Search-based Improvement of Subsystem Decompositions

Olaf Seng, Markus Bauer, Matthias Biehl and Gert Pache
FZI Forschungszentrum Informatik
Haid-und-Neu-Strasse 10-14
Karlsruhe, Germany
{seng,bauer,biehl,pache}@fzi.de

ABSTRACT

The subsystem decomposition of a software system degrades gradually during its lifetime and therefore it gets harder and harder to maintain. As a result this decomposition needs to be reconditioned from time to time. The problem is to determine a suitable subsystem decomposition that can be used as a basis for future maintenance tasks. This paper describes a new methodology that computes such a subsystem decomposition by optimizing metrics and heuristics of good subsystem design. The main idea is to treat this task as a search problem and to solve it using a genetic algorithm.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design

Keywords

Software clustering, Remodularization, Genetic Algorithms, Software Metrics, Design Heuristics

1. INTRODUCTION

The structure of a software system inevitably degrades during its life cycle, because it has to be adapted to new and changing requirements [14]. Since these requirements are usually unforeseen and need to be implemented within a tight time frame, not enough care is spent on maintaining the system's structure. Since the structure has a major impact on the costs of system evolution, it has to be reconditioned from time to time.

One particular structural aspect is the subsystem decomposition of a software system. Subsystems group related functionality of the system into units of development. Therefore, the decomposition of a software system into subsystems is a key to handle the system's complexity. With the help of a proper subsystem decomposition, system understanding can be reduced to global knowledge on how the subsystems interact with each other. Additional implementation-level knowledge required to perform development or maintenance tasks can usually be confined to the inner structures of individual subsystems. Additionally, the decomposition of a software system into subsystems is a prerequisite to properly organize other aspects of the system's development, such as distributed development, reuse, encapsulating design decisions, testability and system deployment and operation.

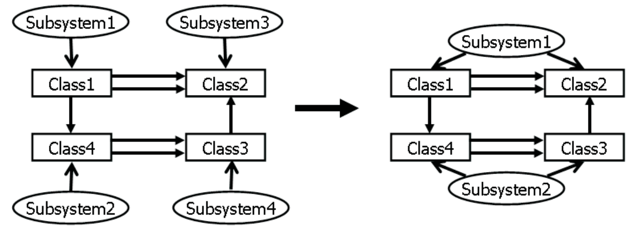

Figure 1: Example structure

Several guidelines and heuristics for good subsystem decompositions such as low coupling values between subsystems exist. In practice however, it is hard to reconcile all of them, since an improvement with respect to one guideline might lead to a degradation with respect to another.

Consider for example the structure shown in Figure 1. In the first diagram, each class forms its own subsystem. It should be obvious, that this is not a good subsystem decomposition. Placing *Class1* and *Class2* into a *Subsystem1* and *Class3* and *Class4* into a *Subsystem2* would certainly lead to more reasonable coupling and cohesion values, however it would also introduce a direct cyclic dependency between the two subsystems, which is generally regarded as an anti-pattern for subsystem decompositions.

The goal of our work is to develop a methodology for object-oriented systems that, starting from an existing subsystem decomposition, determines a decomposition with better metric values and fewer violations of design principles.

2. THE APPROACH

We express the task of improving a subsystem decomposition as a search problem. Software metrics and design heuristics are combined into a fitness function which is used to measure the quality of subsystem decompositions. Our

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'05, June 25–29, 2005, Washington, DC, USA.
Copyright 2005 ACM 1-59593-010-8/05/0006 ...$5.00.

search problem then consists in finding a subsystem composition that optimizes this fitness function.

The fitness function uses coupling, cohesion and complexity metrics, as well as further heuristics such as cyclic dependencies and bottlenecks. Cyclic dependencies between two or more subsystems are undesired because they affect the understandability and maintainability of the system. Bottlenecks are subsystems, which know about and are known by too many other entities [3]. If a desired architecture is given, e.g. a layered architecture, and there are several violations, our approach attempts to determine another decomposition that complies to the given architecture by moving classes around. Additional heuristics can be found in [13].

