvements and extensions have been worked out to improve the algorithm in various ways and have provided promising results for the

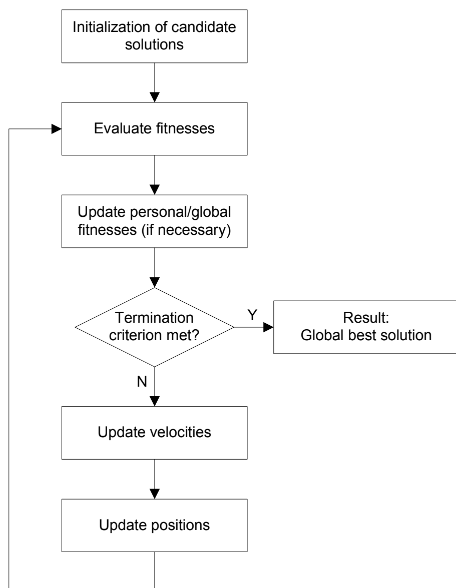

Figure 2: Workflow of a PSO algorithm

optimization of well-known test functions. A novel and auspicious approach is the *Comprehensive Learning Particle Swarm Optimizer* [11] (CL-PSO). It applies a new learning-strategy, where each particle learns from different neighbors for each dimension separately dependent on its assigned learning rate Pc_i. This happens until the particle does not achieve any further improvements for a specific number of iterations called the refreshing gap m; finally yielding a re-assignment of particles.

The mentioned learning-rate Pc_i of particle i is a probability lying between 0.05 and 0.5, determining whether particle i learns from its own or another particle's *pbest* for the current dimension. This probability is assigned to each particle during initialization and remains unchanged for assuring the particle's diverse levels of exploration and exploitation abilities. If the considered dimension is to be learned from another particle's *pbest*, this particle will be appointed in a tournament selection manner: two particles are randomly chosen and the better one is selected. This is done for each dimension d and the resulting list of particles $f_i(d)$ is used in the following velocity update rule for time step t:

$$v_i^d(t) \leftarrow \omega \cdot v_i^d(t-1) + c \cdot r_i^d \cdot (pbest_{f_i(d)}^d(t-1) - x_i^d(t-1)) \quad (1)$$

where the D-dimensional vectors $x_i = (x_i^1, ..., x_i^d, ..., x_i^D)^\top$ and $v_i = (v_i^1, ..., v_i^d, ..., v_i^D)^\top$, with $x_i^d \in [lb^d, ub^d]$, $v_i^d \in [-V_{max}^d, V_{max}^d]$, $d \in [1, D]$ represent the position and velocity of particle i. lb^d, ub^d describe dth dimension's lower and upper bounds whereas V_{max}^d defines its respective maximum and minimum velocity. Accordingly, $pbest_{f_i(d)}^d$ is the personal best position found by the particle assigned for dimension d. The inertia weight ω controls the impact of the previous history on the new velocity. c is an acceleration coefficient weighting the influence of the cognitive and social component respectively in proportion to ith particle's present momentum. Finally, r_i^d is a uniformly distributed random variable in the range of $[0, 1]$ for dimension d.

The fitness of a particle often depends on all D parameters. Hence a particle close to the optimum in some dimensions can probably be evaluated with a poor fitness when processed by the original PSO version, due to the poor solutions of the remaining dimensions. This is counteracted by the learning strategy presented in [11], which consequently enables higher quality solutions to be located. It was shown that CL-PSO in comparison to some other PSO variants yields significantly better solutions for multi-modal problems.

Owing to this, we chose CL-PSO as PSO variant for the upcoming comparison; its detailed configuration is shown in section 4.2.

3.3 Comparison of GA and PSO

Genetic algorithms have been popular because of the parallel nature of their search and essentially because of their ability to effectively solve non-linear, multi-modal problems. They can handle both discrete and continuous variables without requiring gradient information. In comparison, PSO is well-known for its easy implementation, its computational inexpensiveness and its fast convergence to optimal areas of the solution space. Although it yields its best performance on continuous-valued problems, it can also handle discrete variables after slight modifications.

