Outi Räihä, Kai Koskimies and
Erkki Mäkinen

Genetic Synthesis of Software Architecture

DEPARTMENT OF COMPUTER SCIENCES
UNIVERSITY OF TAMPERE

D-2008-4

TAMPERE 2008

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCES
SERIES OF PUBLICATIONS D – NET PUBLICATIONS
D-2008-4, MAY 2008

Outi Räihä, Kai Koskimies and Erkki Mäkinen

Genetic Synthesis of Software Architecture

DEPARTMENT OF COMPUTER SCIENCES
FIN-33014 UNIVERSITY OF TAMPERE

ISBN 978-951-44-7358-6
ISSN 1795-4274

Genetic Synthesis of Software Architecture

Outi Räihä*, Kai Koskimies** and Erkki Mäkinen*
**University of Tampere, Finland*
***Tampere University of Technology, Finland*
outi.raiha@uta.fi

Abstract

Design of software architecture is intellectually one of the most demanding tasks in software engineering. This paper proposes an approach to automatically synthesize software architecture using genetic algorithms. The technique applies architectural patterns for mutations and quality metrics for evaluating individual architectures, producing a proposal for a software architecture on the basis of functional requirements given as a graph of functional responsibilities. Two quality attributes are considered, modifiability and efficiency. The behavior of the genetic synthesis process is analyzed with respect to quality improvement speed, the effect of population size, the effect of dynamic mutation, and the effect of quality attribute prioritization. It is concluded that genetic architecture synthesis in the proposed form is a converging process that is able to produce reasonable architectural solutions, although fully satisfactory architectures have not been synthesized in our tests for an example system.

1. Introduction

A persistent dream of software engineering is to be able to automatically produce software systems based on their requirements. While this can be accomplished in narrow domains, in a general sense it is still out of reach for the current technology. Fully automated software synthesis succeeds in narrow domains because the intellectually difficult parts, the architecture design and the transformation of functional requirements into code, have predetermined domain-specific solutions.

In this paper we study the problem of automated synthesis of software architecture in a domain-independent manner, given some representation of functional and quality requirements. We anticipate that this kind of technology can be exploited in various contexts in software engineering. Obviously, automated architecture synthesis can be used to produce initial or alternative architecture designs in a traditional software engineering process, to support the architect who is still responsible for the final result. On the other hand, if the technology can be made reliable enough, automated architecture synthesis can be exploited for example in MDA (Model-Driven Architecture) approaches to support the mapping of models from problem level to solution level, or in self-adaptive systems [17] to assist in the self-reconfiguration triggered by changes in the environment.

We argue that automated software architecture synthesis is conceivable because a lot of architectural knowledge exists in the form of architectural styles, reference architectures, design patterns, best practices etc. These recommended general solutions, called here collectively *architectural patterns*, typically promote some quality attributes of the system, and possibly weaken others. For example, the use of the message dispatcher architectural style in the communication of components increases the modifiability of the system, but weakens its efficiency.

In addition to architectural patterns, there are a number of rules concerning architecture design that are assumed to be followed in any architecture because of the consistency and cleanness of the design. For example, if two components communicate through a message dispatcher, it does not make sense to allow them direct communication, because that would invalidate the idea of using the message dispatcher. Here we call such rules *architectural laws*.

For the purposes of this paper, the essence of software architecture design is to find a combination of instances of architectural patterns, without breaking any architectural laws, in such a way that the functional requirements hold with optimal quality properties. Here we will focus on two quality attributes, modifiability and efficiency.

Viewed in this way, software architecture synthesis can be seen as a combinatorial problem: given a set of architectural patterns, how to find an optimal configuration of such patterns for a non-trivial system? Typically, good architects apply their prior experiences

about successful applications of architectural patterns, and often come up with a reasonable architecture design after some iterations. However, in automated architecture synthesis this kind of intuition is not available, and an optimal solution must be found mechanically. On the other hand, a mechanical process is also free of prejudices, being able to generate viable solutions the human architect could never think of.

In this paper we study the application of genetic algorithms [14] to software architecture synthesis. Genetic algorithms are a popular heuristic search method that has been successfully applied to problems somewhat similar to ours, see Chapter 2. Architectural patterns provide a natural interpretation for mutations: a mutation can be realized as either the application or removal of an architectural pattern. Fitness function (that is, the "goodness" of an individual) can be expressed in terms of quality metrics, and crossover operation can be realized by merging two architectures without breaking existing pattern instances. Our focus is on developing the required techniques for genetic architecture synthesis, and on the investigation of the overall behavior of the genetic architecture synthesis process. We are also interested to analyze individual architecture proposals produced by the method. The proposed architecture is produced as a UML class diagram with (possibly stereotyped) classes, interfaces and their mutual dependencies.