Well established search techniques like *Simulated Annealing* or *Hill Climbing* can be used to solve this search problem. However, our preferred optimization technique is a *Genetic Algorithm (GA)*, since using its crossover operator one can take advantage of regularities of the search space and therefore traverse it more efficiently [5].

2.1 Overview

Figure 2 shows the general steps of our approach using a genetic algorithm.

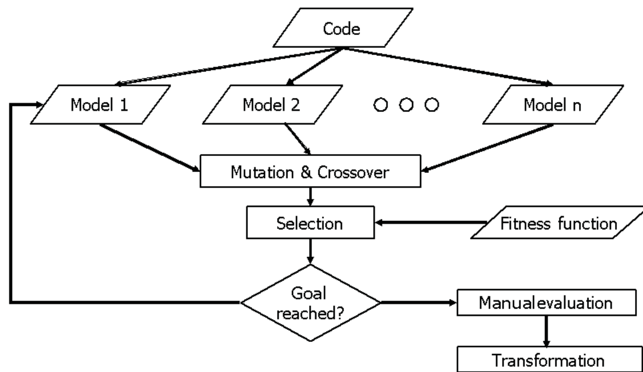

Figure 2: Algorithm

Instead of working directly on the source code, we first transform it into an abstract representation, which is suitable for common object orientied languages. In our GA, several potential solutions, i.e. subsystem decompositions, form a population. The initial population can be created using different initialization strategies, which we are going to describe in Section 2.2. With the help of the genetic operators *mutation* and *crossover*, the n solutions of a population are randomly modified or new solutions are created by recombining existing solutions. Using the fitness function and a *tournament selection strategy* the most promising n elements are selected and form the next generation's population. After a certain, predefined number of generations, the optimization goal is reached and the algorithm picks the best subsystem decomposition of the final population.

Before the algorithm starts, the user can customize the fitness function by selecting several metrics or heuristics as well as by changing thresholds (see Section 2.3). Upon termination of the algorithm, the new structure is presented to the user. By comparing the initial structure with the final structure proposed by our algorithm, the user can easily infer a number of refactorings to improve the decomposition of the software system, such as *moving* a class from one subsystem to another one, *splitting* up an existing subsystem or *joining* two existing subsystems. These refactorings can then be carried out manually or with tool support [7, 16].

Our model is a directed graph. The nodes of the graph can either represent subsystems or classes. Edges between subsystems or subsystems and classes denote containment relations, whereas edges between classes represent dependencies between classes. A class *depends* on some other class, if

- it inherits from the other class
- its implementation calls one ore more methods of the other class
- its implementation accesses attributes of the other class
- it references the type of the other class, e.g. in some variable declaration

We can have multiple dependency edges between classes. These dependencies will be evaluated by the fitness functions (and also by the operators of our GA).

2.2 Hybrid GA design: Forming and preserving building blocks

In this section we propose a hybrid GA for software decomposition that is supported by domain knowledge and problem specific algorithms. According to the building block hypothesis, GAs construct high quality solutions by assembling building blocks [8]. Building blocks are short, low order, highly fit schemata. The generic GA [9] finds, preserves and composes the building blocks over time by applying operators and selection. In our approach we design the components of the GA (operators, representation, initial population) by adding domain knowledge to support the forming and preserving of building blocks.

Our approach is based on the *Grouping Genetic Algorithm* (GGA) which is particularly well suited for finding groups in data [5]. The decomposition of software systems is a special kind of grouping problem, where the set of all classes is decomposed in a complete and consistent set of subsets or subsystems. Therefore the GGA can be applied for software decomposition.

2.2.1 Representation

When building a representation for subsystem decompositions the problem is to map the two-layered data structure of the set of subsystem candidates, which in turn consist of classes, to a sequence of genes. The representation of the GGA allows us to do this by associating subsystem candidates with genes and using the power set of classes as the alphabet for genes. Consequently a gene is associated with a set of classes, i.e. an element of the power set. Furthermore, this representation allows a one-to-one mapping of geno- and phenotype to avoid redundant coding.

