Using Simulated Annealing for Producing Software Architectures

Outi Räihä
Tampere University of Technology
P.O.Box 553
FIN-33101 Tampere, Finland
+358-50-5342813
outi.raiha@uta.fi

Erkki Mäkinen
University of Tampere
Department of Computer Sciences
FIN-33014 University of Tampere, Finland
em@cs.uta.fi

Timo Poranen
University of Tampere
Department of Computer Sciences
FIN-33014 University of Tampere, Finland
tp@cs.uta.fi

ABSTRACT
Automatic design of software architecture by use of genetic algorithms has already been shown to be feasible. A natural problem is to augment – if not replace – genetic algorithms with some other search method in the process of searching good architectures. The present paper studies the possibilities of simulated annealing in designing software architecture. We start from functional requirements given as a graph of functional responsibilities and consider two quality attributes, modifiability and efficiency. It is concluded that simulated annealing as such does not produce "natural" architectures, but it is useful as a method of producing initial populations for genetic algorithms.

Categories and Subject Descriptors
D.2.10. [**Software Engineering**]: *Design*; D.2.11. [**Software Engineering**]: *Software Architectures*

General Terms
Algorithms, Design, Experimentation

Keywords
Simulated annealing, search-based software design

1. INTRODUCTION
The ultimate goal of software engineering is to be able to automatically produce software systems based on their requirements. For the time being, we pass the synthesis of executable programs, and concentrate on the automated derivation of architectural designs of software systems. This is possible because architectural design largely means the application of known standard solutions in a combination that optimizes the quality properties (like modifiability and efficiency) of the software system. These standard solutions are well documented as architectural styles [14] and design patterns [5]. In addition, architectural design is guided by general principles like decomposition and usage of interfaces.

Genetic algorithms (GAs) are shown to be a feasible method for producing software architectures from functional requirements [12, 13]. It is then natural to ask if other search methods are capable of producing equally good architectures alone or in co-operation with genetic algorithms. The purpose of the present paper is to study the possibilities of simulated annealing in the process of searching good architectures when functional requirements are given.

Contrary to genetic algorithms, simulated annealing (SA) is a local search method which intensively uses the concept of neighborhood, i.e., the set of possible solutions that are near to the current solution. The neighborhood is defined via the transformations that change an element of the search space (in our application, software architecture) to another. In our application the transformations mean implementing a design pattern or an architectural style.

This paper proceeds as follows. In Section 2 we sketch current research in the field of search algorithms in software design that is relevant for the present paper. In Section 3 we cover the basics of implementing a simulated annealing algorithm. In Section 4 we introduce our method by defining the input for the SA algorithm, the transformations and the evaluation function. In Section 5 we present the results from our experiments related to an e-home application. In Section 6 we discuss the findings and in Section 7 we give a conclusion of our results.

2. RELATED WORK
Traditionally, search-based software engineering has focused on problems like software clustering and refactoring, see, e.g., [4, 6]. More recently, approaches dealing with higher level structural units, such as patterns, have gained more interest. For example, Amoui et al. [1] have applied genetic algorithms for finding the optimal sequence of design pattern transformations to increase the reusability of a software system.

Simons and Parmee [15, 16] take use cases as the starting point for system specification. Data is assigned to attributes and actions to methods, and a set of uses is defined between the two sets. The notion of class is used to group methods and attributes. Each class must contain at least one attribute and at least one method. Design solutions are encoded directly into an object-oriented programming language.

Räihä et al. [12] have taken a more advanced approach to designing software architecture than Simons and Parmee [15] by starting the design from a responsibility dependency graph and developing the architecture further than the class distribution of actions and data. A GA is used for the automation of design. In this solution, each responsibility is represented by a supergene and a chromosome is a collection of supergenes. The supergene contains information regarding the responsibility, such as dependencies of other responsibilities, and evaluated parameters such as execution time and variability. Mutations are implemented as adding or removing an architectural design pattern or an interface or splitting or joining class(es).

