Summer 2013

Search-based model summarization

Lokesh Krishna Ravichandran

SEARCH-BASED MODEL SUMMARIZATION

by

LOKESH KRISHNA RAVICHANDRAN

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2013

Approved by

ABSTRACT

Large systems are complex and consist of numerous components and interactions between the components. Hence managing such large systems is a cumbersome and time consuming task. Large systems are usually described at the model level. But the large number of components in such models makes it difficult to modify. As a consequence, developers need a solution to rapidly detect which model components to revise. Effective solution is to generate a model summary. Although existing techniques are powerful enough to provide good summaries based on lexical information (relevant terms), they do not make use of structural information (component structure) well. In this thesis, model summarization is considered as an optimization problem that combines structural and lexical information to evaluate possible solutions. A summary solution is defined as a combination of model elements (e.g., classes, methods, comments, etc.) that should maximize, as much as possible, the coverage of both automatically generated structural rules and lexical information. The results of the experiments are reported on 6 open source projects where the majority of generated summaries are approved by developers.

ACKNOWLEDGEMENTS

I take great pleasure in thanking Dr Marouane Kessentini for contributing his ideas to the SEARCH BASED MODEL SUMMARIZATION project. I would also like to thank him for his exemplary advice and guidance in my project work and my graduate career. His efforts and his help have helped me overcome the many obstacles and challenges of the project.

I would like to thank Dr Xiaoqing Liu and Dr Sriram Chellappan for their continued support and guidance. Their advices have been an instrumental part of the project.

I would also like to thank the professors and peers who have been part of my graduate career. The knowledge imparted by them helped face the many facets of the project.

Finally, I would like to thank my friends and family for their continued unfaltering support and advice.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Page

LIST OF TABLES

1. INTRODUCTION

Nowadays, the maintenance and evolution costs of software projects are significantly larger than the ones of initial developments. Many studies report that software maintenance consumes up to 90% of the total cost of a typical software project [1]. To make the situation worse, as the software ages, its code decays, this inevitably increases these costs. As a consequence, the goal of the research community as well as industry is to control the long-term maintenance activities costs.

The major activities of long-term maintenance include adding new functionalities, detecting bugs, correcting them, and modifying the system to improve its quality [2]. To perform them, developers explore and analyze the entire implementation details in systems with millions lines of code. This is a time-consuming process especially when good comments and documentations are missing. One of the efficient solutions is to provide a model summary of the system. As a consequence, developers can review software systems quickly and decide which components to analyze and modify.

A few works already proposed to generate code summaries by adapting text summarization techniques [3], [4]. Although these techniques are already powerful enough to provide good summaries by finding suitable key-words (lexical information), they may present some limitations related to support structural information. This structural dimension is not considered by adapting existing text summarization techniques to source code. In fact, these techniques focus mainly on lexical dimension (e.g., Latent Semantic Indexing [5], tf-idf [6], Vector Space Mode [6], etc.) to detect relevant terms. For example, large methods or classes containing huge numbers of relationships, in general, should be included in the model summaries. However, the name of a large class may not be detected as a relevant term using lexical metrics (e.g., word frequency).

A novel approach is proposed for automatic model summary generation taking into consideration both lexical and structural information. It is a three-step approach:

1. The first step generates structural rules based on software metrics and summary examples
2. The second step uses text retrieval techniques to determine the most important terms in the model (name of classes or methods, comments, etc.)
3. The third step exploits the two previous steps to evaluate the quality of potential summary solutions.

For the first step, regularities that can be found in good summary examples are translated into metrics-based rules covering the structure of good summaries. This is achieved using Genetic Programming (GP) [7]. The second step is achieved by adapting Latent Semantic Indexing (LSI) to find the important terms in the software system model. In the third step, the process starts by generating a summary solutions population based on combinations of model components. An evaluation function calculates, based on the two first steps, an average score between the two criteria of structural information (rules coverage) and lexical information (relevant words coverage). Due to the large number of possible component combinations to generate summaries, a simulated annealing (SA) algorithm [8] is used since a local method is enough to explore this search space.

The primary contributions of the approach can be summarized as follows:

1. A new approach is introduced for generating model summaries, taking into consideration both structural and lexical information. In addition, the approach generates various types of summaries depending on the tasks to accomplish (e.g., detecting defects, adding new functionalities, etc.).

2. The proposed technique is fully automatable from the structural rules generation to the summaries generation and evaluations.

3. The results of the evaluation of the approach on 6 open source projects are reported. A reference summary example for each of these projects is defined. A six-fold cross validation procedure is utilized here. For each fold, one open source project summary is evaluated by using the remaining five systems. The majority of summaries are approved by developers.

The major limitation of the approach is that the process requires a base representing good summaries to generate structural rules. Results indicate that some open source summary examples seem to be usable and could serve as a starting point for a company wishing to use this approach.

2. BACKGROUND

2.1 INTRODUCTION

During software maintenance, developers often cannot read and understand the entire source code of a system and rely on partial comprehension, focusing on the parts strictly related to their task at hand. Recent studies have shown that developers spend more time reading and navigating the code than writing it. During these activities developers often only skim the source code (e.g., read only the header of a method and maybe the leading comments when available). Models are useful but tend to be large or unclear on most occasions. Developers are forced to go through voluminous amounts of code to make sense of the model elements under concern. This takes significant effort and time. Program Comprehension accounts for 50% of the time spent on software maintenance. The two activities (i.e., skimming and reading the whole implementation) are two extreme tactics; the former is very quick yet it can lead to misunderstanding, while the latter is time consuming. Model Summarization is the tradeoff, offering developers a description of the software system while still being concise in nature.

2.2 MODEL SUMMARIZATION

Model summarizations are short yet accurate descriptions of software systems. Models are utilized to make the software system more understandable to the user. But with increase in the software components the models themselves tend to be highly complex for users to decipher. Model summarization allows the user to obtain a summarized description of the model for a specific software problem. Since model summaries have model elements as a part of the description the user is able to focus on those specific entities rather than the system as a whole. Software analysts who perform these summarizations tend to spend a lot of time in developing these summaries due to the large and complex nature of the systems at hand. Use of automated summarization techniques allows the summaries to be generated at a much faster pace and reduce the work load on the analyst.