2.2.2 Operators

We use an adapted GGA crossover operator and three kinds of mutation. The operators proposed by the original GGA in [5] are kept very general and consequently blind for domain knowledge. We adapt them to use domain knowledge that is encoded in the dependency graph, by embedding problem specific heuristics into the operators. Thus we are able to guide the creation of new offsprings in such a

way, that the newly created individuals are likely to have an improved fitness.

Furthermore the operators are designed to be non-destructive and to preserve a complete subsystem candidate as far as possible. The operators take care to produce only consistent and complete decompositions, so we do not waste computation time on infeasible solutions.

The *crossover* operator forms two children from two parents. After choosing the parents, the operator selects a sequence of subsystem candidates in both parents (step 0) and mutually integrates them as new subsystem candidates in the other parent (step 1) and vice versa, thus forming two new children consisting of both old and new subsystem candidates. Old subsystem candidates which now contain duplicated classes are deleted (step 2), their non-duplicated classes are collected (step 3) and distributed over the remaining subsystem candidates (step 4). In this step we consider the number of dependencies between the classes that are to be distributed to new subsystem candidates. We allocate them to those subsystem candidates which have the strongest connections to the classes. The process of the crossover operator is depicted in Figure 3, where we show how one of the two possible children is created.

The *split&join mutation* either divides a subsystem candidate into two smaller subsystem candidates or joins two subsystem candidates by unifying their classes. The operator splits a subsystem candidate in such a way, that the separation in two subsystem candidates occurs at a loosely associated point in the dependency graph. Similarly, the operator connects two subsystem candidates with strong association weight.

Elimination mutation deletes a subsystem candidate and distributes its classes to other subsystem candidates, based on association weights. Elimination mutation is part of our crossover operator discussed above.

Adoption mutation tries to find a new subsystem candidate for an orphan, i.e. a subsystem candidate containing only a single class. Thus our approach naturally implements an orphan adoption technique [18]. Orphan adoption avoids useless subsystems candidates containing only a single class. Our operator simply moves the orphan to the subsystem candidate that has the highest connectivity to the orphan.

2.2.3 Initial population

The building block theory tells us, that the GA constructs solutions by combining building blocks. But where do these building blocks come from? As a general purpose search the GA is claimed to find building blocks over time [8]. But since we design a specialized GA for software decomposition, we can use domain knowledge to shortcut the search for building blocks and speed up the convergence. Thus the suboptimal results of problem specific algorithms can be used to create an initial population that might help the GA to find proper building blocks fast [10].

For good starting populations, two competing properties are desirable. On the one hand the individuals should have a high fitness, so good building blocks are already present in the population. On the other hand, the GA needs diversity in the population to be able to explore the search space.

We propose to balance the competing goals by taking randomly selected connected components of the dependency graph for half of our population and highly fit ones for the rest.

The strategy for finding highly fit individuals may vary depending on the availability of existing decompositions:

- If a suitable decomposition is given (e.g. by the package structure of a Java system), we use it as the highly fit initial population.
- If no decomposition is available, we attempt to build several suboptimal decompositions. Our approach is based on a modification of Kruskal's algorithm for the construction of minimum spanning trees (MST) on the dependency graph [20]. We modify this greedy algorithm by defining a threshold for the unification of two subtrees of the MST. This results in a solution that consists of a forest representing initial building blocks of the decomposition. Using different thresholds, which are chosen randomly from a certain interval, we can create a set of individuals representing the highly fit half of our initial population.

2.3 Fitness function

Our fitness function $fit(s)$ is a multi modal fitness function. Each of the individual functions calculates a value between 0 and 1, where 1 is the optimal value. Such a multi modal fitness function can be easily mapped into a linear fitness function, by just adding up the weighted individual values.