The main contributions of this work are a setup for genetic pattern-based software architecture synthesis and experimental analysis of the behavior of the genetic synthesis process. The former includes an approach to represent functional requirements as a responsibility graph, techniques for representing architectural information as genes, for computing quality based fitness, and for architectural crossover and dynamic pattern-based mutation, and a demonstration of the genetic synthesis using exemplary sets of architectural patterns and laws. The latter includes an analysis of the quality improvement speed, the effect of population size, the effect of dynamic mutations, and the effect of prioritized quality attributes.

We proceed as follows: In the following chapter we briefly review genetic algorithms and existing approaches to apply them in problems related to software architecture problems. In Chapter 3 we present our approach to realize genetic software architecture synthesis. In Chapter 4 we analyze the results of our experiments. Chapter 5 contains a discussion of the character of the architectures proposed by genetic synthesis, on the basis of the experiments run on a test case. Finally, we conclude with some remarks on future work in Chapter 6.

2. Applications of genetic algorithms in software structuring

2.1 Genetic algorithms

Genetic algorithms are used to find a "good" solution from a very large search space. To operate with a genetic algorithm, one needs an encoding of the solution, i.e., a representation of the solution in a form that can be interpreted as a chromosome, an initial population, mutation and crossover operators, a fitness function and a selection operator for choosing the survivors for the next generation. The actual implementations of these fundamental structures vary a lot from an application to another. However, the key idea maintains the same: better individuals (solutions) have greater possibilities to reproduce, while worse solutions have greater possibilities to die and to be replaced by new individuals. It is believed that this process leads to a combination of the properties of the better individuals, which constitutes a good solution to the problem in question. We assume that the reader is familiar with the basics of genetic algorithms, as given, e.g., in [14].

2.2 Software clustering and systems integration

The goal of software clustering or module clustering is to find the best grouping of components to subsystems in an existing software system. The problem is to partition the graph so that the clusters represent meaningful subsystems.

The genetic algorithm presented by Clarke et al. [6] for the clustering problem is quite straightforward: the main challenge is to find a suitable encoding, after which traditional mutation and crossover operators are used. Defining these operations is, however, not so simple. Clarke et al. [6] introduce several cases where a hill-climbing algorithm has outperformed genetic algorithms, and the blame is usually placed with the encoding and crossover used with the genetic algorithm.

Doval et al. [8] have used a genetic algorithm approach for the optimization of the module clustering problem.

Harman et al. [11] approach the clustering problem from a re-engineering point of view: after maintaining a system its modularization might not be as good as it was when it was taken to use. Harman et al. define their problem as searching the space of possible modularizations around the current granularity to see if there exists a better allocation for the components.

Di Penta et al. [7] introduce the Software Renovation Framework (SRF) that attempts to remove unused

objects and code clones and to refactor existing libraries into smaller, more cohesive clusters. Genetic algorithms have been used especially to help with refactoring.

Seng et al. [19] represent the system as a graph, where the nodes are either subsystems or classes, and edges represent containment relations (between subsystems or a subsystem and a class) or dependencies (between classes). In this application each gene represents a subsystem, and each subsystem is an element of the power set of classes.

Systems integration is in a way quite similar to module clustering but with known modules. The problem is to decide the order in which they are incorporated to the system. The order of integration of components can be presented as a permutation of the set of components [6]. Le Hanh et al. [13] present a similar solution to the integration testing problem.

2.3 Systems refactoring

Systems refactoring is a somewhat more delicate problem than module clustering. When refactoring a system, there is the risk of changing the behavior of a system by, e.g., moving methods from a subclass to an upper class [20]. Hence, the refactoring operations should always be designed so that no illegal solutions will be generated or a corrective operation is used to check that the systems behavior stays the same.

O'Keeffe and Ó Cinneide [16] define the refactoring problem as a combinatorial optimization problem: how to optimize the weighting of different software metrics in order to achieve refactorings that truly improve the system's quality. Seng et al. [20] have a similar approach as O'Keeffe and Ó Cinneide [15], as they attempt to improve the class structure of a system by moving attributes and methods and creating and collapsing classes.

O'Keeffe and Ó Cinneide [16] have continued their research with the use of the representation and mutation and crossover operators introduced by Seng et al. [20]. O'Keeffe and Ó Cinneide [16] also compared the genetic algorithm to some other search algorithms.

Harman and Tratt [12] introduce a more user-centered method of applying refactoring. They offer the user the option to choose from several solutions produced by the search algorithm, and also point out that the user should be able to limit the kind of solutions she wants to see.

2.4 Architectural improvement

Architectural transformations apply bigger modifications to the system than simple refactoring operations. An example of architectural transformation is the introduction of design patterns in the architecture.

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. Similar studies are also performed by Grunske [10].

In addition to design-related software engineering problems, there are several other fields of software engineering where heuristic search algorithms have been successfully implemented. However, we do not survey these topics here.

