Outi Räihä

A Survey on Search-Based Software Design

DEPARTMENT OF COMPUTER SCIENCES
UNIVERSITY OF TAMPERE

D-2009-1

TAMPERE 2009

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCES
SERIES OF PUBLICATIONS D – NET PUBLICATIONS
D-2009-1, MARCH 2009

Outi Räihä

A Survey on Search-Based Software Design

DEPARTMENT OF COMPUTER SCIENCES
FIN-33014 UNIVERSITY OF TAMPERE

ISBN 978-951-44-7668-6
ISSN 1795-4274

A Survey on Search-Based Software Design

OUTI RÄIHÄ
University of Tampere, Finland

Search-based approaches to software design are investigated. Software design is considered from a wide view, including topics that can also be categorized under software maintenance or re-engineering. Search-based approaches have been used in research from high architecture level design to software clustering and finally software refactoring. Enhancing and predicting software quality with search-based methods is also taken into account as a part of the design process. The choices regarding fundamental decisions, such as representation and fitness function, when using in meta-heuristic search algorithms, are emphasized and discussed in detail. Ideas for future research directions are also given.

Categories and Subject Descriptors: D.2.10. [**Software Engineering**]: *Design*; D.2.11. [**Software Engineering**]: *Software Architectures*; G.1.6 [**Numerical Analysis**]: *Optimization.*

General Terms: Algorithms, Design

Additional Key Words and Phrases: search-based software engineering, clustering, refactoring, software architecture design, software quality, genetic algorithm, hill climbing, simulated annealing,

1. INTRODUCTION

Interest in search-based approaches in software engineering has been growing rapidly over the past years. Extensive work has been done especially in the field of software testing, and a covering survey of this branch has been made by McMinn [2004]. Other problems in the field of software engineering have been formulated as search problems by Clarke et al. [2003] and Harman and Jones [2001]. Harman [2007] has also provided a brief overview to the current state of search-based software engineering. This survey will cover the branch of software design, where refactoring and modularization have also been taken into account as they are considered as actions of "re-designing" software. New contribution is made especially in summarizing research in architecture level design that uses search-based techniques, as it has been quite overlooked in previous studies of search-based software engineering. Harman [2004] points out how crucial the representation and fitness function are in all search-based approaches to software engineering. When using genetic algorithms [Holland, 1975], which are especially popular in search-based design, the choices regarding genetic operators are just as important and very difficult to define.

This survey emphasizes the choices made regarding the particular characteristics of search algorithms; any new study in the field of search-based software engineering would benefit from learning what kind of solutions have proven to be particularly successful in the past.

This survey proceeds as follows. The underlying concepts for genetic algorithms and simulated annealing are presented in Section 2. Search-based architecture design, clustering and refactoring are presented in Sections 3, 4 and 5, respectively. The background for each underlying problem is first presented, followed by recent approaches applying search-based techniques to the problem. In addition, research regarding meta-heuristic search algorithms and quality predictive models is discussed in Section 6. Finally, some ideas for future work are given in Section 7, and conclusions are presented in Section 8.

2. SEARCH ALGORITHMS

To understand the basic concepts behind the approaches presented here, I will briefly introduce genetic algorithms (GAs) and simulated annealing (SA). In addition to these algorithms hill climbing (HC) in various forms is used in the studies discussed here. However, the basic application of hill climbing techniques is assumed to be known. For a detailed description on GA, see Mitchell [1996] or Michalewicz [1992], for SA, see, e.g., Reeves [1995], and for HC, see Clarke et al. [2003]. For a description on multi-objective optimization with evolutionary algorithms, see Deb [1999] or Fonseca and Fleming [1995].

2.1 Genetic algorithms

Genetic algorithms were invented by John Holland in the 1960s. Holland's original goal was not to design application specific algorithms, but rather to formally study the ways of evolution and adaptation in nature and develop ways to import them into computer science. Holland [1975] presents the genetic algorithm as an abstraction of biological evolution and gives the theoretical framework for adaptation under the genetic algorithm [Mitchell, 1994].

In order to explain genetic algorithms, some biological terminology needs to be clarified. All living organisms consist of cells, and every cell contains a set of chromosomes, which are strings of DNA and give the basic information of the particular organism. A chromosome can be further divided into genes, which in turn are functional blocks of DNA, each gene representing some particular property of the organism. The different possibilities for each property, e.g. different colors of the eye, are called alleles. Each gene is located at a particular locus of the chromosome. When reproducing, crossover occurs: genes are exchanged between the pair of parent chromosomes. The offspring is subject to mutation, where single bits of DNA are changed. The fitness of an organism is the probability that the organism will live to reproduce and carry on to the next generation [Mitchell, 1996]. The set of chromosomes at hand at a given time is called a population.

Genetic algorithms are a way of using the ideas of evolution in computer science. When thinking of the evolution and development of species in nature, in order for the species to survive, it needs to develop to meet the demands of its surroundings. Such evolution is achieved with mutations and crossovers between different chromosomes, i.e., individuals, while the fittest survive and are able to participate in creating the next generation.

In computer science, genetic algorithms are used to find a good solution from a very large search space, the goal obviously being that the found solution is as good as possible. 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.

2.2 Simulated annealing

Simulated annealing is originally a concept in physics. It is used when the cooling of metal needs to be stopped at given points where the metal needs to be warmed a bit before it can resume the cooling process. The same idea can be used to construct a search algorithm. At a certain point of the search, when the fitness of the solution in question is approaching a set value, the algorithm will briefly stop the optimizing process and revert to choosing a solution that is not the best in the current solution's neighborhood. This way getting stuck to a local optimum can effectively be avoided. Since the fitness function in simulated annealing algorithms should always be minimized, it is usually referred to as a cost function [Reeves, 1995].

Simulated annealing optimally begins with a point x in the search space that has been achieved through some heuristic method. If no heuristic can be used, the starting point will be chosen randomly. The cost value c, given by cost function E, of point x is then calculated. Next a neighboring value x_1 is searched and its cost value c_1 calculated. If $c_1 < c$, then the search moves onto x_1. However, even though $c \leq c_1$, there is still a small chance, given by probability p that the search is allowed to continue to a solution with a bigger cost [Clarke et al., 2003]. The probability p is a function of the change in cost function ΔE, and a parameter T:

$$p = e^{-\Delta E/T} .$$

This definition for the probability of acceptance is based on the law of thermodynamics that controls the simulated annealing process in physics. The original function is

$$p = e^{-\Delta E/kt},$$

where t is the temperature in the point of calculation and k is Boltzmann's constant [Reeves, 1995].

The parameter T that substitutes the value of temperature and the physical constant is controlled by a cooling function C, and it is very high in the beginning of simulated annealing and is slowly reduced while the search progresses [Clarke et al., 2003]. The actual cooling function is application specific.

If the probability p given by this function is above a set limit, then the solution is accepted even though the cost increases. The search continues by choosing neighbors and applying the probability function (which is always 1 if the cost decreases) until a cost value is achieved that is satisfactory low.

3. SOFTWARE ARCHITECTURE DESIGN

The core of every software system is its architecture. Designing software architecture is a demanding task requiring much expertise and knowledge of different design alternatives, as well as the ability to grasp high-level requirements and piece them to detailed architectural decisions. In short, designing software architecture takes verbally formed functional and quality requirements and turns them into some kind of formal model, which is used as a base for code. Automating the design of software is obviously a complex task, as the automation tool would need to understand intricate semantics, have access to a wide variety of design alternatives, and be able to balance multi-objective quality factors. From the re-design perspective, program comprehension is one of the most expensive activities in software maintenance. The following sections describe meta-heuristic approaches to software architecture design for object-oriented and service-oriented architectures.

3.1 Object-oriented architecture design

3.1.1 Basics

At its simplest, object-oriented design deals with extracting concepts from, e.g., use cases, and deriving methods and attributes, which are distributed into classes. A further step is to consider interfaces and inheritance. A final design can be achieved through the implementation of architecture styles [Shaw and Garlan, 1996] and design patterns [Gamma et al., 1995]. When attempting to automate the design of object-oriented architecture from concept level, the system requirements must be formalized. After this, the major problem lies within quality evaluation, as many design decisions improve some quality attribute [Losavio et al., 2004] but weaken another. Thus, a sufficient set of quality estimators should be used, and a balance should be found between them. Re-designing software architectures automatically is slightly easier than building architecture from the very beginning, as the initial model already exists, and it merely needs to be ameliorated. However, implementing design patterns is never straightforward, and measuring their impact on the quality of the system is difficult. For more background on software architectures, see, e.g., Bass et al. [1998].

Approaches to search-based software design include improving the reusability of existing software architectures through design patterns [Amoui et al., 2006], building hierarchical decompositions for a software system [Lutz, 2001], designing a class structure [Bowman et al., 2008; Simons and Parmee 2007a; Simons and Parmee, 2007b] and completely designing a software architecture containing some design patterns, based on requirements [Räihä et al., 2008a; Räihä et al., 2008b]. Studies have also been made on identifying concept boundaries and thus automating software comprehension [Gold et al., 2006], re-packaging software [Bodhuin et al., 2007], which can be seen as finding working subsets of an existing architecture, and composing behavioral models for autonomic systems [Goldsby and Chang, 2008; Goldsby et al., 2008], which give a dynamic view of software architecture. The fundamentals of each study are collected in Table 1.

3.1.2 Approaches

Amoui et al. [2006] use the GA approach to improve the reusability of software by applying architecture design patterns to a UML model. The authors' goal is to find the best sequence of transformations, i.e., pattern implementations. Used patterns come from the collection presented by Gamma et al. [1995], most of which improve the design quality and reusability by decreasing the values of diverse coupling metrics while increasing cohesion.

Chromosomes are an encoding of a sequence of transformations and their parameters. Each individual consists of several supergenes, each of which represents a single transformation. A supergene is a group of neighboring genes on a chromosome which are closely dependent and are often functionally related. Only certain combinations of the internal genes are valid. Invalid patterns possibly produced are found and discarded.

Mutation randomly selects a supergene and mutates a random number of genes, inside the supergene. After this, validity is checked. In case of encountering a transformed design which contradicts with object-oriented concepts, for example, a cyclic inheritance, a zero fitness value is assigned to chromosome.

Two different versions of crossover are used. First is a single-point crossover applied at supergene level, with a randomly selected crossover point, which swaps the supergenes beyond the crossover point, but the internal genes of supergenes remain unchanged. This combines the promising patterns of two different transformation sequences. The second crossover randomly selects two supergenes from two parent chromosomes, and similarly applies single point crossover to the genes inside the supergenes. This combines the parameters of two successfully applied patterns.

The quality of the transformed design is evaluated, as introduced by Martin [2000], by its “distance from the main sequence” (D), which combines several object-oriented metrics by calculating abstract classes’ ratio and coupling between classes, and measures the overall reusability of a system.

A case study is made with a UML design extracted of some free, open source applications. The GA is executed in two versions. In one version only the first crossover is applied and in second both crossovers are used. A random search is also used to see if the GA outperforms it. Results demonstrate that the GA finds the optimal solution much more efficiently and accurately. From the software design perspective, the transformed design of the best chromosomes are evolved so that abstract packages become more abstract and concrete packages in turn become more concrete. The results suggest that GA is a suitable approach for automating object-oriented software transformations to increase reusability.

Lutz [2001] uses a measure based on an information theoretic minimum description length principle [Shannon, 1948] to compare hierarchical decompositions. This measure is furthermore used as the fitness function for the GA which explores the space of possible hierarchical decompositions of a system.

In hierarchical graphs links can represent such things as dependency relationships between the components of control-flow or data-flow. In order to consider the best way to hierarchically break a system up into components, one needs to know what makes a hierarchical modular decomposition (HMD) of a system better than another. Lutz takes the view that the best HMD of a system is the simplest. In practice this seems to give rise to HMDs in which modules are highly connected internally (high cohesion) and have relatively few connections which cross module boundaries (low coupling), and thus seems to achieve a principled trade-off between the coupling and cohesion heuristics without actually involving either.

For the GA, the genome is a HMD for the underlying system. The chromosomes in the initial population are created by randomly mutating some number of times a particular “seed” individual. The initial seed individual is constructed by modularizing the initial system. Three different mutation operations are used that can all be thought of as operations on the module tree for the HMD. They are: 1. moving a randomly chosen node from where it is in the tree into another randomly chosen module of the tree, 2. modularize the nodes of some randomly chosen module, and 3. remove a module “boundary”. The crossover operator resembles a tree-based crossover operation used in genetic programming and is most easily considered as an operation on the module trees of the two HMDs involved. However, legal solutions are not guaranteed, and illegal ones are repaired with a concatenation operator. The fitness is given as 1/complexity. Among other systems, a real software design is used for testing. A HMD with significantly lower complexity than the original was found very reliably, and the system could group the various components of the system into a HMD exhibiting a very logical (in terms of function) structure.

Bowman et al. [2008] study the use of a multi-objective genetic algorithm (MOGA) in solving the class responsibility assignment problem. The objective is to optimize the class structure of a system through the placement of methods and attributes. The strength Pareto approach (SPEA2) is used, which differs from a traditional GA by containing an archive of individuals from past populations.

The chromosome is represented as an integer vector. Each gene represents a method or an attribute in the system and the integer value in a gene represents the class to which the method or attribute in that locus belongs. Dependency information between methods and attributes is stored in a separate matrix. Mutations are performed by simply changing the class value randomly; the creation of new classes is also allowed. Crossover is the traditional one-point one. There are also constraints: no empty classes are allowed (although the selected encoding method also makes them impossible), conceptually related methods are only moved in groups, and classes must have dependencies to at least one other class.

The fitness function is formed of five different values measuring cohesion and coupling: 1. method-attribute coupling, 2. method-method coupling, 3. method-generalization coupling, 4. cohesive interaction and 5. ratio of cohesive interaction. A complementary measure for common usage is also used.

Selection is made with a binary-tournament selection where the fitter individual is selected 90% of the time.

In the case study an example system is used, and a high-quality UML class diagram of this system is taken as a basis. Three types of modifications are made and finally the modifications are combined in a final test. The efficiency of the MOGA is now evaluated in relation to how well it fixed the changes made to the optimal system. Results show that in most cases the MOGA managed to fix the made modifications and in some cases the resulting system also had a higher fitness value than the original "optimal" system.

Bowman et al. also compare MOGA to other search algorithms, such as random search, hill climbing and a simple genetic algorithm. Random search and hill climbing only managed to fix a few of the modifications and the simple GA did not manage to fix any of the modifications. Thus, it would seem that a more complex algorithm is needed for the class responsibility assignment problem.

Simons and Parmee [2007a; 2007b] 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.

A single design solution is a chromosome. In a mutation, a single individual is mutated by locating an attribute and a method from one class to another. For crossover two individuals are chosen at random from the population and their attributes and methods are swapped based on their class position within the individuals. Cohesiveness of methods (COM) is used to measure fitness, fitness for class C is defined as $f(\mathrm{C}) = 1/(|\mathrm{Ac}||\mathrm{Mc}|)*\sum(\Delta_{ij})$, where Ac stands for the number of attributes and Mc for the number of methods in class C, and $\Delta_{ij} = 1$, if method j uses attribute I, and 0 otherwise. Selection is performed by tournament and roulette-wheel. In an alternative approach, categorized by the authors as *evolutionary programming (EP)* and inspired by Fogel et al. [1966], offspring is created by mutation and selection is made with tournament selection. Two types of mutations are used, class-level mutation and element-level mutation. At class level, all attributes and methods of a class in an individual are swapped as a group with another class selected at random. At element level, elements (methods and attributes) in an individual are swapped at random from one class to another. Initialization of the population is made by allocating a number of classes to each individual design at random, within a range derived from the number of attributes and methods. All attributes and methods from sets of attributes and methods are then allocated to classes within individuals at random.

A case study is made with a cinema booking system with 15 actions, 16 datas and 39 uses. For GA, the average COM fitness for final generation for both tournament and roulette-wheel is similar, as is the average number of classes in the final generation.

However, convergence to a local optimum is quicker with tournament selection. Results reveal that the average and maximum COM fitness of the GA population with roulette-wheel selection lagged behind tournament in terms of generation number. For EP, the average population COM fitness in the final generation is similar to that achieved by the GA.