$$fit(s) \begin{cases} cohesion(s) & : \dfrac{\sum_{i=1}^{\#s} \sum_{j=1}^{\#c(s_i)} \frac{\#k(c_j)}{\#c(s_i)^2}}{\#s} \\ coupling(s) & : 1 - \sum_{i=1}^{\#s} \frac{\#rO(s_i)}{\#r} \\ complexity(s) & : \sum_{i=1}^{\#s} \left(com(s_i) * \frac{\#c(s_i)}{\#c} \right) \\ cycles(s) & : 1 - \dfrac{\sum_{i=1}^{n} size(scc[i])^k}{\#s^k} \\ bottlenecks(s) & : 1 - \sum_{i=1}^{\#s} \frac{min(inDeg(s_i), outDeg(s_i))}{\#s * maxDeg} \end{cases}$$

Currently we are using standard coupling and cohesion metrics as parts of our fitness function [2]. To measure the *cohesion* for a system s, we sum up the cohesion values for the individual subsystems in s. The cohesion for a subsystem s_i is determined by counting the number of different classes inside s_i known by some class $c_j \in s_i$ ($\#k(c_j)$) and divide this by the square of the number of classes in s_i ($\#c(s_i)$). This value is then normalized by dividing it by the number of subsystems (#s).

The *coupling* function is the sum of the coupling values for each subsystem in s. The coupling value for one subsystem s_i is calculated in the following way: at first, we count the number of dependency edges between classes inside s_i and classes belonging to other subsystems s_j ($\#rO(s_i)$). This number is divided by the overall number of dependency edges (#r) in s.

The *complexity* function adds up the complexity values $com(s_i)$ of all subsystems s_i in s, normalized by the proportion of classes $\frac{\#c(s_i)}{\#c}$ of s_i in s. The complexity value $com(s_i)$ of a subsystem s_i depends on four threshold parameters: $com(s_i)$ is considered to be optimal (i.e = 1) if the complexity of a subsystem is inside the interval $[minO, maxO]$. Otherwise, the value is linearly interpolated between 0 and 1 inside the intervals $[minU, minO]$ and $[maxO, maxU]$. This

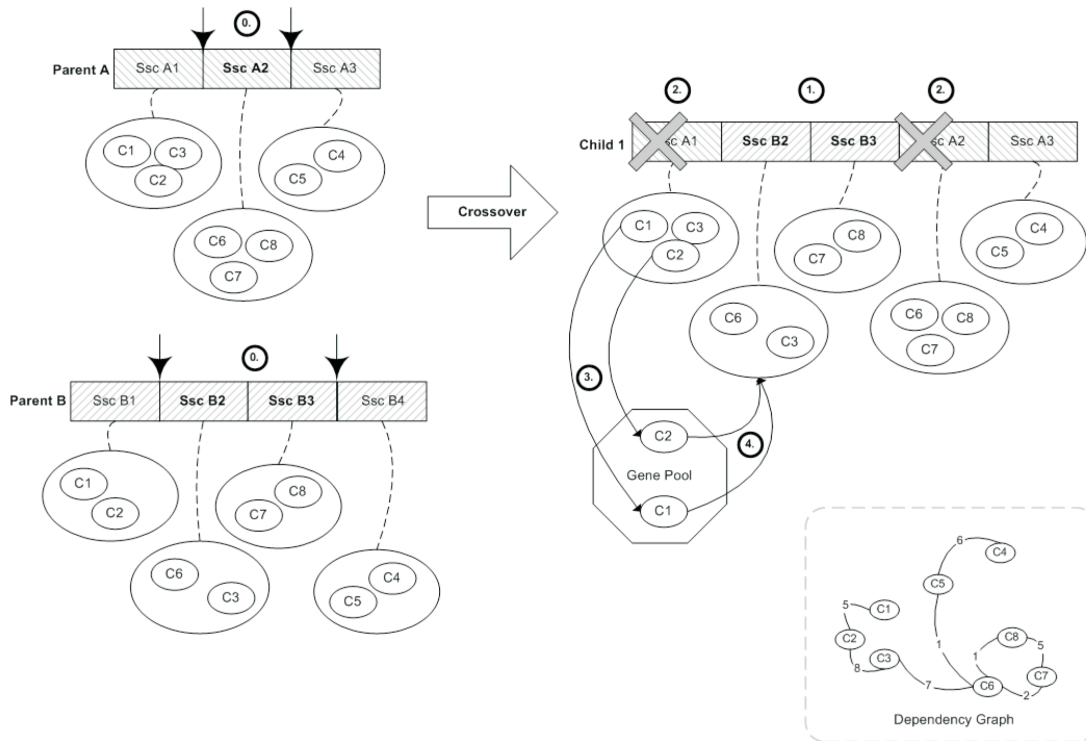

Figure 3: GGA crossover for one child, where *Ssc* stands for *subsystem candidate* and *C* for *class*