Many researchers have compared both optimization techniques over the last years. Hodgson [5] compared them by applying them to the atomic cluster optimization problem. The task consists of minimizing a highly multi-modal energy function of clusters of atoms. His calculations show PSO to be noticeably superior to both a generic GA and a purpose-built problem-specific GA. Clow and White [1] compared both techniques by using them to train artificial neural networks used to control virtual racecars. Due to the continuousness of the neural weights being optimized, PSO turned out to be superior to GA for all accomplished tests - yielding a higher and much faster growing mean fitness. Hassan, Cohanim and De Weck [4] compared both techniques with a set of eight well-known optimization benchmark test problems, and concluded that PSO was equally effective and more efficient in general. Nevertheless, the superiority of PSO turned out to be problem-dependent. The difference in computational efficiency was found to be greater when the search strategies were used to solve unconstrained problems with continuous variables and less when they were applied to constrained continuous or discrete variables. Jones [9] also compared both approaches by letting them identify two mathematical model parameters. Although he noted that both approaches were equally effective, he concluded that GA outperformed PSO with regard to efficiency. However, attention should be paid to Jones' PSO variant, as it used the same random variables for all dimensions during one velocity update - which turned out to perform worse than using different variables for each dimension as proposed originally. This could be the reason for the poor results for PSO presented by Jones. Horák, Chmela, Oliva and Raida [7] analyzed the abilities of PSO and GA to optimize dual-band planar antennas for mobile communication applications. They found that PSO was able to obtain slightly better results than GA, but that PSO took more cpu-time.

The higher effectiveness and efficiency generally ascribed to PSO leads to the hypothesis that it will improve evolutionary structural testing, too.

4. EMPIRICAL COMPARISON

This section describes the experimental setup and the achieved results. Sections 4.1 and 4.2 describe the test system used to realize evolutionary structural testing and the configuration of both search engines to be compared, respectively. Section 4.3 provides some insight into the test objects used and section 4.4 presents the obtained results.

4.1 Test System

Over the last years, DaimlerChrysler has developed a test system that implements the ideas of evolutionary structural testing [14]. Figure 3 shows a high-level view of the structure of this system. The inputs to this system are the source code under test (including the main module, all required header files and dependent modules) and the interface settings. These settings describe which function to test and which value ranges to use for the input variables. For instance, binary signal variables represented by an integer input variable in the source code might be restricted to the values $\{0, 1\}$. The output of the test system is a set of test cases that are intended to achieve a high structural coverage of the given source code. At the moment, the coverage metrics statement coverage, branch coverage, and two versions of condition coverage are implemented and selectable by the user.