The initial average fitness values of the three algorithms are notably similar, although the variance of the values increases from GA tournament to GA roulette-wheel to EP. In terms of COM cohesion values, the generic operators produced conceptual software designs of similar cohesion to human performance. Simons and Parmee suggest that a multi-objective search may be better suited for support of the design processes of the human designer. To take into account the need for extra input, they attempted to correct the fitness function by multiplying the COM value by a) the number of attributes and method in the class (COM.M+A); b) the square root of the number of attributes and methods in the class (COM.√(M+A); c) the number of uses in the class (COM.uses) and d) the square root of the number of uses in a class (COM. √uses).

The number of classes in a design solution is measured and a design solution with higher number of classes is preferred to a design solution with fewer classes. When cohesion metrics that take class size into account are used, there is a broad similarity between the average population cohesion fitness and the manual design. Values achieved by the COM.M+A and COM.uses and cohesion metrics are higher than the manual design cohesion values, while COM.√(M+A)and COM.√uses values are lower. Manually examining the design produced by the evolutionary runs, a difference is observed in the design solutions produced by the four metrics that account for class size, when compared with the metrics that do not. From the results produced for the two case studies, it is evident that while the cohesion metrics investigated have produced interesting cohesive class design solutions, they are by no means a complete reflection of the inherently multi-objective evaluations conducted by a human designer. The evolutionary design variants produced are thus highly dependent on the extent and choice of metrics employed during search and exploration.

Räihä et al. [2008a] take the design of software architecture a step further than Simons and Parmee [2007a] by starting the design from a responsibility dependency graph. The graph can also be achieved from use cases, but the architecture is developed further than the class distribution of actions and data. A GA is used for the automation of design.

In this approach, 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 [Gamma et al. 1995] or an interface, or splitting or joining class(es). Implemented design patterns are Façade and Strategy, as well as the message dispatcher architecture style [Shaw and Garlan, 1996]. Dynamic mutation probabilities are used to encourage the application of basic design choices (the architectural style(s)) in the beginning and more refined choices (such as the Strategy pattern) in the end of evolution. Crossover is a standard one-point crossover. The offspring and mutated chromosomes are always checked after the operations for legality, as design patterns may easily be broken. Selection is made with the roulette wheel method.

The fitness function is a combination of object-oriented software metrics, most of which are from the Chidamber and Kemerer [1994] collection, which have been grouped to measure quality concepts efficiency and modifiability. Some additional metrics have also been developed to measure the effect of communicating through a message dispatcher or interfaces. Furthermore, a complexity measure is introduced. The fitness function is defined as $f = w_1\text{PositiveModifiability} - w_2\text{NegativeModifiability} + w_3\text{PositiveEfficiency} - w_4\text{NegativeEfficiency} - w_5\text{Complexity}$, where w_is are weights to be fixed.

The approach is tested on a sketch of a medium-sized system [Räihä, 2008]. Results show positive development in overall fitness value, while the balancing of weights greatly affects whether the design is more modifiable or efficient.

Räihä et al. [2008b] further develop their work by implementing more design patterns and an alternative approach. In addition to the responsibility dependency graph, a domain model may be given as input. The GA can now be utilized in Model Driven Architecture design, as it takes care of the transformations from Computationally Independent Model to Platform Independent Model. The new design patterns are Mediator and Proxy, and the service oriented architecture style is also implemented by enabling a class to be called through a server. The chromosome representation, mutation and crossover operations and selection method are kept the same. Results show that the fitness values converge to some optima and reasonable high-level designs are obtained.

Kessentini et al. [2008] have also used a search-based approach to model transformations. They start with a small set of examples from which transformation blocks are extracted and use particle swam optimization (PSO) [Kennedy and Eberhart, 1995]. A model is viewed as a triple of source model, target model and mapping blocks between the source and target models. The source model is formed by a set of constructs. The transformation is only coherent if it does not conflict the constructs. The transformation quality of a source model (i.e., global quality of a model) is the sum of the transformation qualities of its constructs (i.e., local qualities).

To encode a transformation an *M*-dimensional search space is defined, *M* being the number of constructs. The encoding is now an *M*-dimensional integer vector whose elements are the mapping blocks selected for each construct. The fitness function is a sum of constructs that can be transformed by the associated blocks multiplied by relative numbers of matched parameters and constructs. The fitness value is normalized by dividing it with 2**M*, thus resulting in a fitness range of [0, 1].

The method was evaluated and experimented with 10 small-size models, of which nine are used as a training set and one as the actual model to be transformed. The precision of model transformation (number of constructs with correct transformations in relation to total number of constructs) is calculated in addition to the fitness values. The best solution was found already after 29 iterations, after which all particles converged to that solution. The test generated 10 transformations. The average precision of these is more than 90%, thus indicating that the transformations would indeed give an optimal result, as the fitness value was also high within the range. The test also showed that some constructs were correctly transformed although there were no transformation examples available for these particular constructs.

Gold et al. [2006] experiment with techniques to integrate boundary overlapping concept assignment using Plausible Reasoning. Hill climbing and GA approaches are investigated. The fixed boundary Hypothesis Base Concept Assignment (HBCA) technique is compared to the new algorithms.

A concept may take the form of an action or object. For each concept, a hypothesis is generated and stored. The list of hypotheses is ordered according to the position of the indicators in the source code. The input for search problem is the hypothesis list. The problem is defined as searching for segments of hypothesis in each hypothesis list according to predetermined fitness criteria such that each segment has the following attributes: each segment contains one or more neighboring hypotheses and there are no duplicate segments.

A chromosome is made up of a set of one or more segments representations. All segments with the same winning concept that overlap are compared and all but the fittest segment are removed from the solution. Tournament selection is used for crossover and mutation. Mutation in GA randomly replaces any hypothesis location within any segment pair with any other valid hypothesis location with the concern for causing the search to become overly randomized. In HC the mutation generates new solutions by selecting a segment pair and increasing or decreasing one of the location values by a single increment. The proposed HC takes advantage of the crossover for GA for the restart mechanism, which recombines all segment pairs to create new segment pairs, which are then added to the current solution if their inclusion results in an improvement to the fitness value. Crossover utilizes the location of the segment pairs, where only segment pairs of overlapping locations are recombined and the remaining are copied to the new chromosome.

The fitness criteria's aims are finding segments of strongest evidence and binding as many of the hypotheses within the hypothesis list as possible without compromising the segment's strength of evidence. The segmentation strength is a combination of the inner fitness and the potential fitness of each segment. The inner fitness fit_i of a segment is defined as $signal_i - noise_i$, where $signal_i$ is the signal level, i.e., the number of hypotheses within the segment that contribute to the winner, and $noise_i$ represents the noise level, i.e., the number of hypotheses within the segment that do not contribute to the winner. In addition, each segment is evaluated with respect to the entire segment hypothesis list: the potential segment fitness, fit_p, is evaluated by taking account of $signal_p$, the number for hypotheses outside of the segment that could contribute to the segment's winning concept if they were included in the segment. The potential segment fitness is thus defined as $fit_p = signal_i – signal_p$. The overall segment fitness is defined as $segfit = fit_i + fit_p$. The total segment fitness is a sum of segment fitnesses. The fitness is normalized with respect to the length of the hypothesis list.

An empirical study is used. Results are also compared to sets of randomly generated solutions for each hypothesis list, created according to the solutions structure. The results from GA, HC and random experiment are compared based on their fitness values. The GA fitness distribution is the same as those of HC and random, but achieves higher values. HC is clearly inferior. Comparing GA, HC and HBCA shows a lack of solutions with low Signal to Noise ratios for GA and HC when compared to HBCA. GA is identified as the best of the proposed algorithms for concept assignment which allow overlapping concept boundaries. Also, the HC results are somewhat disappointing as they are found to be significantly worse than GA and random solutions. However, HC produces stronger results than HBCA on the signal to size measure. The GA and HC are found to consistently produce stronger concept than HBCA.

Bodhuin et al. [2007] present an approach based on GAs and an environment that, based on previous usage information of an application, re-packages it with the objective of limiting amount of resources transmitted for using a set of application features. The overall idea is to cluster together (in jars) classes that, for a set of usage scenarios, are likely to be used together. Bodhuin et al. propose to cluster together classes according to dynamic information obtained from executing a series of usage scenarios. The approach aims at grouping in jars classes that are used together during the execution of a scenario, with the purpose of minimizing the overall jar downloading cost, in terms of time in seconds for downloading the application. After having collected execution trace, the approach determines a preliminary re-packaging considering common class usages and then improves it by using GAs.

The proposed approach has four steps. First, the application to be analyzed is instrumented, and then it is exercised by executing several scenarios instantiated from use cases. Second, a preliminary solution of the problem is found, grouping together classes used by the same set of scenarios. Third, GAs are used to determine the (sub)-optimal set of jars. Fourth, based on the results of the previous steps, jars are created.

For the GA, an integer array is used as chromosome representation, where each gene represents a cluster of classes. The initial population is composed randomly. Mutation selects a cluster of classes and randomly changes its allocation to another jar archive. The crossover is the standard one-point crossover. The fitness function is $F(\mathrm{x}) = 1/N*\sum(Cost_i)$ where N is the number of scenarios. 10% of the best individuals are kept alive across subsequent generations. Individuals to be reproduced are selected using a roulette-wheel selection.

Results show that GA does improve the initial packaging, by 60-90 % to the actual initial packaging and by 5-43% compared to a packaging that contains two jars, “used” and “unused”, and by 13-23% compared to the preliminary optimal solution. When delay increases, the GA optimization starts to be highly more useful than the preliminary optimal solution, while the “used” packaging becomes better. However, for network delay value lower of slightly higher than the value used for the optimization process, the GA optimization is always the best packaging option. It is found that even when there is a large corpus of classes used in all scenarios, a cost reduction is still possible, even if in such a case the preliminary optimized solution is already a good one. The benefits of the proposed approach depend strongly on several factors, such as the amount of collected dynamic information, the number of scenarios subjected to analysis, the size of the common corpus and the networks delay.

Goldsby and Chang [2008] and Goldsby et al. [2008] study the digital evolution of behavioral models for autonomic systems with Avida. In digital evolution a population of self-replicating computer programs (digital organisms) exists in a computational environment and is subject to mutations and selection. In this approach each digital organism is considered as a generator for a UML state diagram describing the systems behavior.

Each organism is given instinctual knowledge of the system in the form of a UML class diagram representing the system structure, as well as optional seed state diagrams. A genome is thus seen as a set of instructions telling how the system should behave. The genome is also capable of replicating itself. In fact, in the beginning of each population there exists only one organism that only knows how to replicate itself, thus creating the rest of the population. Mutations include replacing an instruction, inserting an additional instruction and removing an instruction from the genome. As genomes are self-replicating, crossover is not used in order to create offspring.

The fitness or quality of an organism is evaluated by a set of tasks, defined by the developer. Each task that the behavioral model is able to execute increases its merit. The higher a merit an organism has, the more it will replicate itself, eventually ending up dominating the population.

A behavioral model of an intelligent robot is used as a case study for Avida. Through a 100 runs of Avida, seven behavioral models are generated for the example system. Post-evolution analysis includes evaluation with the following criteria: minimum states, minimum transitions, fault tolerance, readability and tolerance. After the analysis, one of the models meets all but one criterion (safety) and three models meet three of the five criteria. One model does not meet any of the additional criteria. Thus, the produced behavioral models would seem to be of quality in average.

Table 1. Studies in search-based object-oriented software architecture design

Author	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Amoui et al. [2006]	Applying design patterns; high level architecture design	Software system	Chromosome is a collection of supergenes, containing information of pattern transformations	Implementing design patterns	Single-point crossovers for both supergene level and chromosome level, with corrective function	Distance from main sequence	Transformed system, design patterns used as transformations to improve modifiability	New concept of supergene used
Lutz [2001]	Information theory applied in software design; high-level architecture design	Software system	Hierarchical modular decomposition (HMD)	Three mutations operating the module tree for the HMD	A variant of tree-based crossovers, as used in GP, with corrective function	1/complexity	Optimal hierarchical decomposition of system	
Bowman et al. [2008]	Class structure design is (semi-) automated	Class diagram as methods, attributes and associations	Integer vector and a dependency matrix	Randomly change the class of method or attribute	Standard one-point	Cohesion and coupling	Optimal class structure	Comparison between different algorithms
Simons and Parmee [2007a; 2007b]	Class structure design is automated	Use cases; data assigned to attributes and actions to methods	A design solution where attributes and methods are assigned to classes	An attribute and a method are moved from one class to another	Attributes and methods of parents are swapped according to class position	Cohesiveness of methods (COM)	Basic class structure for system.	Design solutions encoded directly into a programming language

Author	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Räihä et al. [2008a]	Automating architecture design	Responsibility dependency graph	Chromosome is a collection of supergenes, containing information of responsibilities and design patterns	Mutations apply architectural design patterns and styles	A standard one-point crossover with corrective function	Efficiency, modifiability and complexity	UML class diagram depicting the software architecture	
Räihä et al. [2008b]	Automating CIM-to-PIM model transformations	Responsibility dependency graph and domain model (CIM model)	Chromosome is a collection of supergenes, containing information of responsibilities and design patterns	Mutations apply architectural design patterns and styles	A standard one-point crossover with corrective function	Efficiency, modifiability and complexity	UML class diagram depicting the software architecture (PIM model)	
Gold et al. [2006]	Using GA in the area of concepts	Hypothesis list for concepts	One or more segment representations	A hypothesis location is randomly replaced within a segment pair	Segment pairs of overlapping locations are combined, rest copied	Strongest evidence for segments and hypothesis binding	Optimized concept assignment	Hill climbing used as well as GA
Bodhuin et al. [2007]	Automating class clustering in jar archives	A grouping of classes of a system	An integer array, each gene is a cluster of classes allocated to the jar represented by integer	Changes the allocation of a class cluster to another jar archive	Standard one-point	Download cost of jar archive	Optimal packaging; finding the subsets of classes most likely to be used together (to be placed in same jar archive)	
Goldsby and Chang [2008]; Goldsby et al. [2008]	Designing a system from a behavioral point of view	A class diagram, optional state diagram	A set of behavioral instructions	Changes, removes or adds an instruction	Self-replication	Number of executed tasks	UML state diagram giving the behavioral model of system	No actual evolutionary algorithm used, but a platform that is “an instance of evolution”

3.2 Service-oriented architecture design

3.2.1. Basics

Web services are rapidly changing the landscape of software engineering, and service-oriented architectures (SOA) are especially popular in business. One of the most interesting challenges introduced by web services is represented by Quality Of Service (QoS)-aware composition and late-binding. This allows to bind, at run-time, a service-oriented system with a set of services that, among those providing the required features, meet some non-functional constraints, and optimize criteria such as the overall cost or response time. Hence, QoS-aware composition can be modeled as an optimization problem. This problem is NP-hard, which makes it suitable for meta-heuristic search algorithms. For more background on SOA, see, e.g., Huhns and Sting [2005]. The following subsection describes several approaches that have used a GA to deal with optimizing service compositions. The fundamentals of each approach are collected in Table 2.

3.2.2. Approaches

Canfora et al. [2005a] propose a GA to deal with optimizing service compositions. The approach attempts to quickly determine a set of concrete services to be bound to the abstract services composing the workflow of a composite service. Such a set needs both to meet QoS constraints, established in the Service Level Agreement (SLA), and to optimize a function of some other QoS parameters.

A composite service S is considered as a set of n abstract services $\{s_1, s_2,\ldots, s_n\}$, whose structure is defined through some workflow description language. Each component s_j can be bound to one of the m concrete services, which are functionally equivalent. Computing the QoS of a composite service is made by combining calculations for quality attributes time, cost, availability, reliability and custom attraction. Calculations take into account Switch, Sequence, Flow and Loop patterns in the workflow.

The genome is encoded as an integer array whose number of items equals to the number of distinct abstract services composing the services. Each item, in turn, contains an index to the array of the concrete services matching that abstract service. The mutation operator randomly replaces an abstract service with another one among those available, while the crossover operator is the standard two-point crossover. Abstract services for which only one concrete service is available are taken out from the GA evolution.

The fitness function needs to maximize some QoS attributes, while minimizing others. In addition, the fitness function must penalize individuals that do not meet the constraints and drive the evolution towards constraint satisfaction D. The fitness function is $f = (w_1\text{Cost} + w_2\text{Time})/ (w_3\text{Availability} + w_4\text{Reliability}) + w_5D$. QoS attributes are normalized in the interval [0, 1). The weights $w_1,\dots,w_5$ are real, positive weights of the different fitness factors.

A dynamic penalty is experimented with, so that w_5 is increased over the generations, w_5* generations/maximum generations. An elitist GA is used where the best two individuals are kept alive across generations. Roulette wheel method is used for selection.