is depicted in Figure 4. The complexity of a subsystem can be measured in different ways. We have successfully used the number of classes or McCabe's control flow complexity. The advantage of such a fitness function is its fuzzy shape, which does not punish complexity values too hard if they are only slightly outside the predefined optimal values.

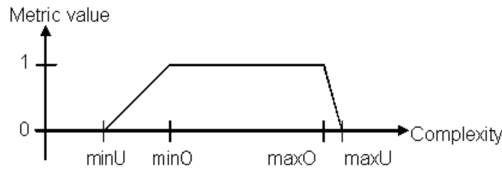

Figure 4: Shape of complexity fitness functions

For calculating the cycle and bottleneck fitness functions, we use a customized graph model, that can be easily constructed using the model described in Section 2.1 . Nodes denote subsystems and the only kind of edges we have are dependency edges between subsystems. A subsystem depends on another subsystem, if one of its classes depends on a class belonging to the other subsystem. The fitness value for *cycles* is calculated by summing up the size of strongly connected components of this modified graph. To give higher penalties to cycles with more than two subsystems involved, we usually use a factor $k \geq 1$. To normalize this fitness value, the sum is divided by the overall number of subsystems.

For calculating the *bottleneck* metric for a subsystem we measure the in-degree and out-degree of each subsystem and divide the minimum of the two by the highest in- or out-degree (*maxDegree*) currently found in our customized graph.

3. EVALUATION

3.1 Implementation

To evaluate our approach, we have implemented a tool prototype, *Evo*. The focus of this prototype is to experiment with different variations of our GA, e.g. testing different combinations and parameters for the operators, strategies for initial populations, representations and fitness functions. Using the *Recoder*[1] library, *Evo* constructs a model (see Section 2) of the system to be analyzed. Our GA is implemented on top of this model. To experiment with our GA, the implementation of the GA is structured into a number of exchangeable components. That way a large number of variations to our GA can easily be evaluated.

Applying *Evo* to a number of case-studies has shown, that even though our operators are a little more complex than those of the classic GA, the execution of our algorithm is quite fast: the execution usually takes a few minutes on an average PC for 100 solutions per population and 100 generations. We gain speed by using the efficient tournament selection instead of roulette wheel selection. Our evaluation on the clustering of different software systems has revealed, that results of roulette wheel selection are only slightly better than those of tournament selection. The adapted operators allow us to use a relatively small population size and few generations.

[1] http://recoder.sf.net

In the following, we present some results from applying our approach to the Java case-study *JHotDraw*. We show that our approach works well and provide some insights on the impact of our fitness function – especially the cycle and bottleneck heuristics. We conclude our evaluation by evaluating and comparing a few variations of our GA.

3.2 Case-study: JHotDraw

Our case-study *JHotDraw*[2] (Version 5.3) is a Java GUI framework for technical and structured Graphics. *JHotDraw* in this version consists of nine packages and 207 classes. Since it is a Java case-study, we used the existing package structure as an initial subsystem decomposition. *JHotDraw's framework* subsystem was intentionally left out, since there are natural static dependencies between a framework and framework users, that cannot be treated properly without classifying the entities beforehand. We concentrated on the four subsystems: *contrib, figures, standard* and *util* that make up most of the application's core (165 classes).

The column of the following table labelled "Initial" shows the fitness values for the initial subsystem decomposition. The higher the values the better.