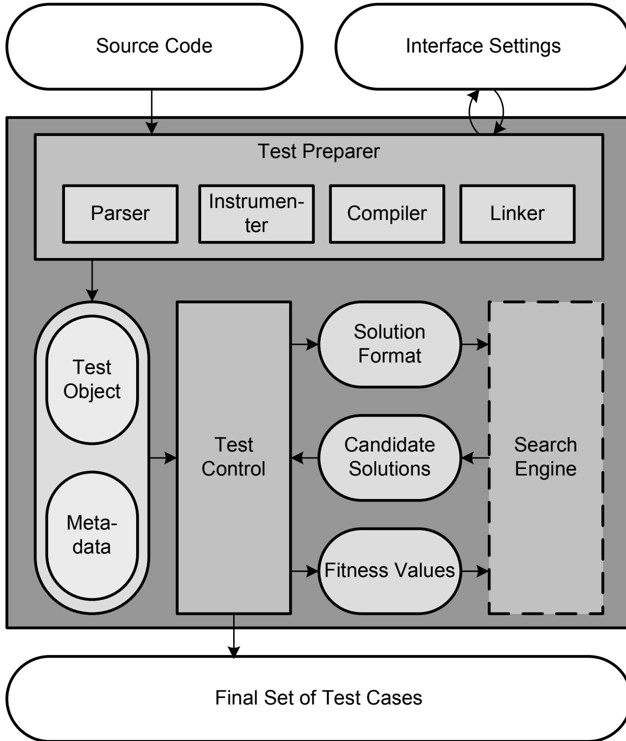

Figure 3: Abstract view of test system structure

The test system consists of the three major components Test Preparer, Test Control, and a search engine. Test Preparer analyzes the given source code and adds the instrumentation statements. These statements are necessary to comprehend the execution flow taken when executing the function under test with a particular set of input values. It also compiles and links the instrumented source code. Additionally, it provides descriptive metadata for the test object, such as the interface specification and the control flow graph. Test Control takes the instrumented test object and performs individual optimizations for each code element to be covered (e.g. in the case of branch coverage, it runs an optimization for each branch of the control flow graph of the function under test). An optimization consists of the following steps: Initially, the interface of the function under test (the input variables) is reported to the search engine in order to generate test inputs that correspond to the functional interface. Afterwards, the search engine provides a set of test inputs. Test Control executes the function under test using the provided inputs and calculates the fitness values based on the produced execution flow. Then, it reports the fitness values to the search engine and waits for the next set of test inputs unless a test input is found that covers the code element under question or another termination criterion applies. These steps are repeated for each test goal. Finally, Test Control creates a final set of test cases that are delivered to the user.

The test system has normally been configured to use the Genetic and Evolutionary Algorithms Toobox (GEATbx) [3] as a search engine. For our experiments, we developed a new toolbox that implements the PSO algorithms mentioned in section 3.2 and provides the same interface as the GEATbx. Hence, the test system can easily switch between the two different search engines.

4.2 Configuration of the search algorithms

Except for two slight modifications described below, we used CL-PSO with all variables set to the values suggested in [11]. The configuration is shown in table 1. The mentioned modifications concern the ability to both handle discrete variables and a changed search bounds condition. The former is solved by simply rounding potential solutions to the nearest integer number when the variables being investigated are discrete. CL-PSO originally uses a boundary condition known as *invisible walls* [13], meaning that the particles are allowed to exceed the borders, but their fitness will solely be calculated and updated if they are within the range. However our experiments with different boundary conditions have shown that this behavior can lead to a sustained and undesired non-observance of particles in some particular cases. Other conditions such as *absorbing* and *reflecting walls* [13] proved to be more or less appropriate depending on the problem. Accordingly, we decided to use the *damping walls* approach proposed by Huang and Mohan [6], realizing a composition of both *absorbing* and *reflecting walls* by stochastically reducing the momentum during reflection of particles on the imaginary boundary walls back to the search space.

Parameter	Value
No. of particles	40
Inertia weight ω	Lin. decreased: 0.9 to 0.4
Acceleration coefficient c	1.49445
Velocity bound V^d_{max}	$(ub^d - lb^d)/2$
Refreshing gap m	7
Boundary condition	Damping walls
Termination	1200 iterations

Table 1: PSO settings

Table 2 shows the settings used for the experiments that

applied the GA as search engine. These settings have emerged over the last few years by experimentation with numerous different industrial test objects. The first 3 rows indicate

parameter		value
subpopulations	number	6
	size	40 individuals each
competition	interval	10
	rate	0.1
	min. size	10
migration	topology	complete net
	interval	13
	rate	0.1
selection	name	stochastic universal sampling
	pressure	1.7
	generation gap	0.9
crossover	type	discrete
	rate	1.0
mutation	type	real number
	precision	17
mutation ranges	subpop. #1	0.1
	subpop. #2	0.01
	subpop. #3	0.001
	subpop. #4	0.0001
	subpop. #5	0.00001
	subpop. #6	0.000001
reinsertion	type	fitness-based
termination	generations	200

Table 2: GA settings

that the GA implements a regional model with 6 subpopulations. Due to the different mutation rates per subpopulation, some subpopulations might be more successful than others. Competition among these subpopulations means that more successful subpopulations receive individuals from less successful ones during the search. Regardless of competition, the best individuals are exchanged among the subpopulations each 13 generations via migration.

We decided to use branch coverage for the coverage criterion since its relevance is widely accepted and enforced by various industrial quality standards. Branch coverage measures the number of branches of the control flow graph of the function under test that are traversed when the generated test cases are executed. For the purpose of test case generation, each branch becomes an individual test goal for which an individual optimization is carried out.

In order to acquire results with sufficient statistical significance, all experiments for both GA and PSO were repeated 30 times.