The GA is able to find solutions that meet the constraints, and optimizes different parameters (here cost and time). Results show that the dynamic fitness does not outperform the static fitness. Even different calibrations of weights do not help. The results of GA and IP are compared by comparing the convergence times of Integer Programming (IP) [Garfinkel and Nemhauser, 1972] and GA for the (almost) same achieved solution. The results show that when the number of concrete services is small, IP outperforms GA. For about 17 concrete services, the performance is about the same. After that, GA clearly outperforms IP.

Canfora et al. [2005b] have continued their work by using a GA in replanning the binding between a composite service and its invoked services during execution. Replanning is triggered once it can be predicted that the actual service QoS will differ from initial estimates. After this, the slice, i.e., the part of workflow still remaining to be executed is determined and replanned. The used GA approach is the same as earlier, but additional algorithms are used to trigger replanning and computing workflow slices. The GA is used to calculate the initial QoS-values as well as optimizing the replanned slices. Experiments were made with realistic examples and results concentrate on the cost quality factor. The algorithms managed to reduce the final cost from the initial estimate, while response time increased in all cases. The authors end with a note that the trade-off between response time and cost quality factors need to be examined thoroughly in the future.

Jaeger and Mühl [2007] discuss the optimization problem when selecting services while considering different QoS characteristics. A GA is implemented and tested on a simulation environment in order to compare its performance with other approaches.

An individual in the implemented GA represents an assignment of a candidate for each task and can be represented by a tuple. A population represents a set of task-candidate assignments. The initial population is generated arbitrarily from possible combinations of tasks and candidates. Mutation changes a particular task-candidate assignment of an individual. Crossover is made by combining two particular task-candidate assignments to form new ones. The fitness value is computed based on the QoS resulting from the encoded task-services assignment and is defined as f = Penalty* (availability*reputation)/(cost*time). Simple additive weighting is applied to compute a normalized value aggregated from the four different QoS-values resulting from each solution.

A trade-off couple between execution time and cost is defined as follows: the percentage a, added to the optimal execution time, is taken to calculate the percentage b, added to the optimal cost, with $a + b = 100$. Thus, the shorter the execution time is, the worse will be the cost and vice versa. The constraint is determined to perform the constraint selection on the execution time first. The aggregated cost for the composition is increased by 20% and then taken as the constraint that has to be met by the selection.

Several variations of the fitness function are possible. Jaeger and Mühl use a multiplication of the fitness to make the difference between weak and strong fitnesses larger. When the multiplying factor is 4, it achieves higher QoS values than those with a smaller factor; however, a factor of 8 does not achieve values as high. The scaled algorithm performed slightly better than the one with a factor of 2, and behaved similarly to the weighted algorithm. The penalty factor was also investigated, and it was varied between 0.01 and 0.99 in steps of 0.01. The results show that a factor of 0.5 would result in few cases where the algorithm does not find a constraint meeting solution. On the other hand, solutions below 0.1 appear too strong, as they represent an unnecessary restriction of the GA to evolve further invalid solutions.

The GA offers a good performance at feasible computational efforts when compared to, e.g., bottom-up heuristics. However, this approach shows a large gap when compared to the resulting optimization of a branch-and-bound approach or to exhaustive search. It appears that the considered setup of values along with the given optimization goals and constraints prevent a GA from efficiently identifying very near optimal solutions.

Zhang et al. [2006] implement a GA that, by running only once, can construct the composite service plan according with the QoS requirements from many services compositions. This GA includes a special relation matrix coding scheme (RMCS) of chromosomes proposed on the basis of the characters of web services selection.

By means of the particular definition, it can represent simultaneously all paths of services selection. Furthermore, the selected coding scheme can denote simultaneously many web service scenarios that the one dimension coding scheme can not express at one time.

According to the characteristic of the services composition, the RMCS is adopted using a neighboring matrix. In the matrix, *n* is the number of all tasks included in services composition. The elements along the main diagonal for the matrix express all the abstract service nodes one by one and are arranged from the node with the smallest code number to the node with the largest code number. The objects of the evolution operators are all elements along the main diagonal of the matrix. The chromosome is made up of these elements. The other elements in the matrix are to be used to check whether the created new chromosomes by the crossover and mutation operators are available and to calculate the QoS values of chromosomes.

The policy for initial population attempts to confirm the proportion of chromosomes for every path to the size of the population. The method is to calculate the proportion of compositions of every path to the sum of all compositions of all paths. The more there are compositions of one path, the more chromosomes for the path are in the population.

The value of every task in every chromosome is confirmed according to a local optimized method. The larger the value of QoS of a concrete service is, the larger the probability to be selected for the task is. The roulette wheel selection is used to select concrete services for every task.

The probability of mutation is for the chromosome instead of the locus. If mutation occurs, the object path will be confirmed firstly whether it is the same as the current path expressed by the current chromosome. If the paths are different, the object path will be selected from all available paths except the current one. If the object is itself, the new chromosome will be checked whether it is the same as the old chromosome. Same chromosome will result in the mutation operation again. If the objects are different paths from the current path, a new chromosome will be related on the basis of the object path.

A check operation is used after the invocations of crossover and mutation. If the values of the crossover loci in two crossover chromosomes are all for the selected web services, the new chromosomes are valid. Else, the new chromosomes need to be checked on the basis of the relation matrix. Mutation checks are needed if changed from selected web service to a certain value or vice versa.

Zhang et al. compared the GA with RMCS to a standard GA with the same data, including workflows of different sizes. The used fitness function is as defined by Canfora et al. [2004]. The coding scheme, the initial population policy and the mutation policy are the different points between the two GAs. Results show that the novel GA outperforms the standard one in terms of achieved fitness values. As the number of tasks grows, so does the difference between fitness values (and performance time, in the favor of the standard solution) between the two GAs. The weaknesses of this approach are thus long running time and slow convergence. Tests on the initial population and the mutation policies show that as the number of tasks grows, the GA with RMCS outperforms the standard one more clearly.

Zhang et al. report that experiments on QoS-aware web services selection show that the GA with the presented matrix approach can get a significantly better composite service plan than the GA with the one dimension coding scheme, and that the QoS policies play an important role in the improvement of the fitness of GA.

Su et al. [2007] continue the work of Zhang et al. [2006] by proposing improvements for the fitness function and mutation policy. An objective fitness function 1 (OF1) is first defined as a sum of quality factors and weights, providing the user with a way to show favoritism between quality factors. The sum of positive quality factors is divided by the sum of negative quality factors. The second fitness function (OF2) is a proportional one and takes into account the different ranges of quality value. The third fitness function (OF3) combines OF1 and OF2, producing a proportional fitness function that also expresses the differences between negative and positive quality factors.

Four different mutation policies are also inspected. Mutation policy 1 (MP1) operates so that the probability of the mutation is tied to each locus of a chromosome. Mutation policy 2 (MP2) has the mutation probability tied to the chromosomes. Mutation policy 3 (MP3) has the same principle as MP1, except that now the child may be identical to the parent. Mutation policy 4 (MP4) has the probability tied to each locus, and has an equal selection probability for each concrete service and the “0” service.

Experiments with the different fitness functions suggest that OF3 clearly outperforms OF1 and OF2 in terms of the reached average maximum fitness value. Experiments on the different mutation policies show that MP1 gains the largest fitness values while MP4 performs the worst.

Cao et al. [2005a; 2005b] present a GA that is utilized to optimize a business process composed of many service agents (SAg). Each SAg corresponds to a collection of available web services provided by multiple-service providers to perform a specific function. Service selection is an optimization process taking into account the relationships among the services. Better performance is achieved using GA compared to using local service selection strategy.

A service selection model using GA is proposed to optimize a business process composed of many service agents. An individual SAg corresponds to a collection of available web services provided by multiple service providers to perform a specific function. When only measuring cost, the service selection is equivalent to a single-objective optimization problem.

An individual is generated for the initial population by randomly selecting a web service for each SAg of the services flow, and the newly generated individual is immediately checked whether the corresponding solution satisfies the constraints. If any of the constraints is violated, then the generated individual is regarded as invalid and discarded. The roulette wheel selection is used for individuals to breed.

Mutation bounds the selected SAg to a different web service than the original one. After an offspring is mutated, it is also immediately checked whether the corresponding solution is valid. If any constraints are violated, then the mutated offspring is discarded and the mutation operation is retried.

A traditional single-point crossover operator is used to produce two new offspring. After each crossover operation, the offspring are immediately checked whether the corresponding solutions are valid. If any of the constraints is violated, then both offspring are discarded and the crossover operation for the mated parents is retried. If valid offspring still cannot be obtained after a certain number of retries, the crossover operation for these two parents is given up to avoid a possible infinite loop.

Cao et al. take cost as the primary concern of many business processes. The overall cost of each execution path can always be represented by the summation cost of its subset components. For GA, integer encoding is used. The solution to service selection is encoded into a vector of integers. The fitness function is defined as $f = U - \sum$(costs of service flows), if cost<U, and otherwise 0. The constant U should be selected as an appropriate positive number to ensure the fitness of all good individuals get a positive fitness value in the feasible solution space. On the other hand, U can also be utilized to adjust the selection pressure of GA.

In the case study the best fitness of the population has a rapid increase at the beginning of the evolution process and then convergences slowly. It means the overall cost of the SAg is generally decreasing with the evolution process. For better solutions, the whole optimization process can be repeated for a number of times, and the best one in all final solutions is selected as the ultimate solution to the service selection problem.

Table 2. Studies in search-based service-oriented software architecture design

Author	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Canfora et al. [2005a]	Service composition with respect to QoS attributes	Sets of abstract and concrete services	Integer array, size is the number of abstract services, each item contains an index to array of concrete services	Randomly replaces an abstract service with another	Standard two-point crossover	Minimize cost and time, maximize availabity and reliabiliy, meet constraints	Optimized service composition meeting constraints, concrete services bound to abstract services	A dynamic penalty was experimented with
Canfora et al. [2005b]	Replanning during execution time	Sets of abstract and concrete services	Integer array, size is the number of abstract services, each item contains an index to array of concrete services	Randomly replaces an abstract service with another	Standard two-point crossover	Minimize cost and time, maximize availability and reliability, meet constraints	Optimized service composition meeting constraints, concrete services bound to abstract services	GA used to calculate initial QoS-value and QoS-values inbetween: replanning is triggered by other algorithms
Jaeger and Mühl [2007]	Service assignment with respect to QoS attributes	Selection of services and tasks to be carried out	A tuple representing an assignment of a candidate for a task	Changes an individual task-candidate assignment	Combining task-candidate assignments	Minimize cost and time, maximize availability and reliability, with penalty	Tasks assigned to services considering QoS attributes	A trade-off couple between execution time and cost is defined

Author	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Zhang et al. [2006]	Task assignment with relation to QoS attributes	Selections of tasks and services	Relation matrix coding scheme	Standard, with corrective function	Standard, with corrective function	Minimize cost and time, maximize availability and reliability, meet constraints	Tasks assigned to services considering QoS attributes	Initial population and mutation policies defined
Su et al. [2007]	Task assignment with relation to QoS attributes	Selections of tasks and services	Relation matrix coding scheme	Standard, with corrective function	Standard, with corrective function	Minimize cost and time, maximize availability and reliability, meet constraints	Tasks assigned to services considering QoS attributes	Initial population and mutation policies defined
Cao et al. [2005a; 2005b]	Business process optimization	Collections of web services and service agents (SAg) composing a business process	Integer encoding, assigning a SAg to a service	Changes the service to which a SAg is bound with corrective function	Standard one-point, producing two new offspring with corrective function	Cost	Services assigned to service agents	

3.3. Other

3.3.1 Background

In addition to purely designing software architecture, there are some factors that should be optimized, regardless of the particularities of an architecture. Firstly, there is the reliability-cost tradeoff. The reliability of software is always dependent on its architecture, and the different components should be as reliable as possible. However, the more work is put to ensure reliability of different components, the more the software will cost. Wadekar and Gokhale [1999] implement a GA to optimize the reliability-cost tradeoff. Secondly, there are some parameters, e.g., tile sizes in loop tiling and loop unrolling, which can be optimized for all software architectures in order to optimize the performance of the software. Che et al. [2000] apply search-based techniques for such parameter optimization.

3.3.2 Approaches

Wadekar and Gokhale [1999] present an optimization framework founded on architecture-based analysis techniques, and describe how the framework can be used to evaluate cost and reliability tradeoffs using a GA. The methodology for the reliability analysis of a terminating application is based on its architecture. The architecture is described using the one-step transition probability matrix P of a discrete time Markov chain (DTMC).

Wadekar and Gokhale assume that the reliabilities of the individual modules are known, with R_i denoting the reliability of module i. It is also assumed that the cost of the software consisting of n components, denoted by C, can be given by a generic expression of the form: $C = C_1(R_1) + C_2(R_2) + \ldots + C_n(R_n)$ where C_i is the cost of component i and the cost C_i depends monotonically on the reliability R_i. Thus the problem of minimizing the software cost while achieving the desired reliability is the problem of selecting module reliabilities.

A chromosome is a list of module reliabilities. Each member in the list, a gene, corresponds to a module in the software. The independent value in each gene is the reliability of the module it represents, and the dependent value is the module cost given by the module cost-reliability relation or a table known *a priori*. The gene values are changed to alter the cost and reliability of a software implementation represented by a particular chromosome.

Mutation and crossover operations are standard. To avoid convergence to a local optimum as the population size increases, the mutation operation is used more frequently. A cumulative-probability based basic selection mechanism is used for selection. Chromosomes are ranked by fitness and divided into rank groups. The probability of selection of chromosomes varies uniformly according to their rank group where chromosomes in the first rank group have the largest probability. A new generation of the population is created by selecting $p_{\mathrm{imax}}/2$ chromosomes, where p_{imax} is maximum population. If the cost reduction is less than or equal to φ% of the current best cost τ number of times, the GA terminates. During any generation cycle if the cost reduction is larger, the counter τ is reset to 0. The reduction percentage factor φ and the counter limit τ are parameters.

The fitness function is $f = (-K/\ln R)/C^{\gamma}$, where K is a large positive constant. The fitness of solutions increases superlinearly with their reliability. The constant γ is used to linearize the cost variation. The maximum fitness is directly proportional to K. An intermediate value of gamma, $\gamma = 1.5$, allows the GA to distinguish between low-cost and high-cost solutions, while selecting a sufficient number of high-cost high-reliability solutions, that may generate the optimal high-reliability low-cost solution.

Wadekar and Gokhale compare the GA against exhaustive search. The results indicate that the GA consistently and efficiently provides optimal or very close to optimal designs, even though the percentage of such designs in the overall feasible design space is extremely small. The results also highlight the robustness of the GA. The case study results show how the GA can be effectively used to select components such that the software cost is minimized, for various cost structures.

Che et al. [2003] present a framework for performance optimization parameter selection, where the problem is transformed into a combinatorial minimization problem. Many performance optimization methods depend on right optimization parameters to get good performance for an application. Che et al. search for the near optimal optimization parameters in a manner that is adaptable for different architectures. First a reduction transformation is performed to reduce the program's runtime while maintaining its relative performance as regard to different parameter vectors. The near-optimal optimization parameter vector based on the reduced program's real execution time is searched by GA, which converges to a near-optimal solution quickly. The reduction transformation reduces the time to evaluate the quality of each parameter vector.

First some transformations are applied to the application, leaving the optimization parameter vector to be read from a configuration file. Second, the application is complied into executable with the native compiler. Then the framework repeatedly generates the configure file with a different parameter vector selected by search and measures the executable's runtime.

The chromosome encoding for the GA is a vector of integer values, with each integer corresponding to an optimization parameter of a solution. No illegal solutions are allowed. The population has a fixed size. A simple integer value mutation is implemented and an integer number recombination scheme is used for crossover. The fitness value reflects the duality of an individual in relation to other individuals. The linear rank-based fitness assignment scheme is used to calculate the fitness values. Selection for a new generation is made by elitism and roulette wheel method. Test results show that the GA can adapt to different execution environments automatically. For each platform, it always selects excellent optimization parameters for 80% programs. Results show that the number of individuals evaluated is far smaller than the size of solution space for each program on each platform. The optimization time is also small.