2.5 Relationships to our work

Our work is similar to [1] in that we use high-level structural units, patterns, as the basis of mutations in a genetic process. We have also applied the supergene idea of [1], to be discussed in Chapter 3, as a starting point for representing the architecture. However, there are several differences. First, we consider not only reusability (or modifiability) as the quality criteria, but in principle we are interested in the overall quality of the architecture. In this paper we focus on two quality attributes, efficiency and modifiability.

Second, we aim at the synthesis of the architecture starting from requirement-level information, rather than at improving an existing architecture. Third, we do not restrict to design patterns, but consider more generally various kinds of architectural solutions at different levels.

Our viewpoint is different from that of system clustering and refactoring. System clustering considers software architecture only from the decomposition perspective, and software refactoring aims at structural fine-tuning of software architecture, whereas our approach strives for automating the entire architecture design process.

3. Genetic architecture synthesis

3.1 Functional requirements

A major problem in automated software architecture synthesis is the representation of functional requirements. Since the technique should be applicable in any domain, we cannot make assumptions about the actual semantics of the functional requirements. Yet, although software architecture design is usually driven by quality requirements rather than by functional requirements (e.g. [3]), the architecture is senseless without functionality. We have adopted here an approach where functional requirements are represented as a

graph of named functional responsibilities. These responsibilities remain as elements of the architecture, although they carry no semantics as far as the architecture synthesis is concerned.

Our approach stems from an old idea related to object-oriented design, CRC cards [2] (Class-Responsibility-Collaboration) originally proposed by Ward Cunningham for teaching object-oriented design. A CRC card contains three parts: the name of a class, the responsibilities of that class, and the collaborators of the class supporting the responsibilities. CRC cards help to find and collect the responsibilities associated with classes, together with the required collaborators (other classes). As a result of a design session based on CRC cards, use cases are refined into a rudimentary class structure, where classes host informal responsibilities rather than concrete methods.

We adopt the CRC idea of refining use cases (representing functional requirements) into responsibilities: the input for architectural synthesis consists of responsibilities. The same responsibilities will appear in the architecture proposals, assigned to interfaces and classes. Responsibilities do not necessarily become actual operations in the detailed design, but we argue that they express the functional aspect of the system at an appropriate level for architectural description.

However, in contrast to CRC design, we assume that each use case is considered only in terms of the responsibilities required to fulfill the use case, without thinking of classes. As in CRC design, we also identify the dependencies, not between classes but between responsibilities. That is, if some responsibility needs another responsibility, the former depends on the latter. A viable architecture must respect the dependencies in the sense that a component whose responsibility depends on a responsibility of another component must be linked to the latter component, and a link can exist only because of that. In addition, we use a special kind of responsibility, a data manager responsibility, for a data entity that is needed in a use case.

To allow the evaluation of the quality (that is, efficiency and modifiability) of the architecture, the responsibilities can be associated with values characterizing the assumed size of the parameter data needed by the responsibility, the assumed time consumption of the responsibility, and the assumed variability factor of the responsibility (the greater the factor, the more prone the responsibility is to change). Naturally, in many cases these are difficult to estimate in the early phases of the software development process, but for the sake of successful architecture synthesis, at least the most obvious and significant responsibility characteristics should be given in order to correctly evaluate the quality of proposed architectures.

In this work we have used an intelligent home system as a case study. Such a system provides an infrastructure and interfaces for controlling various home devices, like lights, drapes, and audio. A fragment of the responsibility graph given as input for the genetic architecture synthesis of this system is depicted in Figure 1, where the dependencies between and names of responsibilities are shown, as well as property values for variability factor, parameter size and time consumption (in this order). The drapeState responsibility is a data responsibility, marked with thicker line. In the middle of the graph is the responsibility CalculateOptimalDrape, which has a variability of 3, as the optimal drape position can be computed differently in different types of homes. It has a parameter size 6, indicating that it needs relatively large parameter set. Its call cost is 90, showing that it is a heavier operation than, e.g., RunDrapeMotor with a cost of 60. All property values are relative rather than absolute. The entire responsibility set for this system contains 42 functional responsibilities, 10 data responsibilities and 90 dependencies between them.

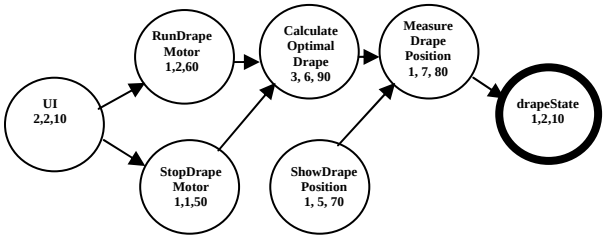

Figure 1. Fragment of a responsibility dependency graph

A responsibility graph is assumed to be produced from requirements by walking through the use cases in the CRC style, identifying the needed functionalities and their dependencies. The architecture produced by the genetic synthesis reflects functional requirements only to the extent the responsibilities have been identified. It may be perfectly sensible to produce an architecture proposal only for a subset of responsibilities that is expected to be architecturally significant.