In general, existing techniques can be classified as extractive document summarization or abstractive document summarization. Extractive summarization involves assigning scores to some sentences of the document and extracting those with highest scores, while abstraction summarization usually needs reformulation and sentence compression [14]. The proposed work can be classified in the category of extractive summarization since the process extracts code fragments from the original source code to include them in the model-summary.

Recently, Haiduc et al. presented a leading work for source code summarization [3], [4]. The authors combine several text summarization techniques, based on text retrieval (e.g., LSI, Vector Space Model, etc.), to generate source code summaries by finding the suitable relevant terms. Haiduc et al. work is the closest one to the work. In fact, the process makes use of LSI to extract relevant terms in the second step of th approach. However, the main difference with this process is the structural information that is used to generate a summary. In addition, the authors did not distinguish between different summary types that depend to the software engineering task to perform.

Another similar work was proposed by Murphy [15] that proposes two techniques. The first technique, the software reflection model technique, summarizes selected structural information. The second technique, the lexical source model extraction technique, supports the summarization process by facilitating the analysis of system artifacts. However, these two techniques were applied to lightweight summarization of software. In addition, this approach requires more programmer-interaction than the proposed technique.

Summarization techniques were applied in different software engineering problems. In Rastkar et al. investigated the summarization of bug reports using machine learning techniques [16], [17]. Kuhn et al. [18] introduce semantic clustering, based on Latent Semantic Indexing, to group source documents that use similar vocabulary. Another work proposed by Poshyvanyk et al. [19] to the problem of concept location in source code using a combination of Formal Concept Analysis (FCA) and LSI. The approach allows the user to search source code and related textual documentation by writing queries. Marcus et al. [20] proposed another technique for concept location based on LSI in the absence of external documentation. The method uses LSI to find semantic similarities between user queries and modules of the software to locate concepts of interest in the source code.

In this approach the process tends to focus on model summarization as an optimization problem. This falls under a field of software engineering known as search based software engineering. The next section explains about search based software engineering and related works in the field.

2.3 SEARCH BASED SOFTWARE ENGINEERING

Search Based Software Engineering (SBSE) uses a search-based approach to solve optimization problems in software engineering [21], [22]. The main object of SBSE is to reformulate software problems as search problems. The term SBSE was first used by Harman and Jones in 2001. The term 'search' is used to refer to the meta-heuristic search–based optimization techniques. Once a software engineering task is framed as a search problem, by defining it in terms of solution representation, fitness function and solution change operators, there are many search algorithms that can be applied to solve that problem. By using an optimization technique, the complete search space can be analyzed to identify solutions for the problem under consideration. This becomes significantly important to handle complex software systems with a large number of components that have a high degree of coupling among them.

Search Based Software Engineering seeks a fundamental shift of emphasis from solution construction to solution description. Rather than devoting human effort to the task of finding solutions, the search for solutions is automated as a search, guided by a fitness function, defined by the engineer to capture what is required rather than how it is to be constructed.

The prime objective is to produce a good summary of model to a specific software engineering problem that will help the user in solving the task at hand. Due to the complexity and size of real time systems, a heuristic search would be more useful analyze the complete software system.

To this end, model summarization can be considered as an optimization problem. In fact, various software engineering tasks have been handled as an optimization problem, and based on the survey proposed by Harman [22] the proposed work represents the first attempt to treat the problem of model summarization as an optimization problem. Considering current approaches, the idea of treating model summarization as a combinatorial optimization problem to be solved by a search-based approach was not studied before.

3. PROBLEM STATEMENT

In this section, the problem of model summarization is described. Furthermore, the specific problems that are addressed by the approach are detailed.

Summary definition: In general, a summary can be defined as a text that is produced from different original texts. The summary includes the significant information contained in the original texts, and is shorter than, at least, half of the original texts [9]. Similarly, source code summarization can be considered as the process of finding the most important information from a source code to produce a reduced version for a specific task (e.g., adding new functionalities, detecting and correcting defects, etc.).

Various techniques have been already proposed in the literature to support text summarization [5], [9]. The majority of these are based on information retrieval and statistical techniques (e.g., Latent Semantic Indexing, Vector Space Model, tf-idf, log, etc.) to find the relevant words in the corpus (documents). Then, these key-words are included in the text-summary.

These techniques have, recently, been adapted to generate code summaries. There are two main steps. The first one creates a source code corpus containing the different code elements (e.g., classes, methods, comments, etc.) filtered using a stop-list (e.g., for, while, int, etc.). Then, the second step applies a combination of text summarization techniques to the code corpus.

Although these techniques are already powerful enough to provide good code summaries, there are some difficulties that inherent to the complex nature of models of a system. Here is the description of the most important difficulties and how they affect an automation process.

— One possible approach is to generate summaries manually, but this is labor-intensive, fastidious and is impractical in situations where the system keeps evolving. In addition, the summary will not be updated when the system is updated, resulting in a summary that is outdated.

— Models are more complex than any other classical text documents. Indeed, relevant summary information for models should be collected from both structural and lexical dimensions. However, most existing work is based on lexical information to determine relevant terms. For example, important model components cannot be detected because the terms composing it are not very frequent for example. For this reason, the process needs to take into consideration the model component structure (e.g., number of methods, number of relationships, etc.) that can be supported, for example, using software metrics.

— When this structural information is available, it is difficult on select what is the component structure information that should be selected to generate good model summaries. In fact, component structure includes a huge amount of information that could be formalized in terms of metrics.

— There is no efficient manner to provide a model summary that depends on the software engineering task to accomplish. Indeed, a summary for detecting and correcting defects is different than another one to add new functionality.

— After generating a model summary, the sequence of presenting the model components composing this summary is important. Developers could spend significantly different time to understand the same summary, but presented in different ways (sequences).

4. AUTOMATED MODEL SUMMARIZATION

4.1 APPROACH OVERVIEW

An approach is proposed based on 3 steps to address the problems involved in producing a good model summary. They are

1. Structural rules generation: The process uses examples of good summaries, manually defined, to derive rules that describe the structure of the summary.
2. Lexical information: The process uses Latent Semantic Indexing (LSI) to find the relevant terms that should be included in the summary.
3. Summary generation: The process uses the derived rules and relevant terms detected in the two first steps to select the best summary solution. The general structure of the approach is illustrated in Figure 4-1.