4. SOFTWARE CLUSTERING

4.1 Basics

As software systems develop and are maintained, they tend to grow in size and complexity. A particular problem is the growing number of dependencies between libraries, modules and components within the modules. Software clustering (or modularization) attempts to optimize the clustering of components into modules in such a way that there are as many dependencies within a module as possible and as few dependencies between modules as possible. This will enhance the understandability of a system, which in turn will make it more maintainable and modifiable. Also, fewer dependencies between modules usually results in better efficiency.

As components or modules (depending on the level of detail in the chosen representation) can be depicted as vertices and dependencies between them as edges in a graph, the software clustering problem can be traced back to a graph partitioning problem, which is NP-complete. Genetic algorithms have successfully been applied to a general graph partitioning problem [Bui and Moon, 1996; Shazely et al., 1998], and thus the related software clustering problem is very suitable for meta-heuristic search techniques.

Although the basic problem is relatively simple to define and the goodness of a modularization can be calculated based on the goodness of the underlying graph partitioning, the nature of software systems provides challenges when defining the actual fitness function for the optimization algorithm. Also, not all necessary information may be encoded into a simple graph representation, and this presents another question to be answered when designing a search-based approach for modularization. The following subsection presents approaches using GAs, HC and SA to find good software modularizations, and the fundamentals of each study are collected in Table 3.

4.2. Approaches

Antoniol et al. [2003] present an approach to re-factoring libraries with the aim of reducing the memory requirements of executables. The approach is organized in two steps: the first step defines an initial solution based on clustering methods, while the second step refines the initial solution via genetic algorithm. Antoniol et al. [2003] propose a GA approach that considers the initial clusters as the starting population, adopts a knowledge-based mutation function and has a multi-objective fitness function. Tests on medium and large-size open source software systems have effectively produced smaller, loosely coupled libraries, and reduced the memory requirement for each application.

The GA is applied to a newly defined problem encoding, where generic mutation may sometimes generate clones. These clones reduce the overall amount of resources required by the executables by removing inter-library dependencies. A multi-objective fitness function is defined, trying to keep low both the number of inter-library dependencies and the average number of objects linked by each application. Given a system composed by applications and libraries, the idea is to re-factor the biggest libraries, splitting them into two or more smaller clusters, so that each cluster contains symbols used by a common subset of applications (i.e., they made the assumption that symbols often used together should be contained in the same library). Given that, for each library to be re-factored, a Boolean matrix MD is composed.

Antoniol et al. have chosen to apply the Silhouette statistic [Kaufman and Rousseeuw, 1990] to compute the optimal number of clusters for each MD matrix. Once the number of clusters is known for each “old library”, agglomerative-nesting clustering was performed on each MD matrix. This builds a dendrogram and a vector of heights that allow identifying a certain number of clusters. These clusters are the new candidate libraries. When given a set of all objects contained into the candidate libraries, a dependency graph is built, and the removal of inter-library dependencies can therefore be brought back to a graph partitioning problem.

The encoding schema indicates each partition with an integer p and represents the genome as an array G, where the integer p in position q means that the function q is contained into partition *p*. The GA is initialized with the encoding of the set of libraries obtained in the previous step.

The mutation operator works in two modes: normally, a random column is taken and two random rows are swapped. When cloning an object, a random position in the matrix is taken; if it is zero and the library is dependent on it, then the mutation operator clones the object into the current library. Of course the cloning of an object increases both linking and size factors, therefore it should be minimized. This GA activates the cloning only for the final part of the evolution (after 66%) of generations in their case studies. This strategy favors dependency minimization by moving objects between libraries; then, at the end, remaining dependencies are attempted to remove by cloning objects. The crossover is a one-point crossover: given two matrices, both are cut at the same random column, and the two portions are exchanged. Population size and number of generations were chosen by an iterative procedure.

The fitness function attempts to balance three factors: the number of inter-library dependencies at a given generation, the total number of objects linked to each application that should be as small as possible, and the size of the new libraries. A unitary weight is set to the first factor, and two weights are selected using an iterative trial-and-error procedure, adjusting them each time until the factors obtained at the final step are satisfactory. The partitioning ratio is also calculated. Case study results show that the GA manages to considerably reduce the amount of dependencies, while the partition ratio stays nearly the same or slightly reduced. The proposed re-factoring process allows obtaining smallest, loosely coupled libraries from the original biggest ones.

Mancoridis et al. [1998] treat automatic modularization as an optimization problem and have created the Bunch tool that uses HC and GA to aid its clustering algorithms. A hierarchical view of the system organization is created based solely on the components and relationships that exist in the source code. The first step is to represent the system modules and the module-level relationships as a module dependency graph (MDG). An algorithm is then used to partition the graph in a way that derives the high-level subsystem structure from the component-level relationships that are extracted from the source code. The goal of this software modularization process is to automatically partition the components of a system into clusters (subsystems) so that the resultant organization concurrently minimizes inter-connectivity while maximizing intra-connectivity. This task is accomplished by treating clustering as an optimization problem where the goal is to maximize an objective function based on a formal characterization of the trade-off between inter- and intra-connectivity.

The clusters, once discovered, represent higher-level component abstractions of a system's organization. Each subsystem contains a collection of modules that either cooperate to perform some high-level function in the overall system or provide a set of related services that are used throughout the system. Intra-connectivity A_i of cluster i consisting of N_i components and m_i intra-edge dependencies as $A_i = m_i/N_i^2$, bound between 0 and 1. Interconnectivity is a measurement of the connectivity between two distinct clusters. A high degree of inter-connectivity is an indication of poor subsystem partitioning. Inter-connectivity E_{ij} between clusters i and j consisting of N_i and N_j components with e_{ij} inter-edge dependencies is 0, if $i = j$, and $e_{ij} / 2{*}N_iN_j$ otherwise, bound between 0 and 1. Modularization Quality (MQ) demonstrates the trade-off between inter- and intra-connectivities, and it is defined for a module dependency graph partitioned into k clusters as $1/k{*}\sum \frac{Ai-1}{k*\frac{k-1}{2}} * \sum Ei,j$ if $k>1$, or A_1 if $k = 1$.

The first step in automatic modularization is to parse the source code and build a MDG. A sub-optimal clustering algorithm works as the traditional hill climbing one by randomly selecting a better neighbor. The GA starts with a population of randomly generated initial partitions and systematically improving them until all of the initial samples converge. The GA uses the “neighboring partition” definition to improve an individual, and thus only contains one mutation operator, which is the same one as used with HC. Selection is done by randomly selecting a percentage of N partitions and improving each one by finding a better neighboring partition. A new population is generated by making N selections, with replacements for the existing population of N partitions. Selections are random and biased in favor of partitions with larger MQs. The algorithm continues until no improvement is seen for t generations or until all of the partitions in the population have converged to their maximum MQ or until the maximum number of generations has been reached. The partition with the largest MQ in the last population is the sub-optimal solution.

Experimentation with this clustering technique has shown good results for many of the systems that have been investigated. The primary method used to evaluate the results is to present an automatically generated modularization of a software system to the actual system designer and ask for feedback on the quality of the results. A case study was made and the results were shown to an expert, who highly appreciated the result produced by Bunch.

Doval et al. [1999] have implemented a more refined GA in the Bunch tool, as it now contains a crossover operator and more defined mutation and crossover rates. The effectiveness of the technique is demonstrated by applying it to a medium-sized software system. For encoding, each node in the graph (MDG) has a unique numerical identifier assigned to it. These unique identifiers define which position in the encoded string will be used to define that node's cluster. Mutation and crossover operators are standard. A roulette wheel selection is used for the GA, complemented with elitism. Fitness function is based on the MQ metric. Crossover rate was 80% for populations of 100 individuals or fewer and 100% for populations of a thousand individuals or more, varying linearly between those values. Mutation rate is $0.004 \log_2(N)$. The MQ values for constant population and generation values were smaller, but fairly close, within 10% to values achieved with final values for population and generation.

Mancoridis et al. [1999] have continued to develop the Bunch tool for optimizing modularization. Firstly, almost every system has a few modules that do not seem to belong to any particular subsystem, but rather, to several subsystems. These modules are called omnipresent, because they either use or are used by a large number of modules in the system. In the improved version users are allowed to specify two lists of omnipresent modules, one for clients and another for suppliers. The omnipresent clients and suppliers are assigned to two separate subsystems.

Secondly, experienced developers tend to have good intuition about which modules belong to which subsystems. However, Bunch might produce results that conflict with this intuition for several reasons. This is addressed with a user-directed clustering feature, which enables users to cluster some modules manually, using their knowledge of the system design while taking advantage of the automatic clustering capabilities of Bunch to organize the remaining modules. Both user-directed clustering and the manual placement of omnipresent modules into subsystems have the advantageous side-effect of reducing the search space of MDG partitions. By enabling the manual placement of modules into subsystems, these techniques decrease the number of nodes in the MDG for the purposes of the optimization and, as a result, speed up the clustering process.

Finally, once a system organization is obtained, it is desirable to preserve as much of it as possible during the evolution of the system. The integration of the orphan adoption technique into Bunch enables designers to preserve the subsystem structure when orphan modules are introduced. An orphan module is either a new module that is being integrated into the system, or a module that has undergone structural changes. Bunch moves orphan modules into existing subsystems, one at a time, and records the MQ for each of the relocations. The subsystem that produces the highest MQ is selected as the parent for the module. This process, which is linear with respect to the number of clusters in the partition, is repeated for each orphan module. Results from a case study support the added features.

Mitchell and Mancoridis [2002; 2006; 2008] have continued to work with the Bunch tool and have further developed the MQ metric. They define MQ as the sum of Clustering Factors for each cluster of the partitioned MDG. The Clustering Factor (CF) for a cluster is defined as a normalized ratio between the total weight of the internal edges and half of the total weight of external edges. The weight of the external edges is split in half in order to apply an equal penalty to both clusters that are connected by an external edge. If edge weights are not provided by the MDG, it is assumed that each edge has a weight of 1. The clustering factor is defined as

$$CF = \text{intra-edges} / (\text{intra-edges} + ½ * \sum(\text{inter-edges})).$$

The measurement is adjusted, as Mitchell and Mancoridis argue that the old MQ tended to minimize the inter-edges that exited the clusters, and not minimize the number of inter-edges in general. The representation also supports weights. The HC algorithm for the Bunch tool has also been enhanced. During each iteration, several options are now available for controlling the behavior of the hill-climbing algorithm. First, the neighboring process may use the first partition that it discovers with a larger MQ as the basis for the next iteration. Second, the neighboring process examines all neighboring partitions and selects the partition with the largest MQ as the basis for the next iteration. Third, the neighboring process ensures that it examines a minimum number of neighboring partitions during each iteration. For this, a threshold n is used to calculate the minimum number of neighbors that must be considered during each iteration of the process. Experience has shown that examining many neighbors during each iteration, so that $n > 75\%$, increases the time the algorithm needs to converge to a solution.

It is observed that as n increases so does the overall runtime and the number of MQ evaluations. However, altering n does not appear to have an observable impact on the overall quality of the clustering results. A simulated annealing algorithm is also made for comparison. Although the simulated annealing implementation does not improve the MQ, it does appear to help reduce the total runtime needed to cluster each of the systems in this case study.

Mitchell and Mancoridis [2003; 2008] continue their work by proposing an evaluation technique for clustering based on the search landscape of the graph being clustered. By gaining insight into the search landscape, the quality of a typical clustering result can be determined. The Bunch software clustering system is examined. Authors model the search landscape of each system undergoing clustering, and then analyze how Bunch produces results within this landscape in order to understand how Bunch consistently produces similar results.

The search landscape is modeled using a series of views and examined from two different perspectives. The first perspective examines the structural aspects of the search landscape, and the second perspective focuses on the similarity aspects of the landscape. The structural search landscape highlights similarities and differences from a collection of clustering results by identifying trends in the structure of graph partitions. The similarity search landscape focuses on modeling the extent of similarity across all of the clustering results.

The results produced by Bunch appear to have many consistent properties. By examining views that compare the cluster counts to the MQ values, it can be noticed that Bunch tends to converge to one or two “basins of attraction” for all of the systems studied. Also, for the real software systems, these attraction areas appear to be tightly paced. An interesting observation can be made when examining the random system with a higher edge density: although these systems converged to a consistent MQ, the number of clusters varied significantly over all of the clustering runs. The percentage of intra-edges in the clustering results indicates that Bunch produces consistent solutions that have a relatively large percentage of intra-edges. Also, the intra-edge percentage increases as the MQ values increase. It seems that selecting a random partition with a high intra-edge percentage is highly unlikely. Another observation is that Bunch generally improves the MQ of real software systems much more and that of random systems with a high edge density. Number of clusters produced compared with number of clusters in the random starting point indicates that the random starting points appear to have a uniform distribution with respect to the number of clusters. The view shows that Bunch always converges to a “basin of attraction” regardless of the number of clusters in the random starting point.

When examining the structural views collectively, the degree of commonality between the landscapes for the systems in the case study is quite similar. Since the results converge to similar MQ values, Mitchell and Mancoridis speculate that the search space contains a large number of isomorphic configurations that produce similar MQ values. Once Bunch encounters one of these areas, its search algorithms cannot find a way to transform the current partition into a new partition with higher MQ. The main observation is that the results produced by Bunch are stable. In order to investigate the search landscape further Mitchell and Mancoridis measure the degree of similarity of the placement of nodes into clusters across all of the clustering runs to see if there are any differences between random graphs and real software systems. Bunch creates a subsystem hierarchy, where the lower levels contain detailed clusters, and higher levels contain clusters of clusters. Results from similarity measures indicate that the results for the real software systems have more in common than the results for random systems do. Results with similarity measures also support the isomorphic “basin of attraction” conjecture proposed.

Mitchell et al. [2000] have developed a two step process for reverse engineering the software architecture of a system directly from its source code. The first step involves clustering the modules from the source code into abstract structures called subsystems. Bunch is used to accomplish this. The second step involves reverse engineering the subsystem-level relations using a formal (and visual) architectural constraint language. Using the reverse engineered subsystem hierarchy as input, a second tool, ARIS, is used to enable software developers to specify the rules and relations that govern how modules and subsystems can relate to each other.

ARIS takes a clustered MDG as input and attempts to find the missing style relations. The goal is to induce a set of style relations that will make all of the use relations well-formed. A relation is well-formed if it does not violate any permission rule described by the style; this is called the edge repair problem. The relative quality of a proposed solution is evaluated by an objective function. The objective function that was designed into the ARIS system measures the well-formedness of a configuration in terms of the number of well-formed and ill-formed relations it contains. The quality measurement $Q(C)$ for configuration C gives a high quality score to configurations with a large number of well-formed use relations and a low quality score to configurations with a large number of ill-formed style relations or large visibility.

Two search algorithms have been implemented to maximize the objective function: HC and edge removal. The HC algorithm starts by generating a random configuration Incremental improvement is achieved by evaluating the quality of neighboring configurations. A neighboring configuration C_n is one that can be obtained by a small modification to the current configuration C. The search process iterates as long as a new C_n can be found such that $Q(C_n) > Q(C)$.

The edge removal algorithm is based on the assumption that as long as there exists at least one solution to the edge repair problem for a system with respect to a style specification, the configuration that contains every possible reparable relation will be one of the solutions. Using this assumption, the edge removal algorithm starts by generating the fully reparable configuration for a given style definition and system structure graph. It then removes relations, one at a time, until no more relations can be removed without making the configuration ill-formed. A case study is made, where the results seem promising as the give intuition to the nature of the system.

Harman et al. [2002] experiment with fitness functions derived from measures of modules granularity, cohesion and coupling for software modularization. They present a new encoding and crossover operator report initial results based on simple component topology. The new representation allows only one representation per modularization and the new crossover operator attempts to preserve building blocks [Salomon, 1998].

Harman et al. [2002] present modularization so that non-unique representations of modularizations artificially increase the search space size, inhibiting search-based approaches to the problem. In their approach modules are numbered, and elements allocated to module numbers using a simple look-up table. Component number one is always allocated to module number one. All components in the same module as component number one are also allocated to module number one. Next, the lowest numbered component, *n*, not in module one, is allocated to module number two. All components into the same module as component number *n* are allocated to module number two. This process is repeated, choosing each lowest number unallocated component as the defining element for the module. This representation must be renormalized when components move as the result of mutation and crossover.

Harman et al.'s crossover operator attempts to preserve partial module allocations from parents to children in an attempt to promote good building blocks. Rather than selecting an arbitrary point of crossover within the two parents, a random parent is selected and one of its arbitrarily chosen modules is copied to the child. The allocated components are removed from both parents. This removal prevents duplication of components in the child when further modules are copied from one or the other parent to the child. The process of selecting a module from a parent and copying to the child is repeated and the copied components are removed from both parents until the child contains a complete allocation. This approach ensures that at least one module from the parents is preserved (in entirety) in the child and that parts of other modules will also be preserved.