3.2 Architectural patterns

In the context of the present paper, an architectural pattern can be any general structural solution applied at the architectural level to improve some quality attribute of the system. Architectural patterns have been systematically catalogued as architectural styles and design patterns, but here we regard basic practices like decomposition and use of interfaces also as architectural patterns. Each architectural pattern gives rise to

two mutation operations: introducing and removing the pattern.

In our experiments, we have used the following list of architectural patterns:

- decomposing a component
- using an interface
- Strategy design pattern [9]
- Façade design pattern [9]
- message dispatcher architectural style [21]
- communication through a dispatcher.

This collection of architectural patterns is of course very small, and intended only for experimentation purposes. We wanted to cover different levels of architectural patterns: basic practices, low-level design patterns (Strategy), medium-level design patterns (Façade), and high-level architectural styles (message dispatcher). The last architectural pattern is introduced for allowing components to join a message dispatcher introduced earlier. We expect that a real architecture synthesis tool would employ hundreds of architectural patterns.

3.3 Architectural laws

The purpose of architectural laws is to prevent various kinds of anomalies in the architecture. Mutation and crossover operations are implemented in such a way that these laws always hold. In our experiments, we have used three kinds of laws. Firstly, these laws ensure uniform calls between two classes: a class can communicate with another class only in a single manner (e.g. through an interface or through a message dispatcher). Secondly, the laws state some ground rules about architecture design, for example, that a responsibility can appear at most once in an interface, and that unused interfaces and data responsibilities implementing interfaces are not allowed. Thirdly, the laws regulate the order of introduction. For instance, a dispatcher must be introduced to the system before responsibilities can use it for communication.

3.4 Initial population

An initial population is first produced, where only basic structures, such as class division and interfaces for the responsibilities are randomly chosen. To ensure as wide a traverse through the search space as possible, four special cases are inserted: all responsibilities being in the same class, all responsibilities being in different classes, all responsibilities having their own interface, and all responsibilities being as much grouped to same interfaces as the class division allows.

3.5 Genetic encoding of architecture

In order for the genetic algorithm to operate on software architecture, the architecture needs to be represented as a chromosome consisting of genes. For efficiency, in this experiment the architecture encoding is designed to suit the chosen set of architectural patterns. We have followed the supergene idea, introduced by Amoui et al. [1]. In traditional chromosome representation, each chromosome consists of several genes, each of which has one field. A supergene, however, has several fields to store data in. Taking this idea as a starting point, it is quite straightforward to place all information regarding one responsibility into one supergene. This also makes it easier to keep the architecture consistent, as no responsibility can be left out of the architecture at any point, and there is no risk of breaking the dependencies between responsibilities.

There are two kinds of data regarding each responsibility r_i. Firstly, there is the basic information given as input. This contains the responsibilities $R_i = \{r_k, r_{k+1}, \ldots, r_m\}$ depending on r_i, its name n_i, type d_i, frequency f_i, parameter size p_i, execution time t_i, call cost c_i and variability v_i. Secondly, there is the information regarding the responsibility r_i's place in the architecture: the class(es) $C_i = \{C_{i1}, C_{i2}, \ldots, C_{iv}\}$ it belongs to, the interface I_i it implements, the dispatcher D_i it uses, the responsibilities $RD_i \subset R_i$ that call it through the dispatcher, and the design pattern P_i it is a part of. The dispatcher is given a separate field as opposed to other patterns for efficiency reasons. Figure 2 depicts the structure of a supergene.

R_i	n_i	d_i	f_i	p_i	t_i	c_i	v_i	C_i	I_i	D_i	RD_i	P_i

Figure 2. Supergene SG_i for responsibility r_i

The actual chromosome is formed by simply collecting all supergenes. Figure 3 illustrates a chromosome with *m* responsibilities.

SG_1	SG_2	….	SG_{m-1}	SG_m

Figure 3. Chromosome

Although basic operations in the architecture are relatively safe with this representation method (i.e., the responsibilities and their dependencies stay intact in the architecture), the design patterns produce challenges at the chromosome level, as careless operations can easily break patterns and make the architecture incoherent. Thus, in order to quickly check the legality of an operation with regard to patterns, a Pattern field is located in every supergene. The Pattern field has as attributes the classes and responsibilities "using" the

pattern, the classes and responsibilities "used by" the pattern, as well as the interfaces involved.

3.6 Mutation and crossover operations

All mutations are implemented as either introducing or removing an architectural pattern. This ensures a free traversal through the search space, as moves that may have seemed good at one time can be cancelled later on.

All mutations except for introducing a message dispatcher or a design pattern operate purely at supergene level by changing the value of one field. Introducing a new dispatcher to the system, however, affects the entire chromosome, and cannot be achieved by altering the data for a specific responsibility. Thus, the incorporation of a dispatcher is achieved by adding a "dummy" gene with only the dispatcher field containing data. Introducing patterns, on the other hand, operate at supergene level, but affect more than one gene. As mentioned in Section 3.5, it is a challenge to keep complex patterns intact through changes that should not affect patterns, e.g., merging classes. Because of this, the legality of a mutation is always checked before it is administered to the selected gene.