4.3 Test Objects

A total of 25 small artificial test objects have been created that feature different grades of complexity. They differ in both the number and type of parameters and the number of local optima. The local optima relate to the search space of the most difficult test goal that each test object exhibits. Each test object has one condition that needs to be satisfied - in the context of branch coverage in order to successfully traverse its path of the corresponding control flow graph. In this context, one branch is typically easy to be covered without requiring an optimization while its sibling branch is relatively hard to be covered and requires an optimization.

The characteristics of the test objects are shown in table 3. Column *vars* shows the number of input parameters the

test object	vars	local opt.	boolean	integer	double
f01	1	0	x		
f02	1	0		x	
f03	1	0			x
f04	1	1	x		
f05	1	1		x	
f06	1	1			x
f07	1	4		x	
f08	1	4			x
f09	2	0	x		
f10	2	0		x	
f11	2	0			x
f12	2	24		x	x
f13	3	0	x		
f14	3	0		x	
f15	3	0			x
f16	3	0	x	x	x
f17	3	49	x	x	x
f18	6	0	x		
f19	6	0		x	
f20	6	0			x
f21	6	0	x	x	x
f22	12	0	x		
f23	12	0		x	
f24	12	0			x
f25	12	0	x	x	x

Table 3: Artificial test objects

test object possesses and that are to be optimized. *Local opt.* shows the number of local optima of the search space. Column *boolean*, *integer* and *double* indicate the types of parameter. For example, *f12* is a function with two parameters (one of type integer and one of type double), constructed to realize the fitness landscape shown in figure 4. It features a total of 24 local optima around the global one in position $(0, 0)$. Double valued parameters were limited to the range $[-10e6, 10e6]$ to minimize the huge search space.

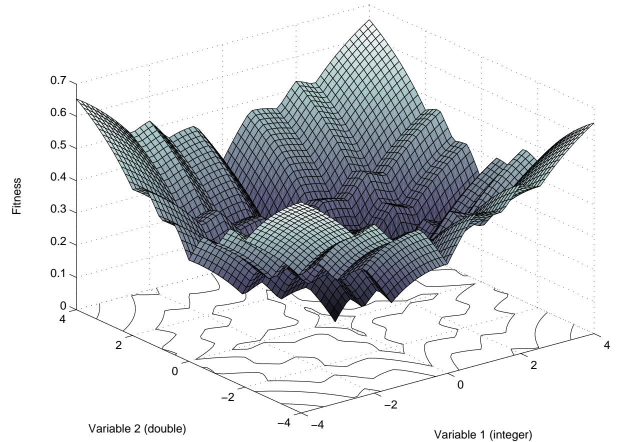

Figure 4: Fitness landscape of true branch of test object f12

Our assortment of industrial test objects is shown in table 4. These functions are taken for example from current

Test object / Function	Lines of code	No. of branches	No. of variables
BrakeAssistant1	405	108	33
BrakeAssistant2	405	108	33
ClassifyTriangle	36	26	8
De-icer1	171	56	12
De-icer2	79	18	12
EmergencyBrake	202	62	31
ICCDeterminer	60	30	7
LogicModule	259	70	38
Multimedia	861	264	66
Preprocessor	990	92	33
PrototypingTool	232	46	135
Reaction	310	96	776
Warning	404	100	13

Table 4: Industrial test objects

Mercedes and Chrysler development projects. They differ in complexity with regards to code length and both the numbers and types of variables. Both *BrakeAssistant1* and *BrakeAssistant2* are used for brake coordination in the brake assist system, whereas *ClassifyTriangle* is an often used test function, that performs a classification of triangles based on their side lengths. *De-icer1* and *De-icer2* are both intended to control the windshield defroster heating units. *EmergencyBrake* is a part of an emergency brake system. *ICCDeterminer* determines the reasons why the cruise control of the vehicle could not be activated. *LogicModule* is used for the definition of the current operational mode of the engine and *Multimedia* for the handling of peripheral equipment. *Preprocessor* realizes parts of the object preprocessing required for situation analysis, and *PrototypingTool* controls the interaction of new code with a prototype of the engine for testing purposes. *Reaction* supervises possible system modes and determines the expected reaction on emerging failures, and *Warning* manages warnings that occur.