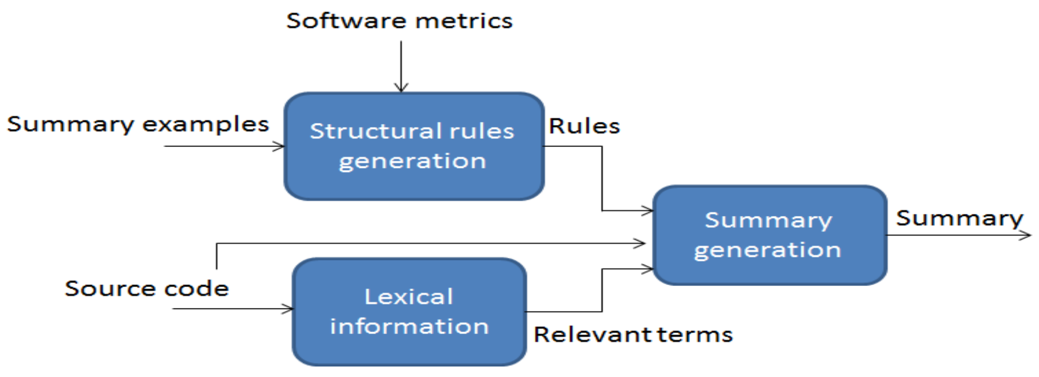


Figure 4-1: Approach Overview

4.1.1 Structural Rules Generation. In this step, the knowledge from summaries examples, defined manually by developers, is used to generate rules describing the structure of good summaries. A summary example is defined as a software system model to summarize and a reference summary. This step takes as inputs a base (i.e., a set) of summaries examples and takes as controlling parameters a set of software metrics [10]. This step generates a set of rules. The rule generation process chooses randomly from the metrics list a combination of metrics (and their threshold values) that best cover the summaries of the base of examples. For example, the following rule states that a class *c* having more than 6 attributes and 10 methods should be included to the summary:

R1: IF NAD(c)≥6 AND NMD(c)≥10 Then AddtoSummary(c)

In this rule example, the number of attributes (NAD) and the number of methods (NMD) of a class correspond to two software metrics. A class will be added to the summary whenever both thresholds of 6 attributes and 10 methods are exceeded.

The summary examples are defined manually by developers. The use of such examples has many benefits. First, it allows deriving structural rules that are closer to the developers “traditions” to generate a model summary. These rules will be more precise and more context faithful. Second, the summary examples can be classified using the software engineering tasks. Thus, completely different rules can be generated that depends to the expected software engineering task to do (e.g., defects detection or correction, etc.). Finally, learning from examples reduces the length of the generated summary that will be coherent with the examples-length average.

The rule generation process is executed periodically over large periods of time using the base of examples. The rule generation step needs to be re-executed only if the base of examples is updated with new summary examples. In fact, adding new examples improves the quality of the rules.

4.1.2 Lexical Information Extraction. After transforming the source code to text corpus, the LSI technique is used to find the most relevant terms in the source code. A stop list is used to filter some programming key words. The determination of relevant terms is based on a cosine similarity score between terms and documents (source code) that represents two vectors. This step is not one of the contributions in this thesis since LSI was applied before in previous works [3] and [4] to extract lexical information.

4.1.3 Summary Generation. The final step takes as controlling parameters the generated structural rules and the set of relevant terms. The summary generation step takes as input a software code to summarize and recommends a summary solution consisting of a code fragments combination. The process of generating a correction solution can be viewed as the mechanism that finds the best way to combine some subset, among all available code elements combination, in such a way to best maximize the coverage of rules set execution and relevant terms.

The second and third steps are aimed to be executed more frequently than the rule generation step, i.e., each time when there is a need to generate a model summary.

4.1.4 Complexity. In the first step, the approach assigns a threshold value randomly to each metric and combines these threshold values within logical expressions (union OR; intersection AND) to create rules. The number *m* of possible threshold values is usually very large and the rule generation process consists of finding the best combination between *n* metrics. In this context, the number *NC* of possible combinations to be explored is at least:

$$NC = (n!)^m$$

This value quickly becomes huge. For example, a list of 5 metrics with 6 possible thresholds necessitates the evaluation of up to 120^6 combinations.

Consequently, the rule generation process is a combinatorial optimization problem. Due to the huge number of possible combinations, a deterministic search is not practical, and the use of a heuristic search is warranted. To explore the search space, a global heuristic search by means of genetic programming is used [7].

As far as the third step of summary generation is concerned, the size of the search space is determined not only by the number of possible model component combinations, but also by the order in which they are presented to developers. Formally, if *k* is the number of components in the model to summarize, then the number *NS* of possible summary solutions is given by all the permutations of all the possible subsets and is at least equal to:

$$NS = (k!)^k$$

Due to the large number of possible summary solutions, another heuristic-based optimization method is used to generate solutions. To this end, an adaptation of the simulated annealing algorithm is proposed [8].

4.1.5 Motivating Example. In order to demonstrate the approach more descriptively a running example is utilized. One of the systems that was used for testing is used to describe the approach. Figure 4-2 is a model representation of the LlamaChat software.