Mutations are given a certain probability with which they are applied. The roulette wheel method, where each mutation is given a "slice" in proportion to its probability, is used for selecting a mutation. A "null" mutation is also possible, giving a chromosome the chance to stay intact into the next generation. In addition, to study the effect of favoring more fundamental solutions in early stages, dynamic mutation probabilities have been defined for chosen patterns (dispatcher, Façade and Strategy). After 1/3 of the generations have been through, the probability of introducing a dispatcher to a system is decreased, and the probability of introducing a Façade pattern is increased respectively. After another 1/3 of generations have passed, the probability of the Façade mutation is decreased, and the probability for implementing a Strategy pattern is increased respectively. The hypothesis is that favoring fundamental solutions (like architectural styles) in the earlier stages of evolution leads to a stronger core architecture that can be more easily refined at later stages by lower-level solutions.

In our approach, the crossover operation is also seen as a type of mutation, and thus, it is also included in the "roulette wheel". The crossover is implemented as a traditional one-point crossover with corrective functions regarding design patterns. To properly correct the crossover result, an order of importance must be decided to deal with overlapping patterns from the two parent chromosomes. Following the idea by Burgess [4], it is decided that the left side of the offspring is always the valid one, and the right side of the crossover point is corrected so that the whole architecture is valid.

The crossover probability increases linearly in regard to how high the fitness of an individual is in the population. This increases the chances that the individual will stay intact after the mutation, as in order to fit the larger crossover "slice" to the "wheel", the probabilities of other mutations are decreased. Also, after crossover, the parents have a chance to be selected to the next population as such. This favors strong individuals to be kept intact through generations.

The actual mutation and crossover points (genes to be mutated) are selected randomly. However, we have taken advantage of the variability property of responsibilities with the strategy and dispatcher communication mutations. This should favor highly variable responsibilities. The chances of a gene being subjected to these mutations increase linearly according to the variability value of the corresponding responsibility.

3.7 Fitness function

The fitness function is based on widely used software metrics [18], most of which are from the metrics suite introduced by Chidamber and Kemerer [5]. These metrics have been used as a starting point for the fitness function, and have been further developed and grouped to achieve clear "sub-fitnesses" 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 architecture style in terms of modifiability, as well as the negative effect of the dispatcher in terms of efficiency. A complexity metric is added to penalize having many classes and interfaces as well as extremely large classes.

Dividing the fitness 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-fitness sf_i, the fitness function $f(x)$ for chromosome 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 multiplier 10 in sf_1

notes that having unused responsibilities in an interface is almost an architecture law, and should be more heavily penalized. The sub-fitness functions are defined as follows (|X| denotes the cardinality of X):

sf_1 = |interface implementors| + |calls to interfaces| + (|calls through dispatcher| ∗ ∑ (variabilities of responsibilities called through dispatcher)) – |unused responsibilities in interfaces| ∗ 10,

sf_2 = |calls between responsibilities in different classes|,

sf_3 = ∑ (|dependingResponsibilities within same class| ∗ parameterSize + ∑ |usedResponsibilities in same class| ∗ parameterSize + |dependingResponsibilities in same class| ∗ parameterSize)),

sf_4 = ∑ ClassInstabilities + |dispatcherCalls| ∗ ∑ callCosts, and

sf_5 = |classes| + |interfaces| + BigClassPenalty.

Selection of individuals for the next population is made with a roulette wheel selection, where the size of each "slice" is linearly in proportion to how high the corresponding individual's fitness is in the population. No individual can be selected more than once. Thus, the "slices" are adjusted after each selection to represent the differences between fitnesses of the remaining individuals.

4. Experiments

In this chapter we present the results from the preliminary experiments done with our approach, using the example system introduced in Section 3.1. All test runs were conducted with a fixed set of mutation probabilities, found after extensive testing. The calculated fitness value is the average of 10 best fitnesses in each generation. In all test runs apart from the first one (shown in Figure 4), the actual y-value for the curve is achieved as the average value from five test runs.

First, we tested that the development of fitness values was similar with each run of the algorithm. As can be seen in Figure 4, the fitness value curves are very similar in each of the five test runs, made with the same weights to all sub-fitnesses and having a population size of 100 and 250 generations.

After assuring that the populations did indeed develop similarly, the effect of the population size and amount of generations could be tested. Figure 5 shows the fitness curves achieved with the same weight for all sub-fitnesses and 250 generations with population size *p*. As can be seen, the curve achieves higher values when the size of the population grows. This is expected, as in bigger populations there are better possibilities to have more exceptionally good individuals when more options can be considered simultaneously.