4.4 Results

The convergence characteristics of the artificial test objects are shown in figure 5. The results for the test objects *f01*, *f04*, *f09*, *f13*, *f18* and *f22* have been omitted. These test objects exclusively use boolean variables and both GA and PSO found optimum solutions during initialization.

Considering the functions that use several parameters of the same type (functions *f02*, *f10*, *f14*, *f19* and *f23* for type integer and functions *f03*, *f11*, *f15*, *f20* and *f24* for type double), it becomes obvious that both techniques need more time with the more parameters there are. An analysis of these results in detail reveals that GA was unable to reach the desired global optimum within the given number of fitness function evaluations when it had to optimize more than one parameter. In contrast, PSO reached it when optimizing one, two and three parameters, regardless of the parameter type. Although both algorithms were unable to converge successfully when optimizing six or twelve parameters, PSO reached better solutions using the same number of iterations. The inclusion of artificially constructed local optima (functions *f05*, *f06*, *f07* and *f08*) yielded similar results.

For the functions that use a mixed parameter set including boolean, integer and double valued parameters (functions *f16*, *f21* and *f25*), PSO outperformed GA in all cases. Either it reached the global optima using less function evaluations or it yielded a better result after the expiration of permitted evaluations. Adding local optima to two simple mixed parameter functions (*f12* and *f17*) resulted in a slightly faster convergence of GA compared to PSO.

In 13 of the 19 cases shown, PSO outperformed GA, while in the remaining 6 cases GA outperformed PSO. However, while the difference is significant in 8 of the 13 cases in which PSO outperformed GA, there is no case in which the superiority of the GA is significant.

In general, GA features a slightly faster convergence for simple functions whereas PSO outperforms GA primarily for complex functions with big search spaces. Additionally, PSO achieved either an equal or a significantly better solution compared to GA for all artificial test objects.

Figure 6 presents the success rates of both GA and PSO for the industrial test objects. Success rate means the relative frequency of a successful optimization (covering test case found), averaged over all test goals (which correspond to the branches) of a test object. The figure shows that PSO was able to find a covering test case for more test goals than GA.

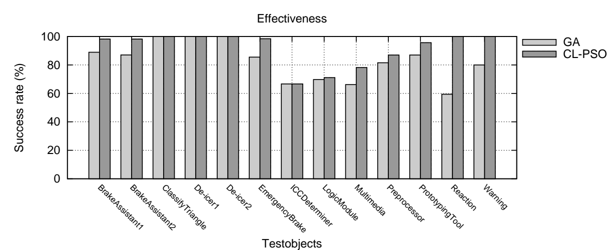

Figure 6: Experimental results for the effectiveness of GA and PSO regarding the industrial test objects

Figure 7 summarizes the efficiency of the two optimization strategies for the industrial test objects. It shows the number of fitness function evaluations required for finding a covering test case or for failing - averaged over the test goals of each test object. The figure also shows the estimated standard deviation. In general, GA needed more evaluations

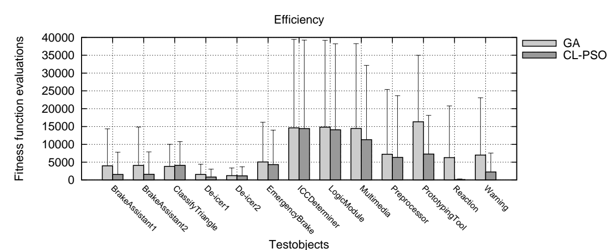

Figure 7: Experimental results for the efficiency of GA and PSO regarding the industrial test objects

than PSO for all of the test objects. The smaller estimated standard deviations of the results of PSO indicate that PSO delivers a more reliable result than GA does. The huge standard deviations are due to the averaging over each test goal of the test objects. Some of these test goals were very easy to reach whereas others could not be reached at all or needed more generations or iterations respectively. Although the average number of fitness function evaluations