Räihä et al. [13] have also applied genetic algorithms in model transformations that can be understood as pattern-based refinements. In MDA (Model Driven Architecture), such transformations can be exploited for deriving a Platform Independent Model from a Computationally Independent Model. The approach uses design patterns as the basis of mutations and exploits various quality metrics for deriving a fitness function. They give a genetic representation of models and propose transformations for them. The results suggest that genetic algorithms provide a feasible vehicle for model transformations, leading to convergent and reasonably fast transformation process.

Other approaches to search-based software design are widely covered in a survey by Räihä [11].

3. SIMULATED ANNEALING

The SA algorithm starts from an initial solution which is tried to be improved during the annealing process by searching and selecting other solutions from the neighborhood of the current solution. There are several parameters that guide the annealing. The search begins with initial temperature t_0 and ends when the temperature t is decreased to the frozen temperature t_l, where $0 \le t_l \le t_0$. The temperature gives the probability of choosing solutions that make the current solution worse. A solution that worsens the current solution by δ, is accepted to be the new current solution if a randomly generated real i is less than or equal to $e^{-\delta/t}$. If a solution improves the current solution, it is accepted directly without a test. A constant r is used to determine when temperature is decreased by multiplying the current temperature by a cooling ratio α, where $0 \le \alpha \le 1$.

The SA has been successfully applied for numerous combinatorial optimization problems, for an instructive introduction to the use of simulated annealing as a tool for experimental algorithmics, see [7, 2]. The idea for combinatorial optimization is generalized by Kirkpatrick et al. [8]. To determine good parameters for a problem needs often experimental analysis. There are also adaptive techniques for detecting the parameters [9].

4. METHOD

We begin with a set of responsibilities (requirements) that can be given some relative values regarding modifiability and efficiency. Using the information given on the dependencies between the responsibilities, this set is then formed into a responsibility dependency graph. The graph is encoded as to a form that can be processed by the selected search algorithm (implemented with Java). The algorithm produces software architecture for the given requirements by implementing selected architecture styles and design pattern, and produces a UML class diagram as the result.

4.1 Responsibility Dependency Graph

A responsibility dependency graph gives the functional requirements of the system in terms of responsibilities. A responsibility is either a task to be carried out by the system (or some part of it), or a data item that has to be managed by the system (or some part of it). Each node in the graph represents a responsibility, and each directed edge represents a dependency between the two responsibilities. Here a dependency means that the source responsibility relies on the target responsibility. In order to evaluate the system after the model has been subjected to transformation, some attributes of the responsibilities are also given, such as variability, parameter size and time consumption. The values for these attributes may be sometimes hard to determine at the requirements analysis stage, but the more accurately these can be estimated, the better will be the result. The given values for the attributes are relative, rather than absolute.

In our work, we have used an e-home [10] as an example system. It contains 52 responsibilities and 90 dependencies between them. A part of the responsibility dependency graph, depicting the drape control component of the example system, is given in Figure 1, where some sample properties (variability, parameter size, time consumption) are marked in the nodes. The drapeState node is marked with a thicker circle, as it is a data manager responsibility. The CalculateOptimalDrape responsibility is a good example of how and why the certain attributes are evaluated: its variability is 3, as the optimal drape position can be calculated differently in different houses and according to different preferences. As it is a heavy operation, also its parameter size (6) and time consumption (90) are among the highest values of those shown here. Responsibilities with such high attribute values play an important role when constructing quality architecture as their placement has a bigger impact on the quality value.

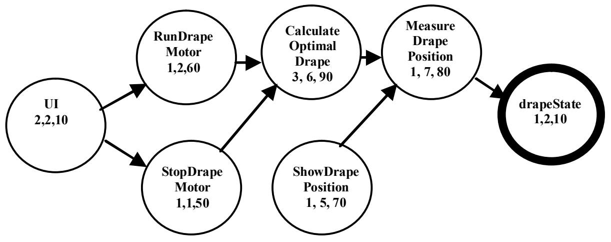
UI 2,2,10
RunDrape Motor 1,2,60
StopDrape Motor 1,1,50
Calculate Optimal Drape 3, 6, 90
ShowDrape Position 1, 5, 70
Measure Drape Position 1, 7, 80
drapeState 1,2,10