In addition to the population size, the amount of generations is another basic parameter to be adjusted to all genetic algorithms. Different generation numbers were tested to see for how long the fitness values continued to increase. These tests were made with a population size of 100 and generation numbers of 250, 500 and 1000. As can be seen in Figure 6, depicting the evolvement of fitness values over 1000 generations, the fitness values achieve their highpoint after around 750 generations, and achieve quite high values already after 500 generations. The development with smaller generations was quite similar to the values that are achieved in this curve until 250 and 500 generations. However, the curves with smaller generations were slightly higher than here, as the dynamic mutation probabilities, discussed in Section 3.6, had a chance to enhance the development earlier in relation to 1000 generations.

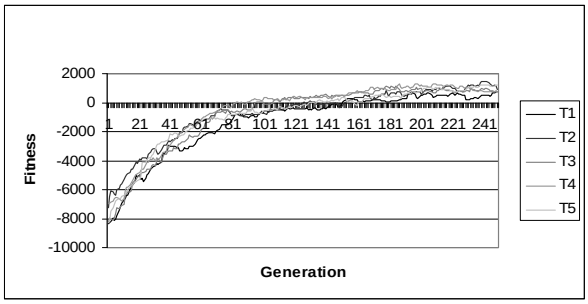

Figure 4. Fitness value development

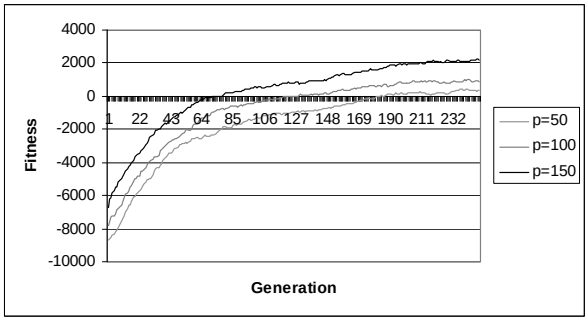

Figure 5. Different population sizes

To analyze the effect of weighing one quality evaluator over another, we have extracted the separate sub-fitness curves for modifiability and efficiency in cases where they were heavily weighted. In the first test, depicted in Figure 7 and made with a population size 100 and 250 generations, the modifiability functions were weighted 10 times higher than the efficiency functions. This results in the "normal" development of the modifiability curve, while the efficiency curve

plummets quite rapidly, and continues to worsen throughout the generations.

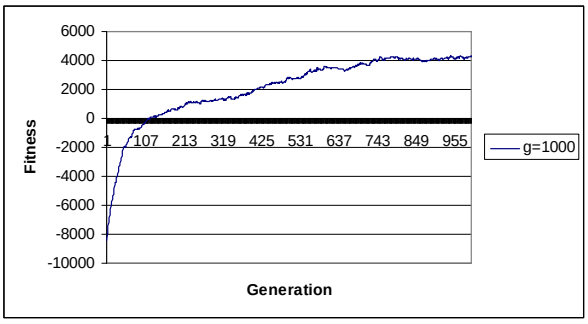

Figure 6. 1000 generations

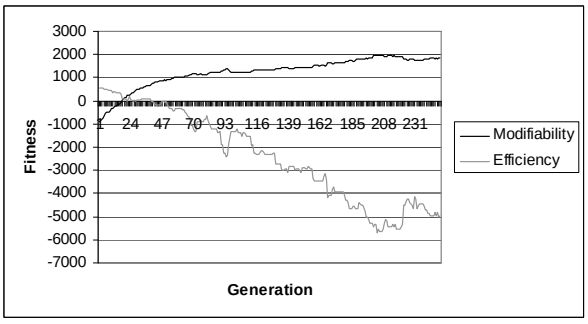

Figure 7. Heavily weighted modifiability

In the second test, also made with a population size of 100 and 250 generations, the efficiency fitnesses were correspondingly weighted 10 times higher than the modifiability functions. Figure 8 shows the respective efficiency and modifiability fitness curves. In this case, the efficiency curve achieves very high values from the very beginning and does not develop as noticeably as the modifiability fitness in the previous case. The modifiability fitness does not, however, reach high values or develop, as it stays close to its initial value around -500. The explanation for the poor development of the efficiency curve lies within the special cases inserted in the initial population. As the efficiency fitness values big classes, it would assign a high fitness value for the case where all responsibilities are in the same class. From this initial case, it is fairly easy to achieve individuals with very few classes, thus achieving high efficiency fitness values, which are hard to top, from the very beginning.

Finally, to analyze the effect of dynamic mutation probabilities for the different design patterns, it was tested whether there actually is a difference in the fitness curve if the mutation probabilities remained the same throughout the generations. In Figure 9, the curves for tests with dynamic and static pattern mutation probabilities are shown. As can be seen, with 250 generations and a population of 100, the fitness curve achieves its high point quite early when the mutations are static, but with the dynamic mutation probabilities, the fitness value continues to develop. At the middle part of the curves, tests with dynamic probabilities achieve lower values than tests with static probabilities. This may result from the population "suffering" from the design decisions that were made "too early", e.g., Strategy patterns, and it takes a while to achieve a level where the more refined design choices can be made so that they actually improve the system. Thus, it appears that dynamic mutation makes the basic structure of the architecture more amenable to fine-tuning in the later phases.