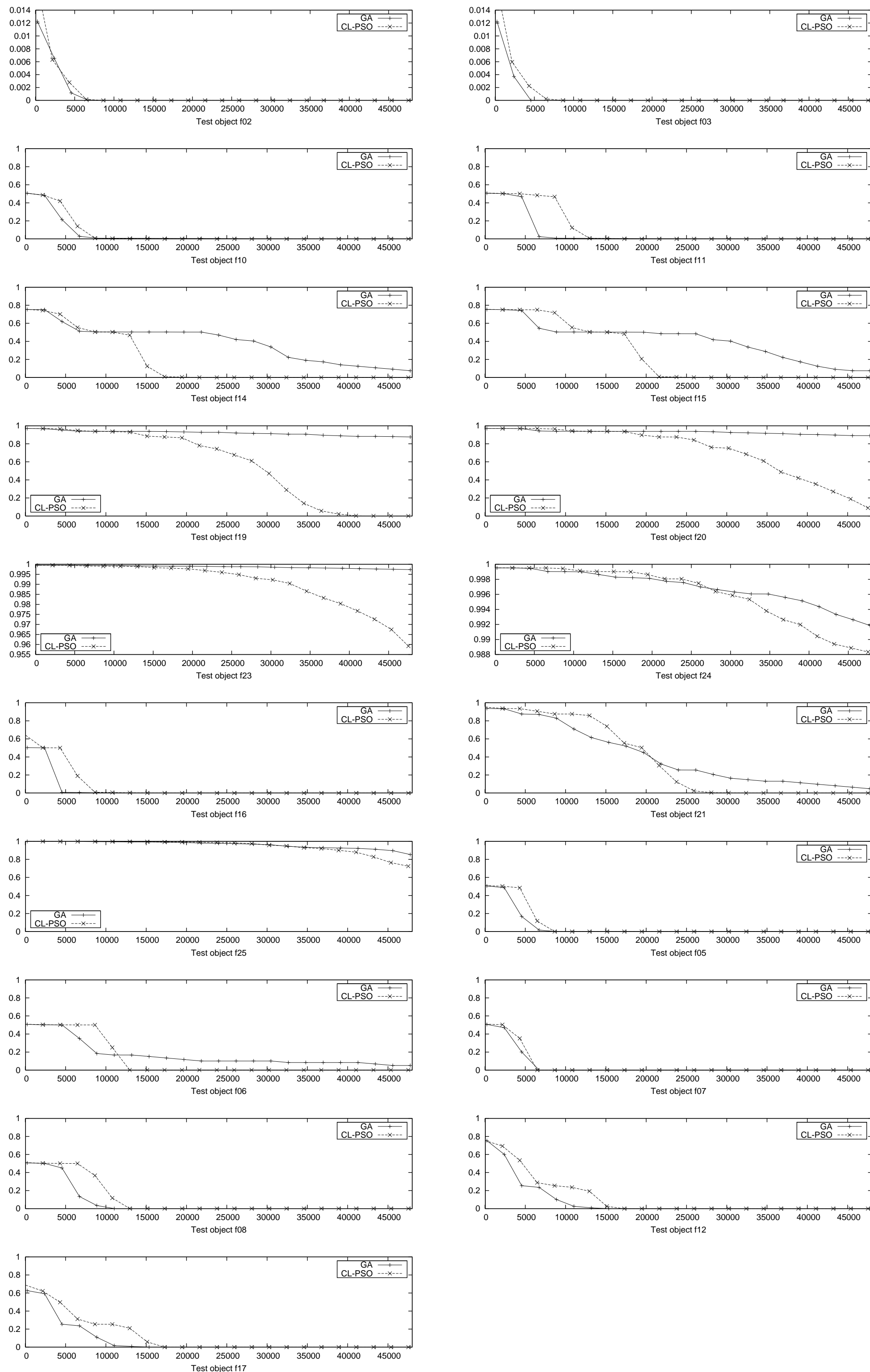

Figure 5: The convergence characteristics of GA and PSO regarding the artificial test objects: the x-axis shows the number of fitness function evaluations, the y-axis shows the best fitness value. Results are averaged over 30 runs. Note that the range of the y-axis may be different for different charts.

for PSO is smaller than for GA, it is worth mentioning that some branches could be covered faster by GA. In order to examine the statistical significance of the results, a t-test was performed for all the test goals of each test object. The outcome of this test indicates that the majority of the differences in efficiency between GAs and PSO was statistically significant. About 80% of the differences turned out to be statistically significant (confidence 95%). Overall 74% of the differences turned out to be even very significant (confidence 99%).