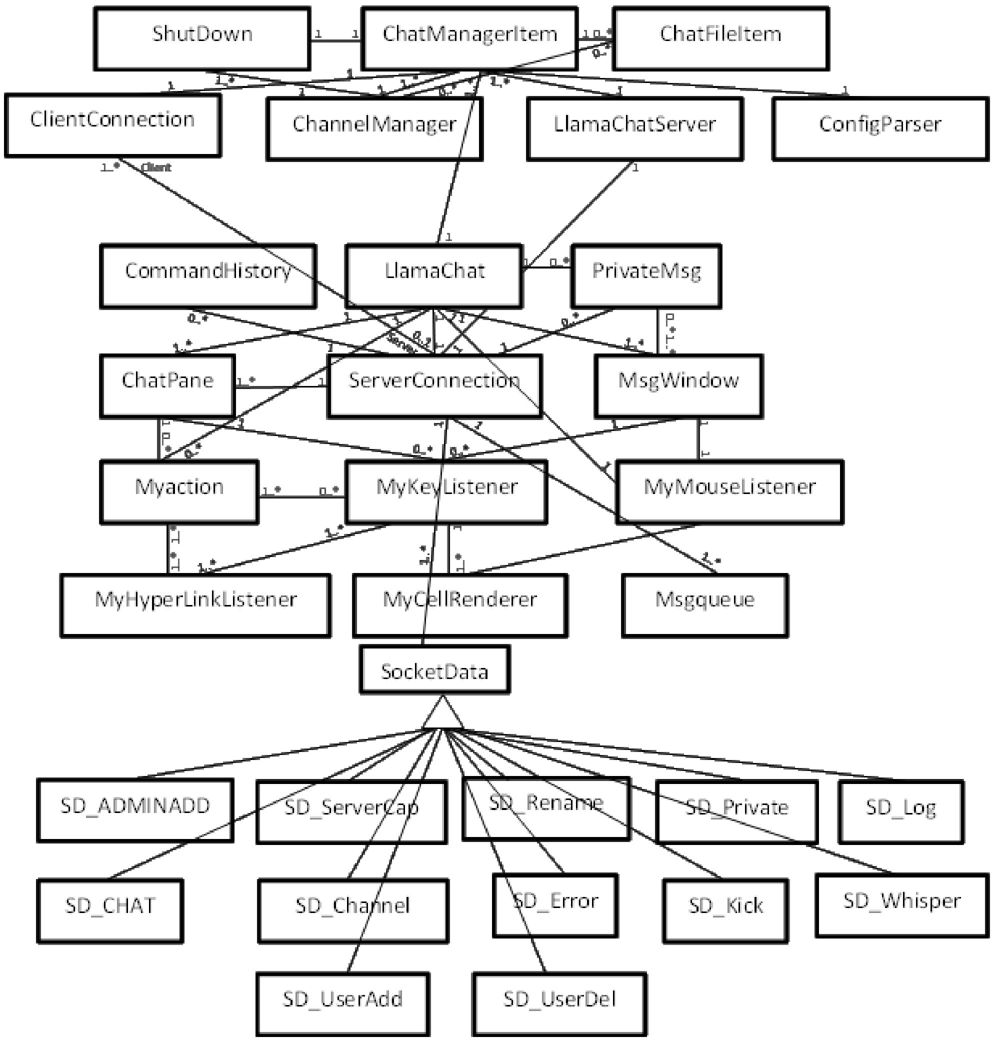


Figure 4-2: Motivating Example LlamaChat

The Llamachat has about 31 components with multiple interactions between the components. These interactions are the cardinal relations between the different components. These components have aggregation relations, specialization relations, 1 to 1 relations, 1 to many relations, 0 to many relations etc. As the number of components and its increases it becomes very difficult for software engineers to manage these large systems even at the model level. Hence software engineers make use of model summaries. But these model summaries need to be specific for the software engineering task at hand. The summary in Figure 4-3 is specifically developed for a particular task (detecting and correcting bugs).

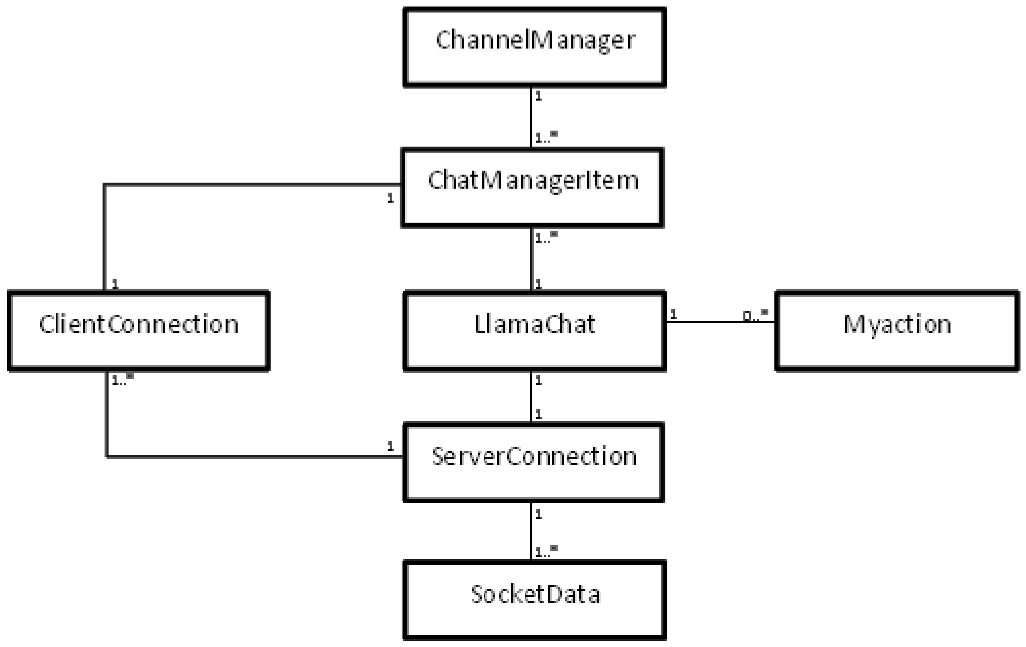


Figure 4-3: Base of Examples - Good Summary

This makes it much easier for software engineer to maintain the system and perform specific task that are necessary make the system more efficient. Figure 4-3 is an example of a good summary that the approach would use in its initial step. The summary would be used a basis to develop rule that conform to the summary.

LSI is used to read specific terms such as component names (classes, methods etc.) from the source code. These are necessary for making sure that they are added to the summary that is generated by the approach. Figure 4-4 is a representation of a generated summary.