Figure 1. A fragment of the responsibility dependency graph

4.2 Transformations

An architecture is transformed (i.e., one of its neighbors is found) by implementing architecture styles and design patterns to a given solution. In our approach, decomposition is also regarded as an architectural pattern. The patterns we have chosen include very high-level architectural styles [14] (dispatcher and client-server), medium-level design patterns [5] (façade and mediator), and low-level design patterns [5] (strategy and proxy). The transformations are implemented in pairs of introducing or removing a pattern. This ensures a wider traverse through the search space, as while implementing a pattern might improve the quality of architecture at one point, it might become redundant over the course of development. The dispatcher architecture style makes a small exception to this rule: the actual dispatcher must first be introduced to the system, after which the responsibilities can communicate through it. The transformations are the following:

- split a class /merge two classes
- introduce/remove message dispatcher
- introduce/remove link to dispatcher
- introduce/remove server
- introduce/remove façade
- introduce/remove mediator
- introduce/remove strategy
- introduce/remove proxy.

However, it is possible that during a transformation, an existing pattern may be broken. Because of this, after each transformation, the resulting solution is subjected to a corrective operation, which ensures that the architecture stays coherent. In addition to ensuring that the patterns present in the system stay coherent and "legal", the corrective function also checks that the model conforms to certain architectural laws. These laws demand uniform calls between two classes (e.g., through an interface or a dispatcher), and state some basic rules regarding architectures, e.g., no responsibility can implement more than one interface. These laws ensure that no anomalies are brought to the architecture.

4.3 Evaluation

Selecting an appropriate evaluation function is probably the most demanding task with any search algorithm application when there is no clear value to measure in the solutions. In the case of software architecture, evaluation is especially difficult. In real world, evaluation of software architecture is almost always done manually by human designers, and metric calculations are only used as guidelines. However, for a search algorithm to be able to evaluate the architecture, a purely numerical fitness value must be calculated.

In a fully automated approach, no human interception is allowed, and thus the evaluation function needs to be based on metrics. The selection of metrics may be as arguable as the evaluations of two different architects on a single software architecture. The reasoning behind the selected metrics in this approach is that they have been widely used and recognized to accurately measure some quality aspects of software architecture. However, the combination of metrics and multiple optimization is another problem entirely, as not many quality values can be optimized simultaneously.

The fitness function is based on software product metrics, most of which are from the metrics suite introduced by Chidamber and Kemerer [3]. These metrics, especially coupling and cohesion, have been used as a starting point for the fitness function, and have been further developed and grouped to achieve clear "sub-functions" for modifiability and efficiency, both of which are measured with a positive and negative metric. The biggest modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-server architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. A complexity metric is added to penalize having many classes and interfaces as well as extremely large classes.

Dividing the evaluation function into sub-functions answers the demands of the real world. Hardly any architecture can be optimized from all quality viewpoints, but some viewpoints are ranked higher than others, depending on the demands regarding the architecture. By separating efficiency and modifiability, which are especially difficult to optimize simultaneously, we can assign a bigger weight to the more desired quality aspect. When w_i is the weight for the respective sub-function sf_i, the evaluation function $f(x)$ for solution x can be expressed as

$$f(x) = w_1 * sf_1 - w_2 * sf_2 + w_3 * sf_3 - w_4 * sf_4 - w_5 * sf_5.$$

Here, sf_1 measures positive modifiability, sf_2 negative modifiability, sf_3 positive efficiency, sf_4 negative efficiency and sf_5 complexity. The sub-functions are defined as follows (|X| denotes the cardinality of X):

sf_1 = |interface implementors| + |calls to interfaces| + (|calls through dispatcher| $*$ $\sum$ (variabilities of responsibilities called through dispatcher)) – |unused responsibilities in interfaces| $*$ 10,

sf_2 = |calls between responsibilities in different classes, that do not happen through a pattern|,

sf_3 = $\sum$ (|dependingResponsibilities within same class| $*$ parameterSize + $\sum$(|usedResponsibilities in same class| $*$ parameterSize + |dependingResponsibilities in same class| $*$ parameterSize)),

sf_4 = $\sum$ ClassInstabilities + (|dispatcherCalls| + |serverCalls|)$*$ $\sum$ callCosts, and

sf_5 = |classes| + |interfaces|.