The fitness function maximizes cohesion and minimizes coupling. In order to capture the additional requirement that the produced modularization has a granularity (number of modules) similar enough to the initial granularity, a polynomial punishment factor is introduced into the fitness function to reward solutions as they approach the target value for granularity of the modularization. The granularity is normalized to a percentage. The three fitness components are given equal weights.

A standard one-point crossover is also implemented for comparison. The GA with the novel crossover outperforms the one with the traditional one, although it quickly becomes trapped in local optima. Results also show that the novel GA is more sensitive to inappropriate choices of target granularity than any other approach.

Harman et al. [2005] present empirical results which compare the robustness of two fitness functions used for software module clustering: MQ is used exclusively for module clustering and EVM [Tucker et al., 2001] has previously been applied to time series and gene expression data. The clustering algorithm was based upon the Bunch algorithm [Mancoridis et al., 1999] and redefined. Three types of MDGs were studied: real program MDGs, random MDGs and perfect MDGs.

The primary findings are that searches guided by both fitness functions degrade smoothly as noise increases, but EVM would appear to be the more robust fitness function for real systems. Searches guided by MQ behave poorly for perfect and near-perfect module dependency graphs (MDGs). The results of perfect graphs (MDGs) show however, that EVM produces clusterings which are perfect and that the clusterings produced stay very close to the perfect results as more noise is introduced. This is true both for the comparison against the perfect clustering and the initial clustering. By comparison, the MQ fitness function performs much less well with perfect MDGs. Comparing results for random and real MDGs, both fitness functions are fairly robust. Further results show that searches guided by MQ do not produce the perfect clustering for a perfect MDG but a clustering with higher MQ values.

These results highlight a possible weakness in MQ as a guiding fitness function for modularization searches: it may be possible to improve upon it by addressing that issue. The results show that EVM performs consistently better than MQ in the presence of noise for both perfect and real MDGs but worse for random MDGs. The results for both fitness functions are better for perfect or real graphs than random graphs, as expected. As the real programs increase in size, there appears to be a decrease in the difference between the performance of searches guided by EVM and those guided by MQ. The results show that both metrics are relatively robust in the presence of noise, with EVM being the more robust of the two.

Huynh and Cai [2007] present an automated approach to check the conformance of source code modularity to the designed modularity. Design structure matrices (DSMs) are used as a uniform representation and they are automatically clustered and checked for conformance by a GA. A design DSM and source code DSM work at different levels of abstraction. A design DSM usually needs higher level of abstraction to obtain the full picture of the system, while a source code DSM usually uses classes or other program constructs as variables labeling the rows and columns of the matrix. Given two DSMs, one at the design level and the other at the source code level, the GA takes one DSM as the optimal goal and searches for a best clustering method in the other DSM that maximizes the level of isomorphism between the two DSMs. One of the two DSMs is defined as the sample graph, and the other one as a model graph, and finally a conformance criterion is defined.

To determine the conformance of the source code modularity to the high level design modularity the variables of the sample graph are clustered and thus a new graph is formed, which is called the conformance graph. Each vertex of the conformance graph is associated with a cluster of variables from the sample graph. The more conforming the source code modularity is to the design modularity, the closer to isomorphic the conformance graph and the model graph will be. In computing the level of isomorphism between two graphs, the graph edit distance is computed between the graphs.

With the given representation of the problem, a GA is formulated with which the goal is to find the clustering of sample graph vertices such that the conformance graph of these clustered nodes is isomorphic, or almost isomorphic, to the model graph. This is a projection. The algorithm first creates an initial population of random projections. The fitness function is defined as $f = -D - P - \lambda - \varphi$, where D is the graph edit distance, P is a penalty, and λ and φ provide finer differentiation between mappings with the same graph edit distance. The last two functions allow configuring a sample graph so that it can be clustered in different ways, each corresponding to how the design targeted DSM is clustered. The λ is a dissimilarity function used to calculate how separated components from each directory grouping are. If a sample graph node attribute matches a name pattern specified by the user but is not correctly mapped to the model graph vertex then the fitness of the projection is reduced through φ.

The GA is run on two DSM models of an example software. The experiments consistently converge to produce the desired result, although the tool sometimes produces a result that is not the desired view of the source code, even though the graphs are isomorphic, i.e., the result conforms with the model. The experiment shows the feasibility of using a GA to automatically cluster DSM variables and correctly identify links between source code components and high level design components. The results support the hypothesis that it is possible to check the conformance between source code structure and design structure automatically, and this approach has the potential to be scaled for use in large software systems.

Mahdavi et al. [2003a; 2003b] show that results from a set of multiple hill climbs can be combined to locate good “building blocks” for subsequent searches. Building blocks are formed by identifying the common features in a selection of best hill climbs. This process reduces the search space, while simultaneously ‘hard wiring’ parts of the solution. Mahdavi et al. also investigate the relationship between the improved results and the system size.

An initial set of hill climbs is performed and from these a set of best hill climbs is identified according to some “cut off” threshold. Using these selected best hill climbs the common features of each solution are identified. These common features form building blocks for a subsequent hill climb. A building block contains one or more modules fixed to be in a particular cluster, if and only if all the selected initial hill climbs agree that these modules were to be located within the same cluster. Since all the selected hill climbs agree on these choices, it is likely that good solutions will also contain these choices.

The implementation uses parallel computing techniques to simultaneously execute an initial set of hill climbs. From these climbs the authors experiment with various cut off points ranging from selecting the best 10% of hill climbs to the best 100% in steps of 10%. The building blocks are fixed and a new set of hill climbs are performed using the reduced search space. The principal research question is whether or not the identification of building blocks improved the subsequent search.

A variety of experimental subjects were used. Two types of MDG were used: first type contains non-weighted edges, second type has weighted edges. The MQ values were gathered after the initial and the final climbs, and compared for difference. Statistical tests provide some evidence towards the premise that the improvement in MQ values is less likely to be a random occurrence due to the nature of the hill climb algorithm. The improvement is observed for MDGs with and without weighted edges and for all size MDGs.

Larger size MDGs show more substantial improvement when the best initial fitness is compared with the best final fitness values. One reason for observing more substantial improvement in larger MDGs may be attributed to the nature of the MQ fitness measure.

To overcome the limitation that MQ is not normalized, the percentage MQ improvement of the final runs over the initial runs is measured. These statistical tests show no significant correlation between size and improvement in fitness for both weighted and non-weighted MDGs.

The increase in fitness, regardless of number of nodes or edges, tends to be more apparent as the building blocks are created from a smaller selection of individuals. This may signify some degree of importance for the selection process.

Results indicate that the subsequent search is narrowed to focus on better solutions, that better clustering are obtained and that the results tend to improve when the selection cut off is higher. These initial results suggest that the multiple hill climbing technique is potentially a good way of identifying building blocks. Authors also found that although there was some correlation between system size and various measures of the improvement achieved with multiple hill climbing, none of these correlations is statistically significant.

Table 3. Research approaches in search-based software clustering

Name	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Antoniol et al. [2003]	Cluster optimization	System containing applications and libraries	Integer array	Two random rows of a column in matrix are swapped or an object is cloned by changing a value from zero to one	A random column is taken as split point and contents are swapped	Inter-library dependencies, number of object-application links and size of libraries	Optimized clustering, sizes and dependencies between libraries diminished	Optimal number of clusters is calculated for a matrix with the Silhouette statistic
Mancoridis et al. [1998]	Automation of partitioning components of a system into clusters	System given as a module dependency graph (MDG)	MDG	N/A	N/A	Minimize inter-connectivity, maximize intra-connectivity, combined as modulariztion quality (MQ)	Optimized clustering of system	
Doval et al. [1999]	Automation of partitioning components of a system into clusters	MDG	String of integers	Standard	Standard	MQ	Optimized clustering of system	Continued work from Mancoridis et al. [1999] by implementing a GA

Name	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Mancoridis et al. [1999]	Automation of partitioning components of a system into clusters	MDG	MDG	N/A	N/A	MQ	Optimized clustering of system	Continued work from Mancoridis et al. [1998]; characterisics of modules taken into account in clustering operations
Mitchell and Mancoridis [2002; 2006; 2008]	Automation of partitioning components of a system into clusters	MDG	String of integers	Standard	Standard	MQ as a sum of clustering factors	Optimized clustering of system	Continued work from Doval et al. [2002]; new definition of the modularization quality and an enhanced HC algorithm
Mitchell and Mancoridis [2003; 2008]	Automation of partitioning components of a system into clusters	MDG	String of integers	Standard	Standard	MQ, search landscape	Optimized clustering of system	Continued work from Mitchell and Mancoridis [2002; 2006; 2008]; search landscape taken into account

Name	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Mitchell et al. [2000]	Automated reverse engineering from source code to architecture	Source code of application	N/A	N/A	N/A	Quality based on use and style relations	Software architecture	HC and edge removal are used as search algorithms from MDG to architecture
Harman et al. [2002]	New encoding and crossover introduced	System as modules and elements	Look-up table for modules	Move component from one module to another	New crossover, preserves partial module allocations	Maximize cohesion, minimize coupling	Optimized clustering	
Harman et al. [2005]	Comparison of robustness between two fitness functions	Clustered system	N/A	N/A	N/A	MQ compared against EVM	-	
Hyunh and Cai [2007]	Conformance check of actual design to suggested design	Design structure matrices for design and source code (DSM)	Graph constructed of DSM	N/A	N/A	Graph edit distance, penalty and differentiation between graphs with same distance	Optimized clustering of actual design conforming to suggested design	
Mahdavi et al. [2003a; 2003b]	Automated clustering of system	MDG	String of integers	Standard	Standard	MQ	Optimized clusteringg of system	Multiple hill climbs are used as search algorithm; building blocks are preserved by using parallel hill climbs

5. SOFTWARE REFACTORING

5.1. Background

Software evolution often results in “corruption” in software design, as quality is overlooked while new features are added, or the old software should be modified in order to ensure the highest possible quality. At the same time resources are limited. Refactoring and in particular the miniaturization of libraries and applications are therefore necessary. Program transformation is useful in a number of applications including program comprehension, reverse engineering and compiler optimization. A transformation algorithm defines a sequence of transformation steps to apply to a given program and it is described as changing one program into another. It involves altering the program syntax while leaving it semantics unchanged. In object-oriented design, one of the biggest challenges when optimizing class structures using random refactorings is to ensure behavior preservation. One has to take special care of the pre- and post-conditions of the refactorings.

There are three problems with treating software refactoring as a search-based problem. First, how to determine which are the useful metrics for a given system. Second, finding how best to combine multiple metrics. Third is that while each run of the search generates a single sequence of refactorings, the user is given no guidance as to which sequence may be best for their given system, beyond their relative fitness values.

In practice, refactoring (object-oriented software) deals mostly with re-organizing methods and attributes and inheritance and delegation structures. The following subsection presents approaches where search-based techniques have been used to automatically achieve refactorings, and the fundamentals of each study are collected in Table 4.

5.2. Approaches

Di Penta et al. [2005] present a software renovation framework (SRF) and a toolkit that covers several aspects of software renovation, such as removing unused objects and code clones, and refactoring existing libraries into smaller ones. Refactoring has been implemented in the SRF using a hybrid approach based on hierarchical clustering, on GAs and hill climbing, also taking into account the developer’s feedback. Most of the SRF activities deal with analyzing dependencies among software artifacts, which can be represented with a dependency graph.

Hierarchical clustering, through the agglomerative-nesting algorithm, and Silhouette statistics [Kaufman and Rousseeuw, 1990] are initially used to determine the optimal number of clusters and the starting population of a software renovation GA. This is followed by a search aimed at minimizing a multi-objective function, which takes into account both the number of inter-library dependencies and the average number of objects linked by each application. As the fitness function is also let to consider the experts' suggestions, the SRF becomes a semi-automatic approach composed of multiple refactoring iterations, which are interleaved by developers' feedback.

Software systems are represented by a system graph SG, which contains the sets of all object modules, all software system libraries, all software system applications and the set of oriented edges representing dependencies between objects. The refactoring framework consists of several steps: 1. software systems applications, libraries and dependencies among them are identified, 2. unused functions and objects are identified, removed or factored out, 3. duplicated or cloned objects are identified and possibly factored out, 4. circular dependencies among libraries are removed, or at least reduced, 5. large libraries are refactored into smaller ones and, if possible, transformed into dynamic libraries, and 6. objects which are used by multiple applications, but which are not yet organized into libraries, are grouped into new libraries. Step five, splitting existing, large libraries into smaller clusters of objects, is now studied more closely.

The refactoring of libraries is done in the SRF in the following steps: 1. determine the optimal number of clusters and an initial solution, 2. determine the new candidate libraries using a GA, 3. ask developers' feedback. The effectiveness of the refactoring process is evaluated by a quality measure of the new library organization, the Partitioning Ratio, which should be minimized.

The genome for the GA is encoded using the bit-matrix encoding. The genome matrix GM for each library to be refactored corresponds to a matrix of k rows and l_x columns, where the entry i_j is 1 if the object j is contained in cluster i. The mutation operator works in two modes. With probability p_{mut}, it takes a random column and randomly swaps two bits, i.e., an object is moved from a library to another. With probability $p_{clone} < p_{mut}$, it takes a random position in the matrix: if it is zero and the libraries are dependent on it, then the mutation operator clones the object into the current library. As clones should be minimized, the GA heuristically activates the cloning only for the final part of the evolution (after 66%). The proposed strategy favors dependency minimization by moving objects between libraries. The developers may also give a Lock Matrix when they strongly believe that an object should belong to a certain cluster. The mutation operator does not perform any action that would bring a genome in an inconsistent state with respect to the Lock Matrix. The crossover is the one point crossover, which exchanges the content of two genome matrices around a random column.

The fitness function *F* should balance four factors: the number of inter-library dependencies, the total number of objects linked to each application, the size of new libraries and the feedback by developers. Thus, *F* consists of the Dependency factor DF, the Partitioning ratio PR, the Standard deviation factor SD and the Feedback factor FF. The FF is stored in a bit-matrix FM, which has the same structure of the genome matrix and which incorporates those changes to the libraries that developers suggested. Each factor of the fitness function is given a separate real, positive weight. DF is given weight 1, as it has maximum influence.

Di Penta et al. report that the presented GA suffers from slow convergence. To improve its performance, it has been hybridized with HC techniques. In their experiment, applying HC only to the last generation significantly improves neither the performance nor the results, but applying HC to the best individuals of each generation makes the GA converge significantly faster. In the case study, the GA reduces dependencies of one library to about 5% of the original amount while keeping the PR almost constant. For two other libraries, a significant reduction of inter-library dependencies is obtained while slightly reducing PR in one and increasing the PR in the other. The addition of HC into GA does not improve fitness function, since GA also converges to similar results, when it is executed on an increased number of generations and increased population size. Noticeably, performing HC on the best individuals of each generation produces a drastic reduction in convergence times.

Fatiregun et al. [2004] use meta-heuristic search algorithms to automate, or partly automate the problem of finding good program transformation sequences. With the proposed method one can dynamically generate transformation sequences for a variety of programs also using a variety of objective functions. The goal is to reduce program size, but the approach is argued to be sufficiently general that it can be used to optimize any source-code level metric. Random search (RS), hill climbing (HC) and GA are used.

An overall transformation of a program *p* to an improved version *p'* typically consists of many smaller transformation tactics. Each tactic consists of the application of a set of rules. A transformation rule is an atomic transformation capable of performing the simple alterations. To achieve an effective overall program transformation tactic many rules may need to be applied and each would have to be applied in the correct order to achieve the desired results.

In HC, an initial sequence is generated randomly to serve as the starting point. The algorithm is restarted several times using a random sequence as the starting individual each time. The aim is to divert the algorithm from any local optimum.

Each transformation sequence is encoded as an individual that has a fixed sequence length of 20 possible transformations. An example individual is a vector of the transformation numbers. In HC, the neighbor is defined as the mutation of a single gene from the original sequence. Crossover is standard one-point crossover. In addition to transformations, cursor moves are also used. The tournament selection is used for selecting mating parents and creating a single offspring, which replaces the worse of the parents. The authors consider optimizing the program with respect to the size of the source-code, i.e., LOC, where the aim is to minimize the number of lines of code as much as possible.