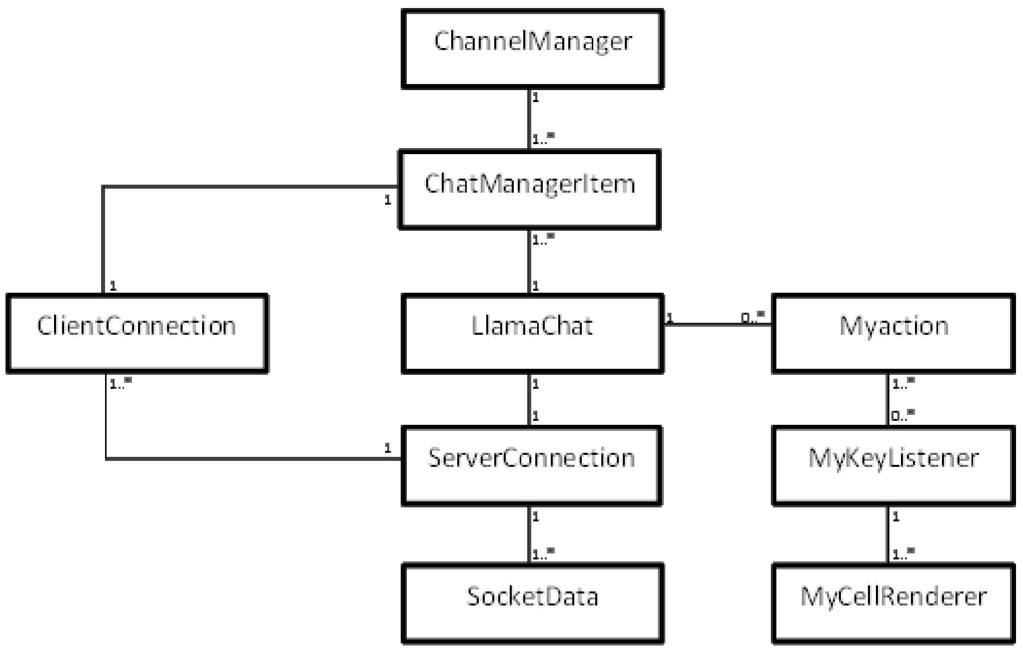


Figure 4-4: Generated Summary

The generated summary tries to be structurally similar to the base of example while including relevant terms. Based on observation the generated summary closely conforms to the summary produced by the experts. The correctness and precision of the solution generated is evaluated to make sure that the best solution is produced.

4.2 HEURISTIC SEARCH MODEL SUMMARIZATION

This section describes how genetic programming (GP) can be used to generate structural rules, and how the simulated annealing algorithm (SA) [8] can be adapted to generate model summaries. Furthermore, details about LSI adaptation that is used to extract relevant terms is given. To apply GP and SA to a specific problem, the following elements have to be defined: representation of the individuals, definition of the fitness function to evaluate individuals for their ability to solve the problem under consideration, and creation of new individuals using some operators to explore the search space.

4.2.1. Genetic Programming for Structural Rules Generation. Genetic programming is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution [11]. The basic idea is to explore the search space by making a population of candidate solutions, also called individuals, evolve toward a "good" solution of a specific problem.

In genetic programming, a solution is a (computer) program which is usually represented as a tree, where the internal nodes are functions, and the leaf nodes are terminal symbols. Both the function set and the terminal set must contain elements that are appropriate for the target problem. For instance, the function set can contain arithmetic operators, logic operators, mathematical functions, etc.; whereas the terminal set can contain the variables (attributes) of the target problem. Every individual of the population is evaluated by a fitness function that determines a quantitative measure of its ability to solve the target problem. The exploration of the search space is achieved by the evolution of candidate solutions using selection and genetic operators such as crossover and mutation. The selection operator ensures selection of individuals in the current population proportionally to their fitness values, so that the fitter an individual is, the higher the probability is that it be allowed to transmit its features to new individuals by undergoing crossover and/or mutation operators. The crossover operator insures the generation of new children, or offspring, based on parent individuals. The crossover operator allows transmission of the features of the best fitted parent individuals to new individuals. This is usually achieved by replacing a randomly selected sub tree of one-parent individual with a randomly chosen sub tree from another parent individual to obtain one child. A second child is obtained by inverting parents. Finally, the mutation operator is applied, with a probability which is usually inversely proportional to its fitness value, to modify some randomly selected nodes in a single individual. The mutation operator introduces diversity into the population and allows escaping from local solutions found during the search.

Once the selection, mutation and crossover operators have been applied with given probabilities, the individuals in the newly created generation are evaluated using the fitness function. This process is repeated iteratively, until a stopping criterion is met. The criterion usually corresponds to a fixed number of generations. The result of GP (the best solution found) is the fittest individual produced over all generations. A high level view of the GP approach to the structural rules generation is summarized in Figure 4-5.

Algorithm: Structural rules generation for model summarization

Input:

Set of software metrics M

Set of summary examples SE

Process:

```
1. I:= rules(M, Max_Number_Rules)
2. P:= set_of(I)
3. initial_population(P, Max_size)
4. repeat
5.    for all I in P do
6.        summary := execute_rules(I)
7.        fitness(I):= compare(summary, SE)
8.    end for
9.    best_solution := best_fitness(I);
10.            P := generate_new_population(P)
11.  it:=it+1;
12. until it=max_it
13. return best_solution
```

Output:

best_solution: structural rules

Figure 4-5: Pseudo Code for GP Adaptation

As this figure shows, the algorithm takes as input a set of software metrics and a set of summary examples that were manually proposed by different developers for some systems, and finds a solution that corresponds to the set of structural rules for code summarization that best covers summaries in the base of examples.

Lines 1–3 construct the initial GP population which is a set of individuals that define possible rules. The function rules (M, Max_Number_rules) returns an individual *I* by randomly selecting/combining a set of metrics/thresholds. The function set_of(I) returns a set of individuals, *i.e.*, structural summary rules, that corresponds to a GP population. Lines 4–13 encode the main GP loop, which explores the search space and constructs new individuals by combining metrics within rules. During each iteration, ethe quality of each individual in the population is evaluated, and save the individual having the best fitness (line 9). A new population (p+1) of individuals (line 10) is generated by iteratively selecting pairs of parent individuals from population p and applying the crossover operator to them; each pair of parent individuals produces two children (new solutions). Both the parent and child variants are included in the new population p. Then, the mutation operator is applied with a probability score for both parent and child to ensure the solution diversity; this produces the population for the next generation. The algorithm terminates when the termination criterion (maximum iteration number) is met, and returns the best set of rules (best solution found during all iterations).