The multiplier 10 in sf_1 means that having unused responsibilities in an interface is almost an architecture law, and should be more heavily penalized.

5. EXPERIMENTS

In this section we present the results from the preliminary experiments done with our approach, using the example system introduced in subsection 4.1. The calculated quality value for each curve is the average value from five test runs. The SA algorithm was run with different starting temperatures and cooling ratios to determine what kind of effect they would have on the quality of the produced architecture and the actual UML diagrams. All tests were made with the constant *r* set to 20. The weights for all sub-functions of the quality evaluation function were set to the same (i.e., all weights w_i were set to 10). The run time for tests with a starting temperature of 7500 and cooling factor of 0.05 was approximately 15 minutes. The "standard" starting temperature was chosen to be 7500 with which the tests with different ratios were run and the "standard" cooling ratio was chosen to be 0.05 with which tests with different start temperatures were run.

Figure 2 depicts the quality curve of the "standard" test run with starting temperature 7500 and cooling ratio 0.05. As can be seen, the development of quality is quite slow in the beginning, but develops rapidly after the temperature cools below 200. The quality function also achieves very high values.

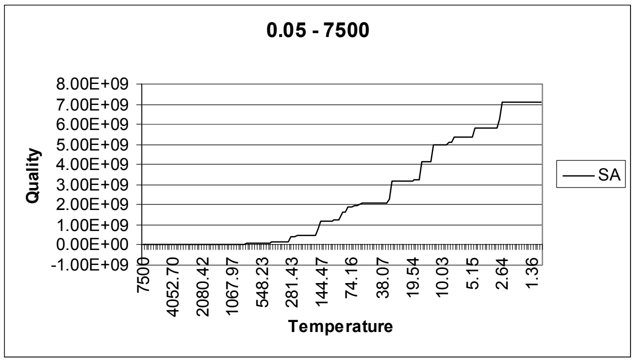

Figure 2. SA curve for slow annealing

In Figure 3 the cooling ratio was increased to 0.15, thus resulting in faster annealing. In this case the development of the quality curve is almost non-existent until the temperature cools down to about 100, after which the curve begins to develop, and there is an exceptionally rapid increase in the quality values after the temperature cools down to about 5. The tests with cooling factor 0.15 took approximately 1-2 minutes.

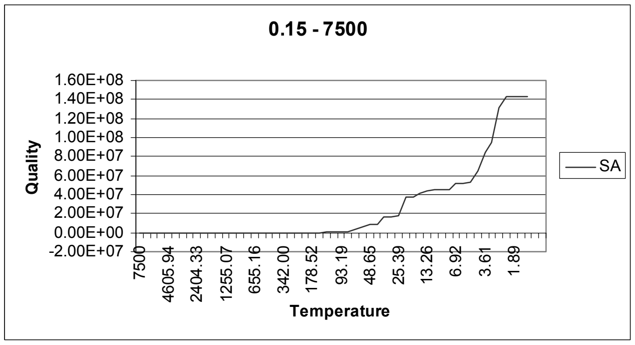

Figure 3. SA curve for fast annealing

Different starting temperatures with cooling factor 0.05 were also tested, and the quality curves for temperatures 10000, 7500 and 5000 are shown in Figure 4. As can be seen, the quality curves for all start temperatures end quite close to one another, and based on this there does not seem to be a clear advantage to start from an exceptionally high temperature, as similar end quality value can be achieved with lower temperatures.