The fitness is measured as the nominal difference in the lines of code between the source program and the new transformed program created by that particular sequence. This is evaluated by a process of five steps: 1. compute length of the input program, 2. generate the transformation sequence, 3. apply the transformation sequence, 4. compute the current length of the program, 5. compute the fitness, which is the difference between steps 1 and 4.

Results show that GA outperforms both RS and HC. In cases where RS outperformed GA and HC, it was noticed that GA and HC are not “moving” towards areas where potential optimizations may be. Analyzing the GA, the authors believe that the GA potentially kills off good subsequences of transformations during crossover.

Seng et al. [2005] describe a methodology that computes a subsystem decomposition that can be used as a basis for maintenance tasks by optimizing metrics and heuristics of good subsystem design. GA is used for automatic decomposition. If a desired architecture is given, e.g., a layered architecture, and there are several violations, this approach attempts to determine another decomposition that complies with the given architecture by moving classes around. Instead of working directly on the source code, it is first transformed into an abstract representation, which is suitable for common object-oriented language.

In the GA, several potential solutions, i.e., subsystem decompositions, form a population. The initial population can be created using different initialization strategies. Before the algorithm starts, the user can customize the fitness function by selecting several metrics or heuristics as well as by changing thresholds. The 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. The approach is based on the Grouping GA [Falkenaur, 1998], which is particularly well suited for finding groups in data. For chromosome encoding, subsystem candidates are associated with genes and the power set of classes is used as the alphabet for gens. Consequently, a gene is associated with a set of classes, i.e., an element of the power set. This representation allows a one-to-one mapping of geno- and phenotype to avoid redundant coding.

An adapted crossover operator and three kinds of mutation are used. The operators are adapted so that they are non-destructive and preserve a complete subsystem candidate as far as possible. The *split&join* mutation either divides one subsystem to two, or vice versa. 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. *Elimination* mutation deletes a subsystem candidate and distributes its classes to other subsystem candidate, based on association weights. *Adoption* mutation tries to find a new subsystem candidate for an orphan, i.e., a subsystem candidate containing only a single class. This operator moves the orphan to the subsystem candidate that has the highest connectivity to the orphan.

Initial population supports the building block theorem. Randomly selected connected components of the dependency graph are taken for half the population and highly fit ones for the rest. The crossover operator forms two children from two parents. After choosing the parents, the operator selects a sequence of subsystem candidates in both parents, and mutually integrates them as new subsystem candidates in the other parent, 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, and their non-duplicated classes are collected and distributed over the remaining subsystem candidates. Fitness function is defined as $f = w_1$* cohesion + w_2* coupling + w_3* complexity + w_4* cycles + w_5* bottlenecks.

For evaluation, a tool prototype has been implemented. Evaluation on the clustering of different software systems has revealed that results on roulette wheel selection are only slightly better than those of tournament selection. The adapted operators allow using a relatively small population size and few generations. Results on Java case study show that the approach works well. Tests on optimizing subsets of the fitness function show that only if all criteria are optimized, the authors are able to achieve a suitable compromise with very good complexity, bottleneck and cyclomatic values and good values for coupling and cohesion.

Seng et al. [2006] have continued their work by developing a search-based approach that suggests a list of refactorings. The approach uses an evolutionary algorithm and simulated refactorings that do not change the system's externally visible behavior. The source code is transformed into a suitable model – the phenotype. The genotype consists of the already executed refactorings. Model elements are differentiated according to the role they play in the systems design before trying to improve the structure. This step is called classification. Not all elements can be treated equally, because the design patterns sometimes deliberately violate existing design heuristics. The approach is restricted to those elements that respect general design guidelines. Elements that deliberately do not respect them are left untouched in order to preserve the developers conscious design decisions.

The initial population is created by copying the model extracted from the source code a selected number of times. Selection for a new generation is made with tournament selection strategy. The optimization stops after a predefined number of evolution steps. The source code model is designed to accommodate several object-oriented languages. The basic model elements are classes, methods, attributes, parameters and local variables. In addition, special elements called access chains are needed. An access chain models the accesses inside a method body, because it is needed to adapt these references during the optimization. If a method is moved, the call sites need to be changed. An access chain therefore consists of a list of accesses. Access chains are hierarchical, because each method argument at a call site is modeled as a separate access chain, that could possible contain further access chains.

The model allows to simulate most of the important refactorings for changing the class structure of a system, which are extract class, inline class, move attribute, push down attribute, pull up attribute, push down method, pull up method, extract superclass and collapse class hierarchy. The genotype consists of an ordered list of executed model refactorings including necessary parameters. The phenotype is created by applying these model refactorings in the order that is given by the genotype to the initial source code model. Therefore the order of the model refactorings is important, since one model refactoring might create the necessary preconditions for some of the following ones.

Mutation extends the current genome by an additional model refactoring; the length of the genome is unlimited. Crossover combines two genomes by selecting the first random *n* model refactorings from parent one and adding the model refactorings of parent two to the genome. The refactorings from parent one are definitely safe, but not all model refactorings of parent two might be applicable. Therefore, the model refactorings are applied to the initial source code model. If a refactoring that cannot be executed is encountered due to unsatisfied preconditions, it is dropped. Seng et al. argue that the advantage of this crossover operator is that it guarantees that the externally visible behavior is not changed, while the drawback is that it takes some time to perform the crossover since the refactorings need to be simulated again.

Fitness is a weighted sum of several metric values and is designed to be maximized. The properties that should be captured are coupling, cohesion, complexity and stability. For coupling and cohesion, the metrics from Briand's [2000] catalogue are used. For complexity, weighted methods per class (WMC) and number of methods (NOM) are used. The formula for stability is adapted from the reconditioning of subsystem structures. Fitness = $\sum(\text{weight}_m * (M(S) - M_{\text{init}}(S))/M_{\text{max}}(S) - M_{\text{init}}(S)$. Before optimizing the structure the model elements are classified according to the roles they play in the systems design, e.g., whether they are a part of a design pattern.

Tests show that after approximately 2000 generations in a case study the fitness value does not significantly change any more. The approach is able to find refactorings that improve the fitness value. In order to judge whether the refactorings make sense, they are manually inspected by the authors, and from their perspective, all proposed refactorings can be justified. As a second goal, the authors modify the original system by selecting 10 random methods and misplacing them. The approach successfully moves back each method at least once.

O'Keeffe and Ó Cinnéide [2004] have developed a prototype software engineering tool capable of improving a design with respect to a conflicting set of goals. A set of metrics is used for evaluating the design quality. As the prioritization of different goals is determined by weights associated with each metric, a method is also described of assigning coherent weights to a set of metrics based on object-oriented design heuristics.

Dearthóir is a prototype design improvement tool that restructures a class hierarchy and moves methods within it in order to minimize method rejection, eliminate code duplication and ensure superclasses are abstract when appropriate. The tool uses the SA technique to find close-to-optimum solutions to the combinatorial optimization problem described here. The refactorings are behavior-preserving transformations in Java code. The refactorings employed are limited to those that have an effect on the positioning of methods within an inheritance hierarchy. In order for the SA search to move freely through the search space every change to the design must be reversible. To ensure this, pairs of refactoring have been chosen that complement each other. The refactoring pairs are: 1. move a method up or down in the class hierarchy, 2. extract (from abstract class) or collapse a subclass, 3. make a class abstract or concrete, and 4. change superclass link of a class.

The following method is intended to filter out heuristics that cannot easily be transformed into valid metrics because they are vague, unsuitable for the programming language in use or dependent on semantics. Firstly, for each heuristic: define the property to be maximized or minimized in the heuristic, determine whether the property can be accurately measured, and note whether the metrics should be maximized or minimized. Secondly, identify the dependencies between the metrics. Thirdly, establish precedence between dependent metrics and a threshold where necessary: prioritize heuristics. Fourthly, check that the graph of precedence between metrics is acyclic. Finally, weights should be assigned to each of the metrics according to the precedences and threshold.

The selected metrics are: 1) minimize rejected methods (RM) (number of inherited but unused methods), 2) minimize unused methods (UM), 3) minimize featureless classes (FC), 4) minimize duplicate methods (DM) (number of methods duplicated within an inheritance hierarchy), 5) maximize abstract superclasses (AS). Metrics should be appreciated so that DM > RM > FC > AS, and UM > FC.

Most of the dependencies in the graph do not require thresholds. However, a duplicate method is avoided by pulling the method up into its superclass, which could result in the method being rejected by any number of classes. Therefore a threshold value is established for this dependency. O’Keeffe and Ó Cinnéide argue that it is more important to avoid code duplication than any amount of method rejection; therefore the threshold can be an arbitrarily high number.

A case study is conducted with a small inheritance hierarchy. The case study shows that the metric values for input and output either become better or stay the same. In the input design several classes contain clumps of methods, where as in the output design methods are spread quite evenly between the various classes. This indicates that responsibilities are being distributed more evenly among the classes, which means that components of the design are more modular and therefore more likely to be reusable. This in turn suggests that adherence to low-level heuristics can lead to gains in terms of higher-level goals. Results indicate that a balance between metrics has been achieved, as several potentially conflicting design goals are accommodated.

O’Keeffe and Ó Cinnéide [2006; 2008a] have continued their research by constructing a tool capable of refactoring object-oriented programs to conform more closely to a given design quality model, by formulating the tasks as a search problem in the space of alternative designs. This tool, CODe-Imp, can be configured to operate using various subsets of its available automated refactorings, various search techniques, and various evaluation functions based on combinations of established metrics.

CODe-Imp uses a two-level representation; the actual program to be refactored is represented as its Abstract Syntax Tree (AST) but a more abstract model called the *JPM* is also maintained from which metric values are determined and refactoring preconditions are checked. The change operator is a transformation of the solution representation that corresponds to a refactoring that can be carried out on the source code.

The CODe-Imp takes Java source code as input and extracts design metric information via a Java Program Model (JPM), calculates quality values according to the fitness function and effects change in the current solution by applying refactorings to the AST as required by a given search technique. Output consists of the refactored input code as well as a design improvement report including quality change and metric information.

The refactoring configuration of the tool is constant throughout the case studies and consists of the following fourteen refactorings. Push down/pull up field, push down/pull up method, extract/collapse hierarchy, increase/decrease field security, replace inheritance with delegation/replace delegation with inheritance, increase/decrease method security, made superclass abstract/concrete. During the search process alternative designs are repeatedly generated by the application of a refactoring to the existing design, evaluated for quality, and either accepted as the new current design or rejected. As the current design changes, the number of points at which each refactoring can be applied will also change. In order to see whether refactorings can be made without changing program behavior, a system of conservative precondition checking is employed.

The used search techniques include first-ascent HC (HC1), steepest-ascent HC (HC2), multiple-restart HC (HCM) and low-temperature SA. For the SA, CODe-Imp employs the standard geometric cooling schedule.

The evaluation functions are flexibility, reusability and understandability of the QMOOD hierarchical design quality model [Bansiya and Davis, 2002]. Each evaluation function in the model is based on a weighted sum of quotients on the 11 metrics forming the QMOOD (design size in class, number or hierarchies, average number of ancestors, number of polymorphic methods, class interface size, number of methods, data access metric, direct class coupling, cohesion among methods of class, measure of aggregation and measure of functional abstraction). Each metric value for the refactored design is divided by the corresponding value for the original design to give the metric change quotient. A positive weight corresponds to a metric that should be increased while a negative weight corresponds to metric that should be decreased.

All techniques demonstrate strengths. HC1 consistently produces quality improvements at a relatively low cost, HC2 produces the greatest mean quality improvements in two of the six cases, HCM produces individual solutions of highest quality in two cases and SA produced the greatest mean quality improvement in one case.

When examining the resulting designs from two inputs A and B, the refactored design for input A was superior in terms of general object-oriented design principles such as the maximization of encapsulation and the use of inheritance only where it is suitable, so there was some evidence that general maintainability had increased. There is no conclusive evidence that the refactored design would be more flexible in particular. The refactored design for input B is not only better in terms of general object-oriented principles, but also can be regarded as more flexible than the input design.

Inspection of output code and analysis of solution metrics provide some evidence in favor of use of the flexibility metric and even stronger evidence for using the understandability function. The reusability in present form is not found suitable or maintenance because it resulted in solutions including a large number of featureless classes. The authors conclude that both local search and simulated annealing are effective in the context of search-based software refactoring.

O'Keeffe and Ó Cinnéide [2007; 2008b] have continued their work by implementing also a GA and a multiple ascent HC (MAHC) to the CODe-Imp refactoring tool and further testing the existing search techniques. The encoding, crossover and mutation for the GA are similar to those presented by Seng et al. [2006], and the power of tool has been increased by adding a number of different refactorings available for use in searching for a superior design.

The fitness function is an implementation of the understandability function from Bansiya and Davis's [2002] QMOOD hierarchical design quality model consisting of a weighted sum of metric quotients between two designs. This design quality evaluation function was previously found by the authors to result in tangible improvements to object-oriented program design in the context of search-based refactoring.

Results for the SA support the recommendation of low values for the cooling factor, since more computationally expensive parameters do not yield greater quality function gains.

In summary, SA has several disadvantages: it is hard to recommend a cooling schedule that will generally be effective, results vary considerably across input programs and the search is quite slow. No significant advantage in terms of quality gain was observed that would make up for these shortcomings. The GA has the advantage that it is easy to establish a set of parameters that work well in the general case, but the disadvantages are that it is costly to run and varies greatly for different input programs. Again, no significant advantage in terms of quality gain was observed that would make up for these shortcomings. Multiple-ascent HC stood out as the most efficient search technique in this study: it produced high-quality results across all the input programs, is relatively easy to recommend parameter for and runs more quickly than any of the other techniques examined. Steepest ascent HC produced surprisingly high quality solutions, suggesting that the search space is less complex than might be expected, but is slow when considered its known inability to escape local optima. Results show MAHC to outperform both SA and GA over a set of four input programs.

Harman and Tratt [2007] show how Pareto optimality can improve search based refactoring, making the combination of metrics easier and aiding the presentation of multiple sequences of optimal refactorings to users. Intuitively, each value on a Pareto front maximizes the multiple metrics used to determine the refactorings. Through results obtained from three case studies on large real-world systems, it is shown how Pareto optimality allows users to pick from different optimal sequences of refactorings, according to their preferences. Moreover, Pareto optimality applies equally to sub-sequences of refactorings, allowing users to pick refactoring sequences based on the resources available to implement those refactorings. Pareto optimality can also be used to compare different fitness functions, and to combine results from different fitness functions.

Harman and Tratt use the move method refactoring presented by Seng et al. [2006]. Three systems are used in the case study, all non-trivial real-world systems. The search algorithm itself is a non-deterministic non-exhaustive hill climbing approach. A random move method refactoring is chosen and applied to the system. The fitness value of the updated system is then calculated. If the new fitness value is worse than the previous value, the refactoring is discarded and another one is tried. If the new fitness value is better than the previous, the refactoring is added to the current sequence of refactorings, and applied to the current system to form the base for the next iteration. A cut-off point is set for checking neighbors before concluding that a local maximum is reached. The end result of the search is a sequence of refactorings and a list of the before and after values of the various metrics involved in the search.

Two metrics are used to measure the quality: coupling and standard deviation of methods per class (SDMPC). Coupling is from Briand's [2000] catalogue. The second metric, SDMPC, is used to act as a 'counter metric' for coupling. An arbitrary combination of the metrics is used, the fitness function being SDMPC*CBO. The new fitness function improves the CBO value of the refactored system while also improving the SDMPC of the system. All the points on a Pareto front are, in isolation, considered equivalently good. In such cases, it might be that the user may prefer some of the Pareto optimal points over others.

The concept of a Pareto front is argued to make as much sense with subsets of data as it does for complete sets. Harman and Tratt also stress the importance of knowing how many runs a search-based refactoring system will need to achieve a reasonable Pareto front approximation. Furthermore, developers are free to execute extra runs of the system if they feel they have not yet achieved points of sufficient quality on the front approximation. Pareto optimality allows to determine whether one fitness function is subsumed by another: broadly speaking, if fitness function *f* produces data which, when merged with the data produced from function *f'*, contributes no points to the Pareto front then we that that *f* is subsumed by *f'*. Although it may not be immediately apparent, Pareto optimally confers a benefit potentially more useful than simply determining whether one fitness function is subsumed by another. If two fitness functions generate different Pareto optimal points, then they can naturally be combined to single front. Pareto optimality is shown to have many benefits for search-based refactoring, as it lessens the need for “perfect” fitness functions.