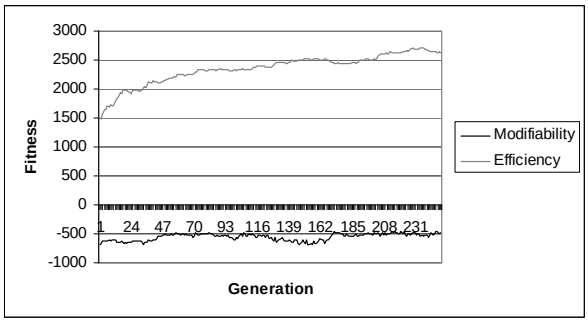

Figure 8. Heavily weighted efficiency

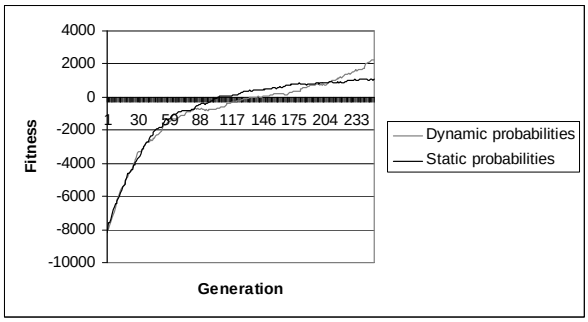

Figure 9. Pattern probability variation

In this chapter we have shown that the quality of an architecture increases quite steadily with the selected evaluators related to modifiability, efficiency and complexity. If some quality attribute is heavily weighted in the process, it may have significant negative effect on another. Using dynamic mutation probabilities seems to offer clear advantages in longer generation sequences.

5. Discussion

The best architecture of the last generation of each run is considered as the result of the architecture synthesis. This choice is somewhat arbitrary, since (i) a better architecture might very well have appeared in previous generations, and (ii) small differences in the fitness values are not significant. However, the "best of the breed" is an appropriate candidate for analyzing the result of the process in general.

Typically, the results contain 50-60 classes (or components) and interfaces. Recall that with 42 functional responsibilities, the theoretical maximum for classes and interfaces would be 84, since each responsibility can be both in a class and in an interface. This implies that the classes and interfaces are fairly small in the average.

In our experiments, none of the results would have been considered fully satisfactory as such by an experienced architect. The reason is that although many of the results contained quite sensible solutions, there were always also some solutions that no human architect would have done. The most common "unwanted" features were small classes (with only one responsibility) and classes with unrelated responsibilities. Obviously, the "right" application of the decomposition/composition pattern is difficult for the genetic process. This is not surprising, given that there is no direct reward for having logically related responsibilities in the same class. In contrast, design patterns and architectural styles (message dispatcher) usually appear in a sensible form.

As an example of the flavor of the results, Figure 10 depicts a part of the result of one run (made with 250 generations, lasting for 120 s), taken directly from the output of our experimental tool (made with Java). The example shows how the resulting architecture suggests using a message dispatcher (in the middle) between the user interface (introduced in Figure 1, present in the topmost class), user management, and the device handlers (below, e.g., Class87 contains drape control responsibilities introduced in Figure 1), with appropriate interfaces (each arrow represents a "call" of a certain responsibility). While this is a perfectly sensible solution, the architecture has in many cases separated logically related functionalities into one-responsibility classes or interfaces.

6. Concluding remarks

We have shown that regularly behaving software architecture synthesis is possible using genetic algorithms, with a reasonable number of generations. Especially the use of higher level structuring patterns, like architectural styles and design patterns, seems to fit the genetic process quite well.