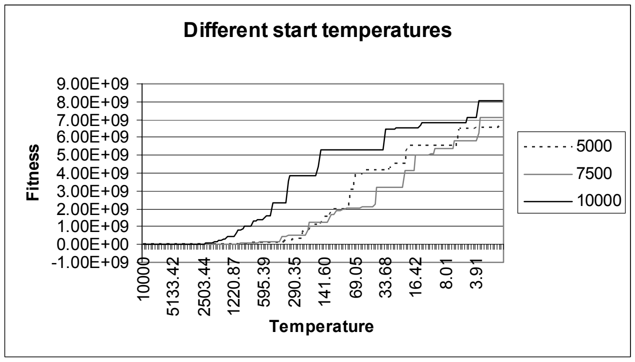

Figure 4. SA curves for different starting temperatures

In addition to examining only SA, we combined the SA algorithm to our previous implementation of GA [13]. Notice that the test runs depicted in Figures 2-4 use the same fitness function as the genetic algorithm used in the rest of the paper (and the same used in our previous tests [11, 12]). First a test was run where the best individual achieved with the GA was given as a base for the SA. GA was run with a population size 100 and 250 generations. As seen in Figure 5, the quality curve of the GA portion of the GA-SA test run achieves very low quality values compared to all the test runs made with the SA.

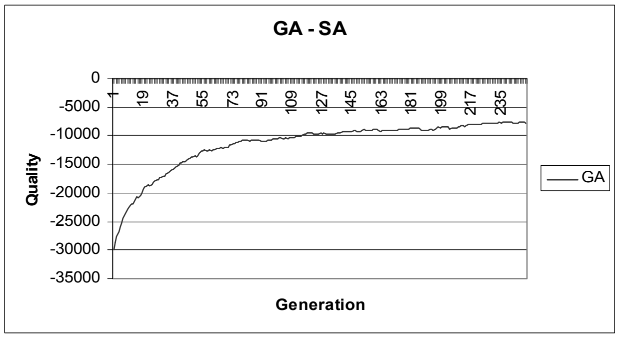

Figure 5. GA curve to be used as a base for SA

Notice that the initial quality is actually same for the GA as for the SA tests, although the initial fitness values of the SA tests seem to begin closer to zero. This is due to the fact that the SA reaches such high quality values that the ranges for the curves are quite different, and small differences can not be seen in the SA graphs.

The SA was run with a cooling factor 0.15 and a starting temperature of 2500. The fast annealing and low starting temperature were chosen as the idea was that the GA should have already done some basic work with achieving a quality architecture, and thus there should not be a need to give the SA algorithm excessive time to operate with the architecture. Figure 6 shows the quality curve for SA portion of the GA-SA test. As can be seen, the quality values stay very low for quite a long time, and only start to noticeably increase after the temperature has cooled down to approximately 30. After this the quality values start to significantly increase, but the end values are much lower than those achieved with plain SA. The runtime of the GA was approximately 1-2 minutes, and as runtime of the SA another minute, this experiment was quite fast.

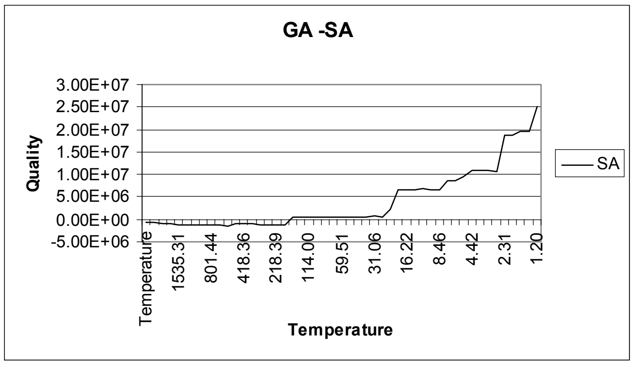

Figure 6. SA curve when best result from the GA used as a base solution

Finally, tests were made were the result of a SA run was used as a base for the start population given for the GA. Again, the cooling factor for the SA was set to 0.15 and the starting temperature at 2500, as now the genetic algorithm would continue to develop and it was not necessary to even attempt a near-perfect solution only through the SA approach.

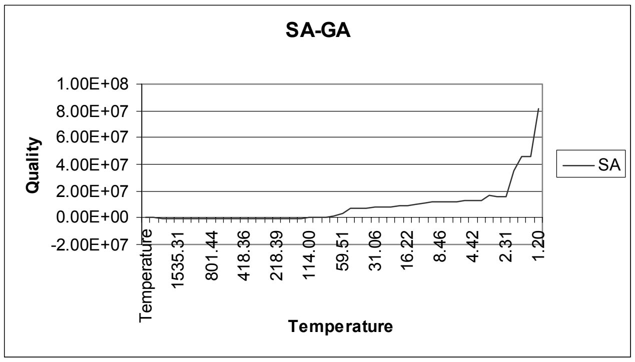

Figure 7. SA curve to be used as a base for GA

Figure 7 shows the quality curve for the SA portion of the test runs. As can be seen, there is rapid development at a very late stage, but the quality values stay relatively low compared to other simulated annealing runs made with slower annealing or higher starting temperatures. However, the quality values are already higher at this point than in Figure 6 with the approach where the GA was used as a base for the simulated annealing.