Table 4. Research approaches in search-based software refactoring

Name	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Di Penta et al. [2005]	A refactoring framework taking into account several aspects of software quality when refactoring existing system.	Software system as a system graph SG	Bit matrix; each library of clusters is represented by a matrix	Swapping two bits in a column or changing a value from 0 to 1 (taking into account preconditions)	N/A	Dependency factor, partitioning ratio, standard deviation and feedback	Refactored libraries	HC and GA used.
Fatiregun et al. [2004]	Program refactoring on source code level	Source code	Integer vector containing transformation numbers	Standard	Standard one-point	Size of source code (LOC)	A sequence of program transformations	Random search, HC and GA are used
Seng et al. [2005]	Optimizing subsystem decomposition for maintenance	Model of system as a graph, extracted from source code	Genes represent subsytem candidates	Split&join, elimination and adoption	Two children from two parents, integrating crossover	Cohesion, coupling, complexity, bottlenecks and cycles	Source code extracted from resulting model	
Seng et al. [2006]	Refactoring a software system with a wide set of operations	Model of system, extracted from source code, with access chains	Ordered list of refactorings	Common class structrure refactorings, the list is extended with a suggested transformation	Minimize rejected, duplicated and unused methods and featureless classes and maximize abstract classes	Refactored software system	SA used as search algorithm, introducing a heuristic for weighting conflicting quality goals	

Name	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
O'Keeffe and Ó Cinnéide [2004]	Automating software refactoring	Software system	N/A	Restructure class hiearchy and method moves, mutations in counter-pairs in order to reverse a move	N/A	Minimize rejected, duplicated and unused methods and featureless classes and maximize abstract classes	Refactored software system	SA used as search algorithm, introducing a heuristic for weighting conflicting quality goals
O'Keeffe and Ó Cinnéide [2006; 2008a]	Automating software refactoring	System as Java source code	N/A	Refactorings regarding visibility, class hierarchy and method placement	N/A	Reusability, flexibility and understandability	Refactored code and design improvement report	Three variations of hill climbing and SA used as search algorithms
O'Keeffe and Ó Cinnéide [2007; 2008b]	Comparison between different search techniques	System as Java source code	Ordered list of refactorings [Seng et al., 2006]	Common class structrure refactorings, the list is extended with a suggested transformation [Seng et al., 2006]	A random set of transformations from one parent chosen, the transformations of the other added to that list [Seng et al., 2006]	Understandability	Refactored code and design improvement report	GA and multiple ascent hill climb implemented
Harman and Tratt [2007]	Pareto optimality used for multi-objective optimization	Software system	N/A	Move method	N/A	Coupling and standard deviation of methods per class	A sequence of refactorings	HC used as search algorithm

6. SOFTWARE QUALITY

6.1 Importance of design quality

Software quality assessment has become an increasingly important field. The complexity caused by object-oriented methods makes the task more important and more difficult. An ideal quality predictive model can be seen as the mixture of two types of knowledge: common knowledge of the domain and context specific knowledge. In existing models, one of the two types is often missing. During its operating time, a software system undergoes various changes triggered by error detection, evolution in the requirements or environment changes. As a result, the behavior of the software gradually deteriorates as modifications increase. This quality slump may go as far as the entire software becoming unpredictable.

Software quality is a special concern when automatically designing software systems, as the quality needs to be measured with metrics and in pure numerical values. The use of metrics may even be argued, as they cannot possible contain all the knowledge that an experienced human designer has. Sahraoui et al. [2000] have investigated whether some object-oriented metrics can be used as an indicator for automatically detecting situations where a particular transformation can be applied to improve the quality of a system. The detection process is based on analyzing the impact of various transformations on these object-oriented metrics using quality estimation models.

Sahraoui et al. have constructed a tool which, based on estimations on a given design, suggests particular transformations that can be automatically applied in order to improve the quality as estimated by the metrics. Roughly speaking, building a quality estimation model consists of establishing a relation of cause and effect between two types of software characteristics. Firstly, internal attributes which are directly measureable, such as size, inheritance and coupling, and secondly, quality characteristics which are measurable after a certain time of use such as maintainability, reliability and reusability. To study the impact of the global transformations on the metrics, first the impact of each elementary transformation is studied and then the global impact is derived. A case study is used for the particular case of the diagnosis of bad maintainability by using the values of metrics for coupling and inheritance as symptoms. Based on the results of this study, Sahraoui et al. argue that using metrics is a step toward the automation of quality improvement, but that experiments also show that a prescription cannot be executed without a validation of a designer/programmer.

The use of evolution metrics for fitness functions has especially been studied [Mens and Demeyer 2001; Harman and Clarke, 2004]. If one looks at the whole process of detecting flaws and correcting them, metrics can help automating a large part of it. However, the results of the experiments show that a prescription cannot be executed without a validation of a designer or programmer. This approach cannot capture all the context of an application to allow full automation.

Some approaches regarding software quality have also been made with search-based techniques. Bouktif et al. [2002; 2004] aim at predicting software quality of object-oriented systems with GAs, and Vivanco and Jin [2007] have implemented a GA to identify possible problematic software components. Bouktif et al. [2006] have also implemented a SA to combine different quality prediction models. The fundamentals of each approach is collected in Table 5.

6.2 Search-based approaches

Bouktif et al. [2002; 2004] study the prediction of stability at object-oriented class level and propose two GA based approaches to solve the problem of quality predictive models: the first approach combines two rule sets and the second one adapts an existing rule set. The predictive model will take the form of a function that receives as input a set of structural metrics and an estimation of the stress and produces as output a binary estimation of the stability. Here, the stress represents the estimated percentage of added methods in a class between two consecutive versions.

The model encoding for the GA that combines rule sets is based on a decision tree. The decision tree is a complete binary tree where each inner node represents a yes-or-no question, each edge is labeled by one of the answers, and terminal nodes contain one of the classification labels from a predetermined set. The decision making process starts at the root of the tree. When the questions at the inner nodes are of form *"Is $x > a$?"*, the decision regions of the tree can be represented as a set of isothetic boxes in an n-dimensional space (n = number of metrics). For the GA representation, these boxes are enumerated in a vector. Each gene is a (box, label) pair, and a vector of these pairs is the chromosome.

Mutation is a random change in the genes that happens with a small probability. In this problem, the mutation operator randomly changes the label of a box. To obtain an offspring, a random subset of boxes from one parent is selected and added to the set of boxes of the second parent. The size of the random subset is v times the number of boxes of the parent where v is a parameter of the algorithm. By keeping all the boxes of one of the parents, completeness of the offspring is automatically ensured. To guarantee consistency, the added boxes are made predominant (the added boxes are “laid over” the original boxes). A level of predominance is added as an extra element to the genes. Each gene is now a three-tuple (box, label, level). The boxes of the initial population have level 1. Each time a predominant box is added to a chromosome, its level is set to 1 plus the maximum level in the hosting chromosome. To find the label of an input vector *x* (a software element), first all the boxes containing *x* are found, and *x* is assigned the label of the box that have the highest level of predominance. To measure the fitness a correctness function is used; the function calculates the number of cases that the rule correctly classifies divided by the total number of cases that the rule classifies. The correctness function is defined as C = 1 - training error. By using the training error for measuring the fitness, it is found that the GA tended to “neglect” unstable classes. To give more weight to data points with minority labels, Youden’s [1961] J-index is used. Intuitively, the *J*-label is the average correctness per label. If one has the same number of points for each label, then $J = C$.

With a GA for adapting a rule set, an existing rule set is used as the initial population of chromosomes, each rule of the rule set being a chromosome and each condition in the rule as well as the classification label being a gene. Each chromosome is attributed a fitness value, which is $C*t$, where t is the fraction of cases that the rule classifies in the training set. The weight t allows giving rules that cover a large set of training cases a higher chance of being selected.

Parents for crossover are selected with roulette wheel method. A random cut point is generated for each parent, i.e., the cut-points are different for each parent. Otherwise, the operation is a traditional one-point crossover. By allowing chromosome within a pair to be cut at different places, a wider variety is allowed with respect to the length of the chromosomes. The chromosomes are then mutated with a certain probability. The mutation of a gene consists of changing the value to which the attribute encoded in the gene is compared with to a value chosen randomly from a predefined set of values for the attribute (or class label, in case the last gene is mutated). The new chromosomes are scanned and trimmed to get rid of redundancy in the conditions that form the rules that they encode. Inconsistent rules are attributed a fitness value of 0 and will eventually die. A fixed population size is maintained. Elitism is performed when the population size is odd. This consists of copying one or more of the best chromosomes from one generation to the next. Before passing from one generation to another, the performance of combined rules to one rule set is evaluated.

In the experimental setting, to build experts (that simulate existing models), stress and 18 metrics (belonging to coupling, cohesion, complexity and inheritance) are used. Eleven object-oriented systems are used to “create” 40 experts. For the combining GA, the elitist strategy is used, where the entire population apart from a small number of fittest chromosomes is replaced. The test results show that the approach of combining experts can yield significantly better results than using individual models. The adaptation approach does not perform as well as the combination, although it gave a slight improvement over the initial model in one case. The authors believe that using more numerous and real experts on cleaner and less ambiguous data, the improvement will be more significant.

Bouktif et al. [2006] have continued their research by applying simulated annealing to combine experts. Their approach attempts to reuse and adapt quality predictive models, each of which is viewed as a set of expertise parts. The search then aims to find the best subset of expertise parts, which forms a model with an optimal predictive accuracy. The SA algorithm and a GA made for comparison were defined for Bayesian classifiers (BCs), i.e., probabilistic predictive models.

An optimal model is built of a set of experts, each of which is given a weight. Each individual, i.e., chunk, of expertise is presented by a tuple consisting of an interval and a set of conditional probabilities. Transitions in the neighborhood are made by changing probabilities or interval boundaries. A transition may also be made by adding or deleting a chunk of expertise. The fitness function is the correctness function.

For evaluation, the SA needs two elements as inputs: a set of existing experts and a representative sample of context data. Results show a considerable improvement in the predictive accuracy, and the results produced by the SA are stable. The values for GA and SA are so similar that the authors do not see a need to value one approach over the other. Results also show that the accuracy of the best produced expert increases ast eh number of reused models increases and that good chunks of expertise can be hidden in inaccurate models.

Vivanco and Jin [2007] present initial results of using a parallel GA as a feature selection method to enhance a predictive model’s ability to identify cognitively complex components in a Java application. Linear discriminant analysis (LDA) can be used as a multivariate predictive model.

It is theorized that the structural properties of modules have an impact on the cognitive complexity of the system, and further on, that modules that exhibit high cognitive complexity result in poor quality components. A preliminary study is carried out with a biomedical application developed in Java. Experienced program developers are asked to evaluate the system. Classes labeled as low are considered easy to understand and use, while a high ranking implied the class is difficult to fully comprehend and would likely take considerable much more effort to maintain. Source code measurements, 63 metrics for each Java class, are computed using a commercial source code inspection application. To establish a baseline, all the available metrics are used with the predictive model. The Chidamber and Kemerer [1994] metrics suite is used to determine if the model would improve. Finally, the GA is used to find alternate metrics subsets. Using the available metrics with LDA, less than half of the Java classes are properly classified as difficult to understand. The CK metrics suite performs slightly better. Using GA, the LDA predictive model has the highest performance using a subset of 32 metrics. The GA metrics correctly classify close to 100% of the low, nearly half of the medium and two thirds of the high complexity classes.

Vivanco and Jin are most interested in finding the potentially problematic classes with high cognitive complexity. A two-stage approach is evaluated. First, the low complexity classes are classified against the medium/high complexity classes. The GA driven LDA highly accurately identifies the low and medium/high complexity classes with a subset of 24 metrics. When only the medium complexity classes are compared to high complexity, a GA subset of 28 metrics results in extremely high accuracy for the medium complexity classes and in identifying the problematic classes. In all GA subsets, metrics that cover Halstead complexity, coupling, cohesion, and size are used, as well as program readability metrics such as comment to code ratios and the average length of method names.

Table 5. Studies in search-based software quality enhancement

Name	Approach	Input	Encoding	Mutation	Crossover	Fitness	Outcome	Comments
Bouktif et al. [2002;2004]	Combining two rule sets vs. adapting a rule set with GA in quality prediction models	Decision tree	Combination: box, label -pairs from decision tree Adaptation: one rule is one chromosome, each condition in the rule is a gene	Combination: change of label Adaptation: change value of attribute encoding	Combination: a random set of boxes from one parent added to the other and level of predominance added to gene (box, label, level) Adaptation: standard one-point, parents selected with roulette-wheel method	Correctness	Optimal rule set	
Bouktif et al. [2006]	Combining software quality prediction models, i.e., experts	Set of example models and context data	Range and conditional probabilities	Modify range or probability or add or remove an expert	N/A	Correctness	Optimal model combined of sub-optimal models	SA used
Vivanco and Jin [2007]	Identification of complex components	Software system	N/A	N/A	N/A	OO metrics	Classes divided according to complexity levels	

7. FUTURE WORK

From search-based approaches presented here, software clustering and software refactoring (i.e., re-design) appear to be at the most advanced stage. Thus, most work is needed with actual architecture design, starting from requirements and not a ready-made system. Also, search-based application of e.g., design patterns, should be investigated more. Another branch of research should be focused on quality metrics. So far the quality of a software design has mostly been measured with cohesion and coupling, which mostly conform to the quality factors of efficiency and modifiability. However, there are many more quality factors, and if an overall stable software system is desired, more factors should be taken into account in evaluation, such as reliability and stability. Also, as demonstrated with the MQ metric in Section 4, metrics that have seemed good in the beginning may prove to be inadequate when investigated further. Fortunately, it seems that most of the work presented here is the result of developing research that is still continuing. The following research questions should and could very well be answered in the foreseeable future:

- What kind of architectural decisions are feasible to do with search-based techniques?

Research with search-based software architecture design is at an early stage, and not all possible architecture styles and design patterns have been tested. Some architectural decisions are more challenging to implement automatically than others, and in some cases it may not be possible at all. The possibilities should be mapped to effectively research the extent of search-based designs capabilities.

- What is a sufficient starting point to being software architecture design with search-based technique?

So far requirements with a limited set of parameters have been used to build software architecture, or a ready system has been improved. Some design choices need very detailed information regarding the system in order to effectively evaluate the change in quality after implementing a certain design pattern or architecture style. The question of what information is needed for correct quality evaluation is not by any means easily answered.

- What would be optimal representation, crossover and mutation operators regarding the software modularization problem?

Much work has been done with software modularization, and the chromosome encoding, crossover and mutation operators vary greatly. Optimal solutions would be interesting to find.

- What would be optimal representation, crossover and mutation operators regarding the software refactoring problem?

Much research has been done with software refactoring, and the chromosome encoding, crossover and mutation operators vary greatly. Especially the set of mutations is interesting, as they define how greatly the software can be refactored. An optimal encoding might enable a larger set of mutations, thus giving the search-based algorithm a larger space to search for optimal solutions.

- What metrics could be seen as a “standard” for evaluating software quality?

The evaluation of quality, i.e., the fitness function, is a crucial part of evolutionary approaches to software engineering. Some metrics, e.g., coupling and cohesion, have been widely used to measure quality improvements at different levels of design. However, these metrics only evaluate a small portion of quality factors, and there are several versions of even some very “standard” metrics.

- How can metrics be grouped to achieve more comprehendible quality measures?

Metrics achieve clear values, but if a human designer would attempt to use a tool in the design process, notions such as “efficiency” and “modifiability” are more comprehendible than “coupling” and “cohesion”. Thus, being able to group sets of metrics to correspond to certain real-world quality values would be beneficial when making design tools available for common use.

8. CONCLUSIONS

This survey has presented on-going research in the field of search-based software design. There has been much progress in the sub-fields of software modularization and refactoring, and very promising results have been achieved. A more complex problem is automatically designing software architecture from requirements, but some initial steps have already been taken in this direction as well. The surveyed research shows that metrics, such as cohesion and coupling can accurately evaluate some quality factors, as the achieved, automatically improved designs, have been accepted by human designers. However, many authors also report problems: the quality of results is not as high as wished or expected, and many times the blame is placed with a less than optimal encoding and crossover operators. Extensive testing of different encoding options is practically infeasible, and thus inspiration could be found in those solutions that have produced the most promising results. As a whole, software (re-)design seems to be an appropriate field for the application of meta-heuristic search algorithms, and there is much room for further research.