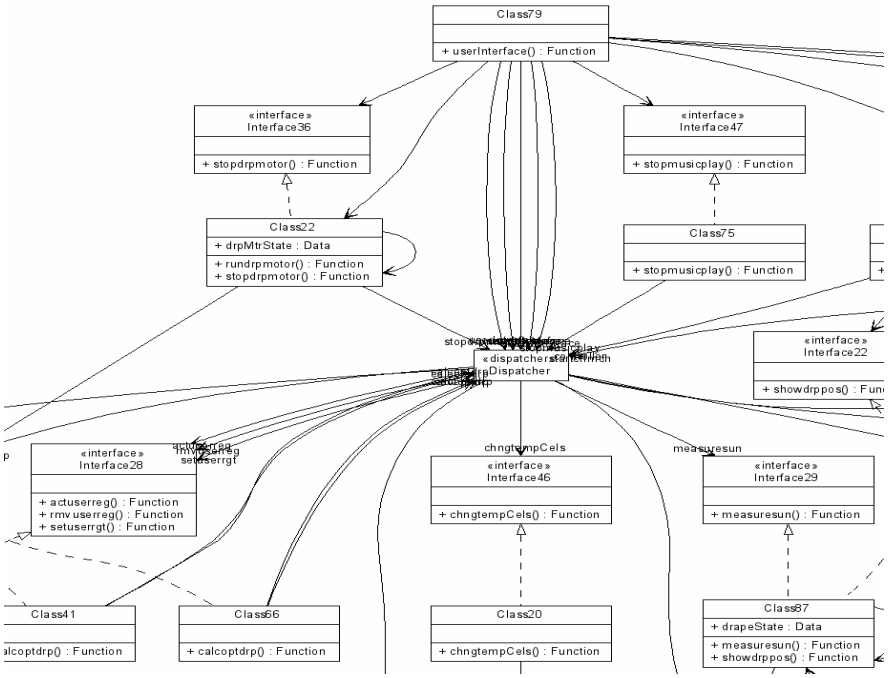

Figure 10. A part of an architecture proposed by genetic synthesis

Dynamic mutation probabilities of such patterns lead to improved development of the quality. Yet, fully satisfactory individual architecture proposals could not be produced.

We see this work as a first step on a fairly long road. We are still far from a situation where the result produced by genetic architecture synthesis could be adopted as such for a system, without human intervention. However, we feel strongly that reasonably high-quality architectures can be produced by genetic synthesis. This requires improved techniques for specifying and evaluating quality requirements, more comprehensive formulation of the architectural laws and patterns, and more fine-tuning of the genetic parameters, especially the fitness function. An attractive approach would be "guided" evolution, where the genetic process would be combined with deterministic design decisions in specific situations where a certain solution is known to work well. A dynamically changing fitness function to more correctly evaluate architectures at different stages of evolution is also a tempting direction to continue to. These topics are studied in our future work.

References

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

[2] K. Beck and W. Cunningham, A laboratory for teaching object-oriented thinking. In: *Proc. OOPSLA '89, Sigplan Notices* **24** (1989), 1-6.

[3] H. de Bruin and H. van Vliet, Quality-driven software architecture composition. *Journal of Systems and Software* **66**, 3 (June 2003), 269-284.

[4] C.J. Burgess, A genetic algorithm for the optimisation of a multiprocessor computer architecture, In: *Proc. GALESIA'9, IEE Conference Publication* **414**, 1995, 39-44.

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

[6] J. Clarke, J.J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper and M. Shepperd, Reformulating software engineering as a search problem, *IEE Proceedings - Software* **150**, 3 (2003), 161-175.

[7] M. Di Penta, M. Neteler, G. Antoniol and E. Merlo, A language-independent software renovation framework, *The Journal of Systems and Software* **77** (2005), 225-240.

[8] D. Doval, S. Mancoridis and B.S. Mitchell, Automatic clustering of software systems using a genetic algorithm, In: *Proc. of the Software Technology and Engineering Practice*, 1999, 73-82.

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

[10] L. Grunske, Identifying "good" architectural design alternatives with multi-objective optimization strategies. In: *Proc. of the 28th International Conference on Software Engineering,* Shanghai, China, 2006, 849 - 852.

[11] M. Harman, R. Hierons and M. Proctor, A new representation and crossover operator for search-based optimization of software modularization. In: *Proc. GECCO 2000*, 1351–1358.

[12] M. Harman and L. Tratt, Pareto optimal search based refactoring at the design level, In: *Proc. GECCO 2007,*1106-1113.

[13] V. Le Hanh, K. Akif, Y. Le Traon and J-M. Jézéquel, Selecting an efficient OO integration testing strategy: an experimental comparison of actual strategies. In: *Proc. ECOOP 2001*, LNCS **2072**, 2001, 381-401.

[14] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolutionary Programs.* Springer-Verlag, 1992.

[15] M. O'Keeffe and M. Ó Cinnéide, Towards automated design improvements through combinatorial optimization, In: *Workshop on Directions in Software Engineering Environments - 26th International Conference on Software Engineeering,* 2004, 75-82.

[16] M. O'Keeffe and M. Ó Cinnéide, Getting the most from search-based refactoring In: *Proc. GECCO 2007,* 1114-1120.

[17] P. Robertson, H. Shrobe and R. Laddaga (eds.), *Self-Adaptive Software*. Springer, 2000.

[18] H.A. Sahraoui, R. Godin and T. Miceli, Can metrics help bridging the gap between the improvement of OO design quality and its automation? In: *Proc. ICSM '00*, 2000, 154-162.

[19] O. Seng, M. Bauyer, M. Biehl and G. Pache, Search-based improvement of subsystem decomposition, In: *Proc. GECCO 2005,* 1045-1051.

[20] O. Seng, J. Stammel and D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, In: *Proc. GECCO 2006,* 1909-1916.

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