The obtained results lead to the conclusion that, when applied to evolutionary structural testing, PSO tends to outperform GA for complex functions in terms of effectiveness and efficiency.

5. CONCLUSION AND FUTURE WORK

This paper reported on the application of particle swarm optimization to structural software testing. We described the design of our test system that allows easy exchange of the employed search strategy. This system was used to carry out experiments with 25 artificial and 13 industrial test objects that exhibit different search space properties. Both particle swarm optimization and genetic algorithms were used to automatically generate test cases for the same set of test objects.

The results of the experiments show that particle swarm optimization is competitive with genetic algorithms and even outperforms them for complex cases. Even though the genetic algorithm yields a covering test case faster than particle swarm optimization in some cases, the latter is much faster than the genetic algorithm in the majority of the cases. This result indicates that particle swarm optimization is an attractive alternative to genetic algorithms since it is just as good as or even better than genetic algorithms in terms of effectiveness and efficiency, and is a much simpler algorithm with significantly fewer parameters that need to be adjusted by the user.

However, more experiments with further test objects taken from various application domains must be carried out in order to be able to make more general statements about the relative performance of particle swarm optimization and genetic algorithms when applied to software testing. Systematically varying the algorithm settings for the experiments would also help to draw more comprehensive conclusions. As an alternative comparison approach, a theoretical analysis of both optimization techniques in the context of software testing would give some more insights as to the suitability of either approach. These directions are all items for future work. The development of a testability meta-heuristic that selects the most promising search strategy for each test goal individually is especially interesting. Maybe software measures can help support this selection heuristic. Then, we would not only assume that genetic algorithms and particle swarm optimization are relevant search strategies, but would also broaden the spectrum of candidate search techniques, such as hill climbing, simulated annealing or even random search.

Acknowledgments

We would like to thank our colleague Harmen Sthamer for the thorough review of this paper and his valuable feedback. This work was funded by EU grant IST-33472 (EvoTest).

6. REFERENCES

[1] B. Clow and T. White. An evolutionary race: A comparison of genetic algorithms and particle swarm optimization for training neural networks. In *Proceedings of the International Conference on Artificial Intelligence, IC-AI '04, Volume 2*, pages 582–588. CSREA Press, 2004.

[2] R. C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the 6th International Symposium on Micromachine Human Science*, pages 39–43, 1995.

[3] Genetic and Evolutionary Algorithm Toolbox for use with Matlab. http://www.geatbx.com.

[4] R. Hassan, B. Cohanim, and O. de Weck. A comparison of particle swarm optimization and the genetic algorithm. In *Proceedings of the 46th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, 2005.

[5] R. J. W. Hodgson. Partical swarm optimization applied to the atomic cluster optimization problem. In *GECCO*, pages 68–73, 2002.

[6] T. Huang and A. S. Mohan. A hybrid boundary condition for robust particle swarm optimization. *IEEE Antennas and Wireless Propagation Letters*, 4:112–117, 2005.

[7] L. Oliva J. Horák, P. Chmela and Z. Raida. Global optimization of the dual-band planar antenna: Pso versus ga. In *Radioelektronika*, 2006.

[8] B. F. Jones, H. Sthamer, and D. E. Eyres. Automatic test data generation using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, September 1996.

[9] K. O. Jones. Comparison of genetic algorithm and particle swarm optimization. In *Proceedings of the International Conference on Computer Systems and Technologies*, 2005.

[10] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4. IEEE Press, 1995.

[11] J. J. Liang, A. K. Qin, P. N. Suganthan, and S. Baskar. Comprehensive learning particle swarm optimizer for global optimization of multimodal functions. *IEEE Transactions on Evolutionary Computation*, 10:281–295, 2006.

[12] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[13] J. Robinson and Y. Rahmat-Samii. Particle swarm optimization in electromagnetics. *IEEE Transactions on Antennas and Propagation*, 52:397–407, Feb. 2004.

[14] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(1):841–854, 2001.

[15] S. E. Xanthakis, C. C. Skourlas, and A.K. LeGall. Application of genetic algorithms to software testing. In *Proceedings of the 5th International Conference on Software Engineering and its Applications*, pages 625–636, 1992.