Figure 8 shows the curve of the GA portion of the SA-GA test. The quality values start to develop immediately, and seem to reach an optimum quite early, only after some 75 generations. The quality values are also higher than in any other test done with either GA or simulated annealing. This suggests that using simulated annealing as a basis for genetic algorithm would be an exceptionally promising branch of research to continue in terms of software architecture design. The drawback to this approach is, however, that the runtime for each test was the highest of all, as each run took over an hour.

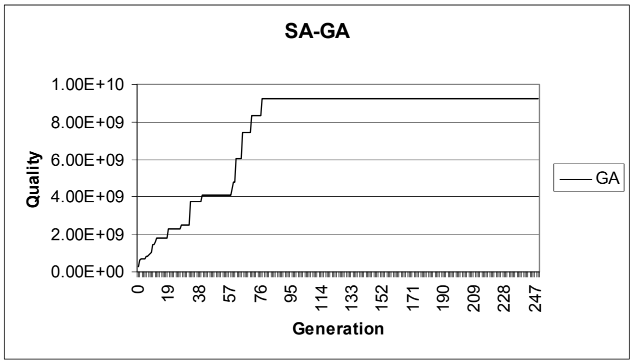

Figure 8. GA curve when the SA result is used as a base for the start population.

6. DISCUSSION

In Section 5 we presented the quality curves of different experiments made with the simulated annealing algorithm. However, as discussed in Section 4.3, evaluating software architecture is extremely difficult, and even more so when the evaluation must rely on raw numbers. Thus, the actual UML graphs given as output should also be examined to get a wholesome idea of whether the results with extreme quality values are actually good. The evaluation of these graphs is bound to be subjective, but some general guidelines were used as to whether to consider one design better than the other. Firstly, use of patterns should be "sensible". Secondly, there should not be any "god classes", nor should each responsibility be in a class of its own, but some sensible grouping should be seen. Finally, and most importantly, the dependencies between classes must be "structured". Failing in the first two points will definitely result in an overall messy graph, if the dispatcher is used without caution, there are patterns used everywhere and each responsibility is in a class of its own. This will produce a graph that will also fail in the third point and will be extremely unpleasant to interpret.

When examining the graphs achieved by using only simulated annealing, it could be seen that the high quality values were achieved by excessive use of the dispatcher architecture style. Especially in the test runs were the starting temperature was 10000 (and the cooling factor 0.05), nearly every dependency between the responsibilities was communicated through the message dispatcher, the responsibilities were nearly all in their own classes, and there was heavy use of the proxy and strategy patterns. With lower starting temperatures the use of the dispatcher was somewhat more restricted, and in the tests with a temperature of 5000, some bigger classes could already be seen, which clarified the graph to some extent as there were not as many dependencies. As for the different cooling factors, slower annealing seems to favor the message dispatcher style, while the tests with a larger cooling factor (i.e., faster annealing) favored the strategy and proxy patterns more. The dispatcher was also used, but not as much, and the relative amount of the other patterns was bigger.

The graphs achieved with using a GA base and continuing to develop that solution with the simulated annealing were inconsistent in quality. Some graphs were extremely "messy", while a few contained some structure. This could be seen as a result from the GA solution containing larger classes, as was seen in results obtained by using only GA. This also reflects the fundamental problem of using software metrics: although the used metrics, modifiability and efficiency, gained steadily improving values as measured by the fitness function, the overall "goodness" of the solutions as evaluated by a human observer did not improved accordingly. That is, the metrics do not always catch something essential in the architectures.