	Initial	CC	CCC	CCCBC
Coupling	0.573	0.86	0.63	0.57
Cohesion	0.51	0.54	0.55	0.55
Complexity	0.04	0.23	0.98	0.95
Bottlenecks	0.583	0.91	0.76	0.93
Cycles	0	0.88	0.30	0.93

We can observe that the values for cohesion, complexity and cycles are not too good. This is due to the fact that the subsystems contain too many classes, classes inside the subsystems are not really related and all subsystems form one strongly connected component. The next three columns show the average results of seven optimizations carried out three times for different optimization goals. At first, we only optimized coupling and cohesion (CC), at second, we additionally used our new complexity fitness function (CCC) and at third, we made use of the cycle and bottleneck heuristics (CCCBC).[3] Using the first optimization goal, we were able to achieve really good coupling values. Cohesion, bottleneck and cycle values are ok, but complexity values are pretty bad. This is due to the fact, that almost all classes are put into one giant subsystem. This also explains the good values for the cycle and bottleneck heuristics. Using complexity as an additional optimization goal, we can achieve properly-sized subsystems, but the values for bottlenecks and cycles are bad.

Only if we optimize for all of our criteria, we are able to achieve a suitable compromise with very good complexity, bottleneck and cycle values and good values for coupling and cohesion. The best solution resulted in a subsystem decomposition consisting of 25 subsystems with an average subsystem size of seven classes. The strongly connected component originally consisting of all subsystems has now been reduced to a strongly connected component consisting of only five out of 25 subsystems. We achieved 15 subsystems with an in- or out-degree of zero. The highest in- out-degree is six. Overall, our approach managed to improve the existing decomposition.

Manual inspection revealed, that we successfully decomposed the large subsystems, e.g. all classes of the *util*-package concerned with version management (*VersionControlStrategy, VersionManagement, StandardVersionControlStrategy* and *VersionRequester*) were put in a separate subsystem. Furthermore, the classes of the *contrib*-package responsible for modelling polygons and triangles (*PolygonFigure, PolygonScaleHandle, ChopPolygonConnector, PolygonHandle, TriangleRotationHandle* and *TriangleFigure*) have also been grouped into a separate subsystem.

3.3 Evaluation of the design decisions in our GA

In Section 2.2 we presented different possibilities for the operators of our GA and several strategies for creating the initial population. We compare these possibilities by analyzing their effects on the development of the population by applying each possibility exclusively on the same case-study. For this evaluation we chose *javax.swing* of the Sun JDK 1.4, containing more than 1500 classes. We examine our GA with a population of 100 individuals. For each variation we record the development of its fitness for 100 generations and calculate both average and maximum fitness of the population. To minimize the impact of random effects, we have repeated our measurements 8 times and present both the medium average fitness and the medium maximum fitness of the individuals in the respective population.

Figure 5 shows the effects of crossover, split&join mutation and adoption mutation applied separately as well as in combination. For this case-study, the split&join mutation creates individuals of the highest fitness, closely followed by the combination of all operators. In general, however, the most reliable way to produce highly fit individuals is to employ a combination of all operators.

In Figure 6 we present one of our experiences with different strategies to create initial populations. We experimented with purely random decompositions and highly fit decompositions created by the randomized MST algorithm. We could not determine that one of the two strategies for initial populations is consistently superior to the other one, but each time the quality gap between them is quite large. Thus we combine the strategies to get the best out of both.

4. RELATED WORK

The related work to our paper can be categorized into two major areas of research: genetic algorithms and computing subsystem decompositions of software systems (software clustering).

Genetic algorithms [8] have become a valid tool for tackling computationally hard problems. Finding groups in data is such a problem. The basic, classical GA [9] has been successfully applied to this problem area. However, Falkenauer [5] pointed out the limited suitability of the basic GA for grouping problems and proposes a heavily adapted GA, *GGA*, to solve the problem of clustering data into groups. In Section 2.2 we already showed why we chose Falkenauer's GGA to form the foundation for the algorithm used in our approach and how we tailored it using specific knowledge from the domain of software decomposition.

[2]http://www.jhotdraw.org/

[3]For *JHotDraw* we chose the thresholds for the complexity function to consider subsystem sizes between five and twelve classes as optimal.