ACKNOWLEDGMENTS

REFERENCES

AHUJA, S.P. 2000. A genetic algorithm perspective to distributed systems design. In: *Proceedings of the Southeastcon 2000*, 2000, 83 – 90.

AMOUI,, M., MIRARAB, S., ANSARI, S. AND LUCAS, C. 2006. A genetic algorithm approach to design evolution using design pattern transformation, *International Journal of Information Technology and Intelligent Computing* **1** (1, 2), June/ August, 2006, 235 – 245.

ANTONIOL, G., DI PENTA, M. AND NETELER, M. 2003. Moving to smaller libraries via clustering and genetic algorithms. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'03),* 2003, 307 – 316.

BASS, L., CLEMENTS, P., AND KAZMAN, R. 1998. *Software Architecture in Practice*, Addison-Wesley, 1998.

BODHUIN, T., DI PENTA, M., AND TROIANO, L. 2007.A search-based approach for dynamically re-packaging downloadable applications, In: *Proceedings of the Conference of the Center for Aadvanced Studies on Collaborative Research (CASCON'07)*, 2007, 27 – 41.

BOUKTIF, S., KÉGL, B. AND SAHRAOUI, H.2002. Combining software quality predictive models: an evolutionary approach. In: *Proceedings of the International Conference on Software Maintenantce (ICSM'02)* 2002.

BOUKTIF, S., AZAR, D., SAHRAOUI, H., KÉGL, B. AND PRECUP, D. 2004. Improving rule set based software quality prediction: a genetic algorithm-based approach, *Journal of Object Technology*, **3**(4), April 2004, 227 – 241.

BOUKTIF, S.. SAHRAOUI, H. AND ANTONIOL, G. 2006. Simulated annealing for improving software quality prediction, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006),* 1893 – 1900.

BOWMAN, M., BRIAND, L.C., AND LABICHE, Y. 2008. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms, Technical report SCE-07-02, Carleton University.

BRIAND, L., WÜST, J., DALY, J., PORTER, V. 2000. Exploring the relationships between design measures and software quality in object oriented systems. *Journal of Systems and Software,* 51, 2000, 245 – 273.

BUI , T.N., AND MOON, B.R. 1996. Genetic algorithm and graph partitioning, *IEEE Transactions on Computers*, **45**(7), July 1996, 841 – 855.

CANFORA, G., DI PENTA, M., ESPOSITO, R., AND VILLANI, M.L. 2005a. An approach for qoS-aware service composition based on genetic algorithms, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2005,* June 2005, 1069–1075.

CANFORA, G., DI PENTA, M., ESPOSITO, R., AND VILLANI, M.L. 2005b. QoS-aware replanning of composite web services, In: *Proceedings of IEEE International Conference on Web Services (ICWS'05) 2005*, 2005, 121–129.

CANFORA, G., DI PENTA, M., ESPOSITO, R., AND VILLANI, M.L. 2004. A lightweight approach for QoS-aware service composition. In: *Proceedings of the ICSOC* 2004 – short papers. IBM Technical Report, New York, USA.

CAO, L., LI , M. AND CAO, J. 2005a. Cost-driven web service selection using genetic algorithm, In: *LNCS* **3828**, 2005, 906 – 915.

CAO, L., CAO, J., AND LI, M. 2005b. Genetic algorithm utilized in cost-reduction driven web service selection, In: *LNCS **3802***, 2005, 679 – 686.

CHE, Y., WANG, Z., AND LI, X. 2003. Optimization parameter selection by means of limited execution and genetic algorithms, In: *APPT 2003, LNCS* **2834**, 2003, 226–235.

CHIDAMBER, S.R., AND KEMERER, C.F. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* **20** (6), 1994, 476 – 492.

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. 2003.Reformulating software engineering as a search problem, *IEE Proceedings - Software,* **150** (3), 2003, 161 – 175.

DEB, K. 1999.Evolutionary algorithms for multicriterion optimization in engineering design, In: *Proc. Evolutionary Algorithms in Engineering and Computer Science* (EUROGEN'99) 135 – 161.

DI PENTA, M., NETELER, M., ANTONIOL, G. AND MERLO, E. 2005. A language-independent software renovation framework, *The Journal of Systems and Software* **77**, 2005, 225 – 240.

DOVAL, D., MANCORIDIS, S., AND MITCHELL, B.S., 1999. Automatic clustering of software systems using a genetic algorithm, In: *Proceedings of the Software Technology and Engineering Practice*, 1999, 73 – 82.

FALKENAUR, E. 1998. *Genetic Algorithms and grouping problems*, Wiley, 1998.

FATIREGUN, D., HARMAN, M. AND HIERONS, R. 2004. Evolving transformation sequences using genetic algorithms. In *Proceedings of the 4th International Workshop on Source Code Analysis and Manipulation (SCAM 04),* Sept. 2004, IEEE Computer Society Press, 65 – 74.

FONSECA C., AND FLEMING, P. 1995.An overview of evolutionary algorithms in multiobjective optimization, Evolutionary Computation **3**(1): 1-16 (1995).

GAMMA, E., HELM, R., JOHNSON, R. AND VLISSIDES, J. 1995. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

GARFINKEL, R., AND NEMHAUSER, G.L. 1972. *Integer Programming,* John Wiley and Sons, 1972.

GOLD, N., HARMAN, M., LI AND, Z., AND MAHDAVI, K. 2006. A search based approach to overlapping concept boundaries. In Proceedings *of the 22nd International Conference on Software Maintenance (ICSM 06),* USA Sept. 2006, 310 – 319.

GOLDSBY, H. AND CHANG, B.H.C. 2008. Avida-mde: a digital evolution approach to generating models of adaptive software behavior. In *Proceedings of the Genetic Evolutionary Computation Conference (GECCO 2008),* 2008, 1751 – 1758.

GOLDSBY, H., CHANG, B.H.C., MCKINLEY, P.K.,, KNOESTER, D., AND OFRIA, C.A. 2008. Digital evolution of behavioral models for autonomic systems. In *Proceedings of 2008 International Conference on Autonomic Computing,* 2008, 87 – 96.

HARMAN, M. 2007.The current state and future of search based software engineering, In: *Proceedings of the 2007 Future of Software engineering (FOSE'07),* 342 – 357.

HARMAN, M., HIERONS, R. AND PROCTOR, M. 2002. A new representation and crossover operator for search-based optimization of software modularization. In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002),* July 2002, 1351–1358.

HARMAN, M., SWIFT , S. AND MAHDAVI, K. 2005. An empirical study of the robustness of two module clustering fitness functions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005),* USA, June 2005. ACM Press,1029 – 1036.

HARMAN, M. AND JONES, B.F. 2001. Search based software engineering. *Information and Software Technology* 2001, **43**(14), 833 – 9.

HARMAN, M., AND CLARK, J. 2004. Metrics are fitness functions too. In *10th International Software Metrics Symposium (METRICS 2004),* USA Sept. 2004, IEEE Computer Society Press. 58 – 69.

HARMAN, M. AND TRATT, L. 2007. Pareto optimal search based refactoring at the design level, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07),* 2007, 1106 – 1113.

HARMAN, M. AND WEGENER, J. 2004. Getting results with search-based approaches to software engineering. In *Proceedings of the ICSE 2004,* 728 – 729.

HOLLAND, J.H. 1975. *Adaption in Natural and Artificial Systems*. MIT Press, Ann Arbor, 1975.

HUHNS, M., AND SINGH, M. 2005. Service-oriented computing: Key concepts and principals. *IEEE Internet Computing, Jan-Feb 2005,* 75 – 81.

HUYNH, S. AND CAI, Y. 2000. An Evolutionary approach to software modularity analysis, In: *Proceedings of the First international workshop on Assessment of Contemporary Modularization Techniques ACoM'07, ICSE Workshops,* May 2007, IEEE Computer Society, 1– 6.

IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Standard 1471– 2000, 2000.

JAEGER, M.C. AND MÜHL, G. 2007. QoS-based selection of services: the implementation of a genetic algorithm, In: T. Braun, G. Carle and B. Stiller (Eds.): *Kommunikation in Verteilten Systemen (KiVS) 2007 Workshop: Service-Oriented Architectures und Service-Oriented* Computing, VDE Verlag, March 2007, 359 – 371.

KAUFMAN, L. AND ROUSSEEUW, P. 1990. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, 1990.

KENNEDY, J. AND EBERHART, R.C. 1995. Particle swarm optimization, In: *Proceedings of the IEEE International Conference on Neural Networks,* 1995, 1942 – 1948.

KESSENTINI, M., SAHRAOUI, H. AND BOUKADOUM, M. 2008. Model transformation as an optimization problem, In: *Proceedings of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS'08)*, 2008, 159 – 173.

LOSAVIO, F., CHIRINOS, L., MATTEO, A., LÉVY, N., AND RAMDANE-CHERIF, A. 2004. ISO quality standards for measuring architectures. *The Journal of Systems and Software* **72**, 2004, 209 – 223.

LUTZ, R. 2001. Evolving good hierarchical decompositions of complex systems, *Journal of Systems Architecture,* 47, 2001, 613 – 634

MAHDAVI, K., HARMAN, M., AND HIERONS, R. 2003a. A multiple hill climbing approach to software module clustering, In*: Proceedings of ICSM 2003,* 315 – 324.

MAHDAVI, K., HARMAN, M. AND HIERONS, R. 2003b. Finding building blocks for software clustering In: LNCS **2724,** 2003, 2513 – 2514.

MANCORIDIS, S., MITCHELL, B.S., RORRES, C., CHEN, Y.-F. AND GANSNER, E.R. 1998. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the International Workshop on Program Comprehension (IWPC'98),* USA, 1998 IEEE Computer Society Press, 45 – 53.

MANCORIDIS, S. MITCHELL, B.S., CHEN, Y.-F., AND GANSNER, E.R. 1999. Bunch: A clustering tool for the recovery and maintenance of software system structures. In: *Proceedings of the IEEE International Conference on Software Maintenance,* 1999, IEEE Computer Society Press, 50 – 59.

MARTIN, R.C. 2000. Design Principles and Design Patterns, available at http://www.objectmetor.com.

MCMINN, P. 2004. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability 14(2),* 105 – 56.

MENS, T., AND DEMEYER, S., 2001. Future trends in evolution metrics, In: *Proceeding of the International. Workshop on Principles of Software Evolution*, 2001, 83 – 86.

MICHALEWICZ, Z. 1992. *Genetic Algorithms + Data Structures = Evolutionary Programs*. Springer-Verlag, 1992.

MITCHELL, B. 2002. A Heuristic Search Approach to Solving the Software Clustering Problem. Ph. D. Thesis, Drexel University, Philadelphia, January 2002.

MITCHELL, B.S., AND MANCORIDIS, S. 2002. Using heuristic search techniques to extract design abstractions from source code. In: *Proceedings of the Genetic and Evolutionary Computation Conference(GECCO 2002).*, USA, July 2002, 1375 – 1382.

MITCHELL, B.S., AND MANCORIDIS, S. 2003. Modeling the search landscape of metaheuristic software clustering algorithms, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003)*, 2003, 2499 – 2510.

MITCHELL, B.S. AND MANCORIDIS, S. 2006. On the automatic modularization of software systems using the Bunch tool, *IEEE Transactions on Software Engineering*, **32** (3), March 2006, 193 – 208.

MITCHELL, B.S., AND MANCORIDIS, S. 2008. On the evaluation of the Bunch search-based software modularization algorithm, *Soft Computing* **12**(1), 2008, 77 – 93.

MITCHELL, B.S., MANCORIDIS, S., AND TRAVERSO, M. 2000. Search based reverse engineering, In. *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02)* 2002, 431 – 438.

MITCHELL, M. 1996. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

O'KEEFFE , M., AND Ó CINNÉIDE, M. 2004. Towards automated design improvements through combinatorial optimization, In: Workshop *on Directions in Software Engineering Environments (WoDiSEE2004), W2S Workshop -26th International Conference on Software Engineering*, 2004, 75 – 82.

O'KEEFFE, M., AND Ó CINNÉIDE, M. 2006. Search-based software maintenance, In: *Proceedings of CSMR* 2006, 249 – 260.

O'KEEFFE, M., AND Ó CINNÉIDE, M. 2007. Getting the most from search-based refactoring In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07)*, 2007, 1114 – 1120.

O' KEEFFE , M., AND Ó CINNÉIDE, M. 2008a. Search-based refactoring for software maintenance, *Journal of Systems and Software*, **81** (4), April 2008, 502 – 516 .

O' KEEFFE , M., AND Ó CINNÉIDE, M. 2008b. Search-based refactoring: an empirical study, *Journal of Software Maintenance and Evolution: Research and Practice*, **20**, August 2008, 345 – 364 .

RÄIHÄ, O. 2008. Genetic Synthesis of Software Architecture, University of Tampere, Department of Computer Sciences, Lic. Phil. Thesis, September 2008.

RÄIHÄ, O., KOSKIMIES, K., AND MÄKINEN, E. 2008a.Genetic Synthesis of Software Architecture, In: *Proceedings of The 7th International Conference on Simulated Evolution and Learning (SEAL'08)*, December 2008, Melbourne, Australia. *LNCS* **5361**,565 – 574

RÄIHÄ, O., KOSKIMIES, K., MÄKINEN, E., AND SYSTÄ, T. 2008b. 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, August 2008, 129 – 144, to appear in the *Nordic Journal of Computing*.

REEVES, C.R. (ed.). 1995. *Modern Heuristic Techniques for Combinatorial Problems*. McGraw-Hill, 1995.

SAHRAOUI, H.A., GODIN, R., AND MICELI, T. 2000.Can metrics help bridging the gap between the improvement of OO design quality and its automation? In*: Proceedings of the International Conference on Software Maintenance (ICSM '00)*, 154 – 162.

SALOMON, R. 1998. Short notes on the schema theorem and the building block hypothesis in genetic algorithms, In:*Evolutionary Progrsamming VII*, *LNCS* **1447**, 1998, 113 – 122.

SENG, O., BAUYER, M., BIEHL, M., AND PACHE, G. 2005. Search-based improvement of subsystem decomposition, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'05),* 2005, 1045 – 1051.

SENG, O., STAMMEL, J., AND BURKHART, D. 2006. Search-based determination of refactorings for improving the class structure of object-oriented systems, In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'06),* 2006, 1909–1916.

SHAZELY, S., BARAKA, H., AND ABDEL-WAHAB, A. 1998. Solving graph partitioning problem using genetic algorithms. In Midwest *Sympoium on Circuis and Systems*, 1998, 302 – 305.

SHANNON, C.E. 1948. The mathematical theory of communications. *Bell System Technical Journal 27 (379 – 423)*, 623 – 656.

SHAW, M., AND GARLAN, D.1996. *Software Architecture - Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

SIMONS, C. L., AND PARMEE, I.C. 2007a. Single and multi-objective genetic operators in object-oriented conceptual software design. In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07),* 2007 1957 – 1958.

SIMONS, C.L., AND PARMEE, I.C. 2007b. A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design, *Engineering Optimization* **39** (5) 2007, 631 – 648.

SU, S., ZHANG, C., AND CHEN, J. 2007. An improved genetic algorithm for web services selection, In: *LNCS* **4531**, 2007, 284 – 295.

TUCKER, A., SWIFT, S., AND X., LIU. 2001. Grouping multivariate time series via correlation. *IEEE Transactions on Systems, Man, and Cybernetics. Part B: Cybernetics,* **31**(2), 2001, 235 – 245.
TZERPOS V., HOLT R.C., MoJo: A distance metric for software clusterings. In: *Proc. of IEEE Working Conference on Reverse Enginerring,* IEEE Computer Society Press, Los Alamitos, USA, 1999, 187 – 195.
VIVANCO, R.A., AND JIN, D. 2007. Selecting object-oriented source code metrics to improve predictive models using a parallel genetic algorithm In: *Proceedings of OOPSLA'07,* 2007, 769 – 770.
WADEKAR, S. AND GOKHALE, S. 1999. Exploring cost and reliability tradeoffs in architectural alternatives using a genetic algorithm, In: *Proceedings of the 10th International Symposium on Software Reliability Engineering*, 1999, 104 – 113.
YOUDEN, W.J. 1961. How to evaluate accuracy. In *Materials Research and Standards, ASTM,* 1961.
ZHANG, C., SU, S., AND CHEN, J. 2006. A novel genetic algorithm for qos-aware web services selection, In: *LNCS* **4055**, 2006, 224 – 235.