Finally, the approach using SA as a base for GA, which gave the highest quality values, also performed the best from all the approaches using SA. Using the GA to refine the solution achieved by the SA had a "structuring" effect on the architecture. Responsibilities were better grouped in classes, the proxy and strategy patterns that were used were clear, and although the message dispatcher was still very central in the solutions, the connections through it were more "structured", and the overall appearance of the UML graph was noticeably better than in the results achieved with plain simulated annealing.

7. CONCLUSIONS

We have presented an approach that uses SA in software architecture design. A responsibility dependency graph is given as input and architecture styles and design patterns are used as transformations when searching for a better solution in the neighborhood. The solution is evaluated with regard to quality and efficiency. The experimental results achieved with this approach show that although extremely high quality values are achieved with this approach, their "true" quality as evaluated by examining the UML class diagrams is not actually as good. However, when combining the solution achieved with SA with a GA implementation, the actual quality of the produced solutions increases as well as the calculated metric values. This would suggest that further work should be done with studying the combination of these two algorithms in software architecture design. Studying the definition of evaluation functions for simulated annealing and genetic algorithm should be done as well, as using the same function apparently gives quite different types of solutions when using the different algorithms.

Our future work attend to these questions as well as deriving real test cases to further evaluate the approach, and adding more design patterns to cover a larger search space of possible architectures. We also plan to implement a multi-objective fitness function primarily for the GA implementation.

8. REFERENCES

[1] Amoui, M., Mirarab, S., Ansari, S., and Lucas, C. A genetic algorithm approach to design evolution using design pattern transformation, *International Journal of Information Technology and Intelligent Computing* 1, 2006, pp. 235-245.

[2] Aragon, C.R., Johnson, D.S., McGeoch, L.A., and Schevon, C. Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning, *Operations Research*, 39(3), 1991, pp. 378-406.

[3] Chidamber, S.R., and Kemerer, C.F. A metrics suite for object oriented design, *IEEE Transactions on Software Engineering* 20(6), 1994, pp. 476-492.

[4] Clarke, J., Dolado, J.J., Harman, M., Hierons, R. M., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., and Shepperd, M. Reformulating Software Engineering as a Search Problem, *IEE Proceedings - Software, 150 (3)*, 2003, pp. 161-175.

[5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley. 1995.

[6] Harman, M., Hierons, R., and Proctor, M. A new representation and crossover operator for search-based optimization of software modularization. In: *Proc. of GECCO'02*, 2002,pp. 1351–1358.

[7] Johnson, D. S., Aragon, C.R., McGeoch, L. A., and Schevon, C. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning, *Operations Research*, 37(6), 1989, pp. 865-892.

[8] Kirkpatrick, S., Gelatt, C., and Vecchi, M. Optimization by simulated annealing, *Science*, 220, 1983, pp. 671-680.

[9] van Laarhoven, P. J. M., and Aarts, E. *Simulated Annealing: Theory and Applications*, Kluwer, 1987.

[10] Räihä, O. Evolutionary Software Architecture Design, University of Tampere, Department of Computer Sciences, Report D-2008-11, 2008.

[11] Räihä, O. A survey on search-based software design. University of Tampere, Department of Computer Sciences, Report D-2009-1, 2009.

[12] Räihä, O., Koskimies, K., and Mäkinen, E. Genetic Synthesis of Software Architecture, In: *Proc. of SEAL'08*, 2008, Springer LNCS 5361, pp. 565-574.

[13] Räihä, O., Koskimies, K., Mäkinen, E., and Systä, T. Pattern-Based Genetic Model Refinements in MDA, In: *Proceedings of the Nordic Workshop on Model-Driven Engineering (NW-MoDE'08),* Reykjavik, Iceland. University of Iceland, 2008, pp. 129-144.

[14] Shaw, M. and Garlan, D. *Software Architecture - Perspectives on an Emerging Discipline,* Prentice Hall, 1996.

[15] Simons, C.L. and Parmee, I.C. Single and multi-objective genetic operators in object-oriented conceptual software design. In: *Proc. of GECCO'07*, 2007, pp. 1957-1958.

[16] Simons, C.L. and Parmee, I.C. A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design, In: *Engineering Optimization* 39 (5) 2007, pp. 631-648.