4.2.2 Individual Representation. An individual is a set of IF – THEN rules. For example, Figure 4-6 shows the rule interpretation of an individual. In these rules, the number of methods (NMD) of a class, the number of comments (NCOM) and the number of lines of code in a method (LOCMETHOD) correspond to three metrics. These are intrinsic components of the Individual rule and are parts of its structure. For the first rule, a class will be added to the summary whenever both thresholds of 15 methods and 4 comments are exceeded.

R1: IF (NMD(c1) $\geq$ 15 OR (NCOM(c1) $\geq$ 4) THEN add_to_summary(c1)

R2: IF (LOCMETHOD(m1,c2) $\geq$ 151 THEN add_to_summary(c2.m1)

Figure 4-6: Rule Interpretation of an Individual

Consequently, a structural rule has the following structure:

IF "Combination of metrics with their threshold values" THEN "add associated code elements to the code summary"

The "if clause" describes the conditions or situations under which a model element should be added to the summary. These conditions correspond to logical expressions that combine some metrics and their threshold values using logic operators (AND, OR).

One of the most suitable computer representations of rules is based on the use of trees. In this case, the rule interpretation of an individual will be handled by a tree representation which is composed of two types of nodes: terminals and functions. The terminals (leaf nodes of a tree) correspond to different software metrics with their threshold values. The functions that can be used between these metrics correspond to logical operators, which are Union (OR) and Intersection (AND). This tree representation corresponds to an OR composition of three sub-trees, each sub tree representing a rule: R1 OR R2 OR R3.

4.2.3 Genetic Operators. Selection **-** To select the individuals that will undergo the crossover and mutation operators, stochastic universal sampling (SUS) [11] was used, in which the probability to select an individual is directly proportional to its relative fitness in the population. For each iteration, SUS was used to select *population_size/2* individuals from population p to form population p+1. These (population_size/2) selected individuals will "give birth" to another (population_size/2) new individuals using crossover operator.

Crossover **-** Two parent individuals are selected, and a sub tree is picked on each one. Then, the crossover operator swaps the nodes and their relative sub trees from one parent to the other. Each child thus combines information from both parents.

Mutation **-** The mutation operator can be applied either to function or terminal nodes. This operator can modify one or many nodes. Given a selected individual, the mutation operator first randomly selects a node in the tree representation of the individual. Then, if the selected node is a terminal (threshold value of a quality metric), it is replaced by another terminal. The new terminal either corresponds to a threshold value of the same metric or the metric is changed and a threshold value is randomly fixed. If the selected node is a function (AND operator, for example), it is replaced by a new function (i.e., AND becomes OR). If a tree mutation is to be carried out, the node and its sub trees are replaced by a new randomly generated sub tree.

4.2.4 Evaluation of an Individual. The quality of an individual is proportional to the quality of the different rules composing it. In fact, the execution of these rules on the different projects extracted from the base of examples adds various code elements to the code summaries. Then, the quality of a solution (set of rules) is determined with respect to the number of common code elements in comparison to the expected ones in the base of examples. In other words, the best set of rules is the one that is generate the most similar summaries to the reference ones.

The encoding of an individual should be formalized in a fitness function that quantifies the quality of the generated rules. The goal is to define an efficient and simple (in the sense of not computationally expensive) fitness function.

The fitness function aims to maximize the generated code summaries and the expected ones in the base of examples. In this context, the fitness function of a solution is defined, normalized in the range [0, 1], in Figure 4-7:

$$f_{norm} = \frac{\frac{\sum_{i=1}^{p} a_i}{t} + \frac{\sum_{i=1}^{p} a_i}{p}}{2}$$

Figure 4-7: Fitness Function

Where t is the number of model elements in the base of examples, p is the number of model elements in the generated summary, and a_i has value 1 if the ith model element exists in the base of examples, and value 0 otherwise.

Consider Figure 4-3 and Figure 4-4. Based on the fitness function in Figure 4-7 the fitness of generated summary in Figure 4-4 could be evaluated. The fitness of the generated summary is given in Figure 4-8. Based on observation the generated summary has a fitness of 0.86 or 86 %.

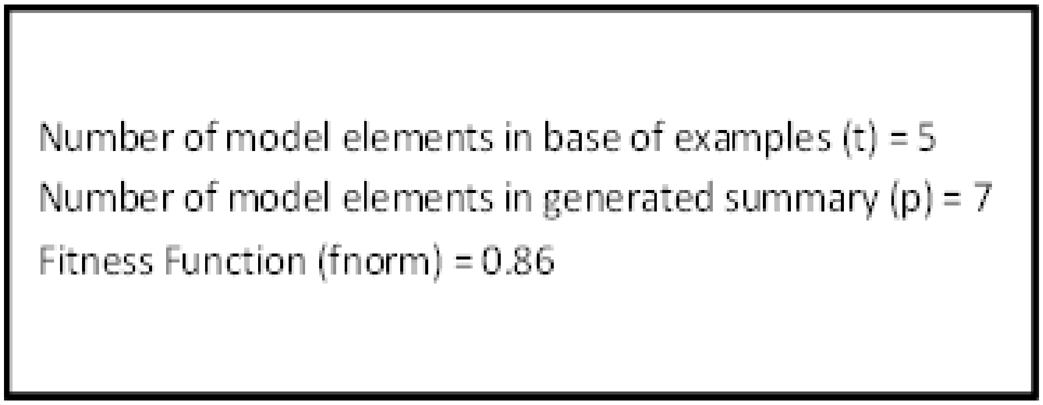

Number of model elements in base of examples (t) = 5
Number of model elements in generated summary (p) = 7
Fitness Function (fnorm) = 0.86

Figure 4-8: Fitness of Generated Solution for Motivating Example

4.2.5 Lexical Information Extraction. Latent semantic indexing (LSI) has been proposed as an automated way to retrieving documents based upon keywords contained in a document. In addition, LSI statistically analyzes the degree of semantic relationship that exists between terms and documents.

Before using LSI to select relevant terms, a document collection was created that corresponds to the source code to be summarized (e.g., methods, classes, etc.). The approach uses a stop list to filter these documents (programming terms, identifiers, etc.). A matrix is generated to calculate the similarities between terms and the document. These similarities scores are used to select the best relevant terms.