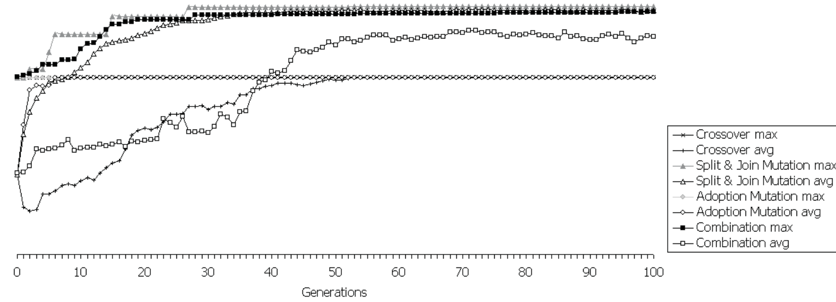

Figure 5: Contribution of the operators in the case-study *javax.swing*

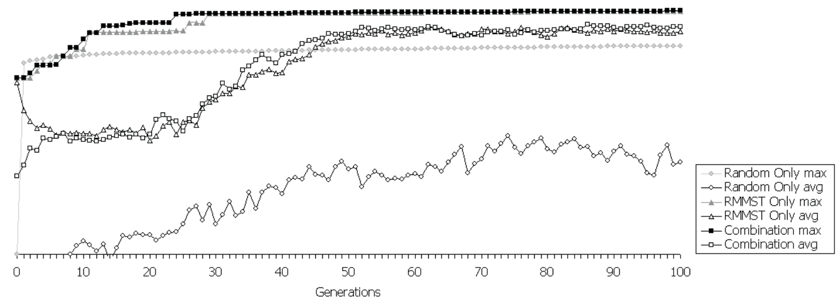

Figure 6: Contribution of the initial populations in the case-study *javax.swing*

For the remaining part of this section, we will review some of the previous work in the research area of *clustering software systems*:

In the last decade, there have been numerous publications on using clustering algorithms for the remodularization of software systems: Wiggerts [19], Anquetil and Lethbridge [1], Tzerpos and Holt [17] are among the first to systematically explore the application of clustering algorithms for system remodularization and architecture recovery. In general, these approaches employ an abstract model of the structure of a software system, which consists of its basic entities (e.g. functions, classes or files) and relationships between them (such as calls or include dependencies or type dependencies). A clustering algorithm is then used to group related entities of the software system into module or subsystem candidates. The clustering algorithm is guided by similarity measurements derived from the relationships between the system's entities.

An important group of clustering algorithms are hierarchical, agglomerative algorithms. These algorithms start with the assumption that each element forms a cluster and iteratively join the most similar clusters into bigger ones, thus producing a tree of clusters. In practice these algorithms may compute good results (see [1] and [15]), however, for larger systems a lot of manual work is involved to pick suitable subsystem or module candidates from the resulting cluster tree.

Another important group of algorithms are graph based algorithms. For example, Bauer and Trifu [2] use an algorithm (MMST) that clusters object-oriented systems into modules by picking clusters from subgraphs that can be derived from the minimal spanning tree of the software system's dependency graph.

Mancoridis, Mitchell and others [12] treat clustering as an optimization problem: Find the system decomposition that optimizes a system modularization quality function MQ which expresses that the clusters of the decomposition should have high internal cohesion (intra-connectivity) and low external coupling (inter-connectivity). To solve the optimization problem, a GA is used [4]. A tool, *Bunch*, implements these ideas [11]. An extensive set of case-studies is presented to prove the suitability of the approach for software architecture recovery. Although our approach is similar to theirs, there are a number of differences: (1) Besides coupling and cohesion, our fitness function evaluates additional quality properties of the system's decomposition such as individual subsystem sizes and the absence of bottlenecks and cycles. (2) Our GA uses a different encoding to map the clusters into a sequence of genes. Our algorithm is derived from the GGA and associates a gene with a set of nodes of the dependency graph, thus allowing for the embedding of problem specific heuristics into the crossover and mutation operators (see Section 2.2). *Bunch*'s encoding (the genetic string maps each node of the dependency graph to a cluster's id), its crossover and mutation operators are derived from the classic GA. (3) Our GA uses a problem specific initialization strategy (by generating a number of individuals of the initial population using a randomized MMST clustering algorithm or deriving them from an existing subsystem structure), whereas *Bunch* randomly creates the initial population.