4.2.6 Simulated Annealing for Model Summary Generation. Simulated Annealing (SA) [8] is a local search algorithm that gradually transforms a solution following the annealing principle used in metallurgy. Starting from an initial solution, SA uses a pseudo-cooling process where a pseudo temperature is gradually decreased. For each temperature, the following three steps are repeated for a fixed number of iterations:

1. Determine a new neighboring solution;
2. Evaluate the fitness of the new solution;
3. Decide on whether to accept the new solution in place of the current one based on the fitness function and the temperature value.

Solutions are accepted if they improve quality. When the quality is degraded, they can still be accepted, but with a certain probability. The probability is high when the temperature is high and the quality degradation is low. As a consequence, quality-degrading solutions are easily accepted in the beginning of process when the temperatures are high, but with more difficulty as the temperature decreases. The introduction of a stochastic element in the decision process avoids being trapped in a local minimum solution. The temperature begins with a high value, for a high probability of accepting a solution during the early iterations. Then, it decreases gradually (cooling phase) to lower the acceptation probability as the iteration sequence advances. For each temperature value, the three steps of the algorithm are repeated for a fixed number of iterations.

One attractive feature of the SA algorithm is that it is problem-independent and can be applied to most combinatorial optimization problems. However, SA is usually slow to converge to a solution.

4.2.7 Summary Representation. The SA algorithm starts with an initial solution generated randomly from the model representation of the systems in the base of examples. Summary solutions are coded by assigning a model element to each construct (dimension) in order to form a vector. In fact, the set of potential summary solutions is viewed as points in an n-dimensional space where each dimension corresponds to one of the n code elements of the systems in the base of examples. The resulting n-tuple of code elements then defines a vector position in the n-dimensional space. For instance, Figure 4-9 shows an example of a summary generated randomly for the open source system StringSearch.

BNDMWildcards	StringSearch.search Bytes()	StringSearch.searchChars()	Dispatch

Figure 4-9: Summary Solution Representation

The vector contains four-dimensions including two classes and two methods. This is a simplified way to represent a model summary. In fact, the process uses different model granularity/abstraction levels for the solution representation. For example, w=the process could use as vector dimensions for any instances of JAVA-metamodel elements.

4.2.8 Change Operators. A change operator is needed to modify a candidate solution in order to produce a new one. In this case, it modifies a generated summary in order to produce a new one.

This is accomplished by changing, randomly, some model elements composing the summary, which is equivalent to changing the coordinates of the solution in the search space. In fact, the change operator involves randomly choosing a number of dimensions (model elements) and replacing them by new ones extracted from the model to summarize. For instance, Figure 4-10 shows a new solution derived from the one of Figure 4-9. Only one dimension (number 2) is selected for change by another model element. The maximum number of dimensions to change is a parameter of the SA algorithm.

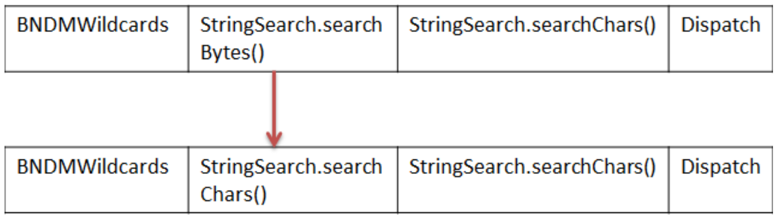

BNDMWildcards
StringSearch.searchBytes()
StringSearch.searchChars()
Dispatch
BNDMWildcards
StringSearch.searchChars()
StringSearch.searchChars()
Dispatch

Figure 4-10: SA Change Operators

4.2.9 Summary Evaluation. The fitness function quantifies the quality of a summary solution. This evaluation function must consider the two following aspects:

1. Structural information coverage: a good summary solution should be coherent, as much as possible, with structural rules;
2. Lexical information coverage: a good summary solution should contain, as much as possible, the relevant terms extracted by LSI.

In this context, the fitness function aims to maximize the number of structural rules and relevant terms to cover. In this context, the fitness function of a solution is defined, normalized in the range [0, 1], as seen in Figure 4-11.

$$f_{norm} = \frac{S+L}{2},$$

Where, $S = \dfrac{\text{Number of structural rules covered}}{\text{Number of structural rules}}$,

and $L = \dfrac{\text{Precision(relevant terms)} + \text{Recall(relevant terms)}}{2}$

Figure 4-11: Normalized Fitness Function

.

5. EVALUATION

5.1 INTRODUCTION

Analyzing the quality of a model summary involves a complex process consisting of both automated and manual procedures. In this section the mains goals and objectives behind the evaluation phase are explored. The feasibility of the approach is demonstrated by conducting an experiment with 6 open source systems. The section starts by presenting the experimental setting. The experimentation is used to obtain results which would then be used to analyze the performance and accuracy of the approach.

Then, the section describes and discusses the obtained results. Evaluation is particularly done on how much benefit the summary can provide to the user in performing some specific software engineering tasks (e.g., adding new functionality, detecting and correcting defects).

5.2 GOALS AND OBJECTIVES

The goal of the study is to evaluate the efficiency of the approach for generating model summary from the perspective of a software analyst who is expected to perform some specific maintenance tasks.

The results of the experiment are aimed at answering the following research questions:

— *RQ1:* To what extent can the proposed approach generates good summaries?

— *RQ2:* What types of summary the proposed approach can generate?

To answer RQ1, a 6-fold cross validation procedure based on 6 open source systems is used. For each fold, one summary is generated by using the remaining 5 examples. Then the produced summary of each fold is checked for quality. The quality of a summary was measured by two methods: automatic evaluation (AE) and manual evaluation (ME). Automatic evaluation consists of comparing the derived summary to a reference one, element by element. AE method has the advantage of being automatic and objective. However, since different possibilities can exist to summarize a source code, AE could reject a good summary because it yields a different summary from the one provided. To account for those situations, ME is also used which manually evaluates the generated summary, here again element by elements.

To answer RQ2, investigation was performed on the types of summaries that could be generated using the approach. Expert users who are knowledgeable about the different open source systems were asked to provide summary examples that depend on a maintenance tasks: detecting and correcting defects. Thus, the experts provide two different bases of examples; each of them contains five summaries that depend on these tasks. Then, the differences between the generated summaries were studied, depending on the used examples.