5. SUMMARY AND FUTURE WORK

In our paper, we have presented an approach to improve the subsystem decomposition of a software system. Core of the approach is a genetic algorithm that automatically computes a good decomposition of a software system into subsystems. Our approach significantly differs from the work of others, because (1) it uses a fitness function that adds additional heuristics of good software design, and (2) it employs a hybrid GA whose representation and operators embed domain knowledge. We have shown that our approach works well, therefore we believe that it forms a convincing foundation for further research in this area.

Specifically, we propose to further improve and extend our approach in the following ways:

- Exploit more heuristics on good subsystem decomposition like Robert Martin's stability metric [13] and embed them into the fitness function.
- Use a weighted graph to model the system's dependencies (see section 2.1). Using weighted dependencies will allow for the distinction of the different dependency types found in object-oriented systems. For example, an inheritance relationship between classes should result in a stronger coupling than a simple attribute access. Work by Bauer and Trifu [2] and Rayside et al. [15] indicates that this will further improve the clustering results. Note, that this will require a modified fitness function that calculates coupling and cohesion metrics from weighted edges of the dependency graph.
- Improve the clustering results by classifying the system's entities and their relationships according to the role the play in a system's architecture and introduce weights into the dependency graph accordingly. For example, a call to a *facade* [6] should be result in lower coupling metrics than calls between classes that implement the functionality hidden by the facade. As Bauer and Trifu have shown in [2], this will improve the subsystem decomposition for many situations. In many cases, for example, framework code can be successfully separated from application code using this strategy.

We would also like to experiment with genetic algorithms to improve the structure of the system on a finer level of abstraction. A genetic algorithm could move methods between classes or improve the inheritance tree in an object oriented software systems. First results in this area are promising [16].

6. REFERENCES

[1] N. Anquetil and T. C. Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 235–255, 1999.

[2] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of oo systems. In *Eighth European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 3 – 14, 2004.

[3] O. Ciupke. Automatic Detection of Design Problems in Object-Oriented Reengineering. In *Technology of Object-Oriented Languages and Systems - TOOLS 30*, 1999.

[4] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of the IEEE Conference on Software Technology and Engineering Practice*, pages 73–81, 1999.

[5] E. Falkenauer. *Genetic algorithms and grouping problems*. Wiley, New York, 1998.

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

[7] T. Genssler and V. Kuttruff. Source-to-source transformation in the large. In *Proceedings of the Joint Modular Language Conference*. Springer, August 2003.

[8] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, 1989.

[9] J. H. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

[10] J. Hromkovic. *Algorithmics for hard problems*. Springer, 2. ed. edition, 2003.

[11] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 50–59, 1999.

[12] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the Sixth International Workshop on Program Comprehension*, pages 45–52, 1998.

[13] R. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[14] D. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, 1994.

[15] D. Rayside, S. Reuss, E. Hedges, and K. Kontogiannis. The effect of call graph construction algorithms for object-oriented programs on automatic clustering. In *Proceedings of the Eighth International Workshop on Program Comprehension*, pages 191–200. IEEE, 2000.

[16] O. Seng and G. Pache. Search based structure improvement. In *Proceedings of the first International Workshop on Software Evolution Transformations (SET 2004)*, pages 7–10. Queens University, Kingston, Ontario, Canada, Nov. 2004.

[17] V. Tzerpos and R. Holt. Software botryology. automatic clustering of software systems. In *Proceedings of the Ninth International Workshop on Database and Expert Systems Applications*, pages 811–818, 1998.

[18] V. Tzerpos and R. C. Holt. The orphan adoption problem in architecture maintenance. In *Working Conference on Reverse Engineering (WCRE 1997)*, page 76, Amsterdam, The Netherlands, Oktober 1997.

[19] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43, 1997.

[20] C. T. Zahn. Graph theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on Computers*, 20(1), 1971.