5.3 EXPERIMENTAL SETTINGS

The development environment used for the implementation of the GA is Eclipse with Java as programming language. The programming language was chosen for its high degree of portability and performance. Six open-source Java projects were used to perform the experiments: LlamaChat, AICodeSummary, ECore2Predicates, Jlayer, Gantt, and Xerces [13]. The size of these systems is between 8 and 1159 classes. The classes are used as a measure to observe the complexity of a given system.

These libraries were chosen to illustrate the scalability of the process. Table 5-1 provides some descriptive statistics about these projects. These projects were used and analyzed in the past by the group of graduate and undergraduate students asked to define manually summary examples.

Table 5-1: Project Statistics

Systems	**Number of classes**	**KLOC**
Xerces	1159	104
GanttProject	678	48
Jlayer	61	7
Llamachat	31	2
AICodeSummary	24	2
ECore2Predicates	8	2

In fact, six groups of students analyzed the libraries to generate a summary for each system. The summary version depends on the task to do; detecting and correcting bugs.

Ten different software metrics were used to generate the individuals (rules) for detecting the summaries. Table 5-2 lists these software metrics.

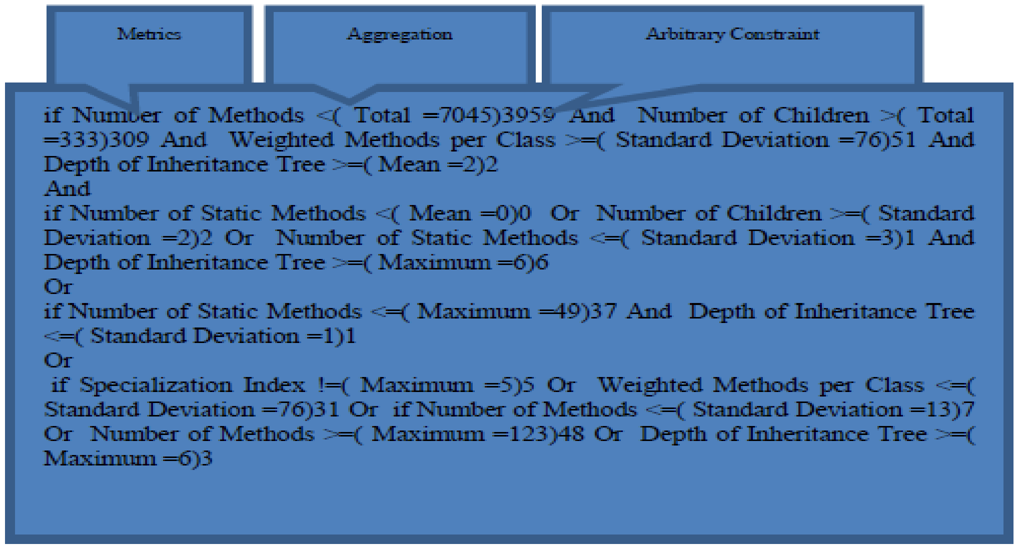
Table 5-2: Software Metrics used in Defining Individuals

S.No	Metrics
1	Number of Overridden Methods
2	Number of Attributes
3	Number of Children
4	Number of Methods
5	Depth of Inheritance Tree
6	Lack of Cohesion of Methods
7	Number of Static Methods
8	Specialization Index
9	Weighted Methods per Class
10	Number of Static Attributes

An Individual is formed using a subset of these metrics. The subset is formed randomly with an equal probability of selecting any metrics from the set. Each Individual may comprise of one to five metrics. The number of metrics a rule might contain is again chosen randomly.

The threshold values for the metrics in the individual function are chosen randomly from a list of four aggregate functions. The four aggregate functions that are utilized are Maximum, Mean Standard Deviation and Total of the metric in use.

The Relational operator between the metrics and the constraints are chosen arbitrarily. The choice of arbitrary metrics and constraints is performed to preserve the randomness of the individual population. Figure 5-1 illustrates the different parameters used in the generation of an individual.



if Number of Methods <(Total =7045)3959 And Number of Children >(Total =333)309 And Weighted Methods per Class >=(Standard Deviation =76)51 And Depth of Inheritance Tree >=(Mean =2)2
And
if Number of Static Methods <(Mean =0)0 Or Number of Children >=(Standard Deviation =2)2 Or Number of Static Methods <=(Standard Deviation =3)1 And Depth of Inheritance Tree >=(Maximum =6)6
Or
if Number of Static Methods <=(Maximum =49)37 And Depth of Inheritance Tree <=(Standard Deviation =1)1
Or
if Specialization Index !=(Maximum =5)5 Or Weighted Methods per Class <=(Standard Deviation =76)31 Or if Number of Methods <=(Standard Deviation =13)7 Or Number of Methods >=(Maximum =123)48 Or Depth of Inheritance Tree >=(Maximum =6)3

Figure 5-1: Parameters in an Individual

These rules are applied on the systems to obtain components that satisfy these rules. These components are then used by the algorithm to evaluate the fitness of the solutions. The running example illustrated section 4 could be used to demonstrate this process. Consider the example generated summary Figure 4-4. This is compared with the expected summary Figure 4-3 in Table 5-3

This allows the evaluation of the fitness of the proposed candidate solution based on the components identified by the candidate solution.

Table 5-3: Generated Summary Vs Expected Summary

Model Component	Base of Examples	Generated Summary
ChannelManager	X	X
ChatManagerItem	X	X
ClientConnection	X	X
LlamaChat	X	X
Myaction	X	X
ServerConnection	X	X
MyKeyListener		X
SocketData	X	X
MyCellRenderer		X

5.4 TEST CASE

Xerces is used as a test case to illustrate the efficiency and the effectiveness of the approach. Xerces is Apache's collection of software libraries for parsing, validating, serializing and manipulating XML. The library implements a number of standard APIs for XML parsing, including DOM, SAX and SAX2. The implementation is available in Java, C++ and Perl programming languages.

The test case is used to illustrate the growth of fitness for a solution in the approach. 100 iterations on Xerces is performed. Figure 5-2 illustrates how the fitness grows for Xerces from iteration 91 to 100. The crossover and mutations functions improve the fitness of the solution at each iteration level.

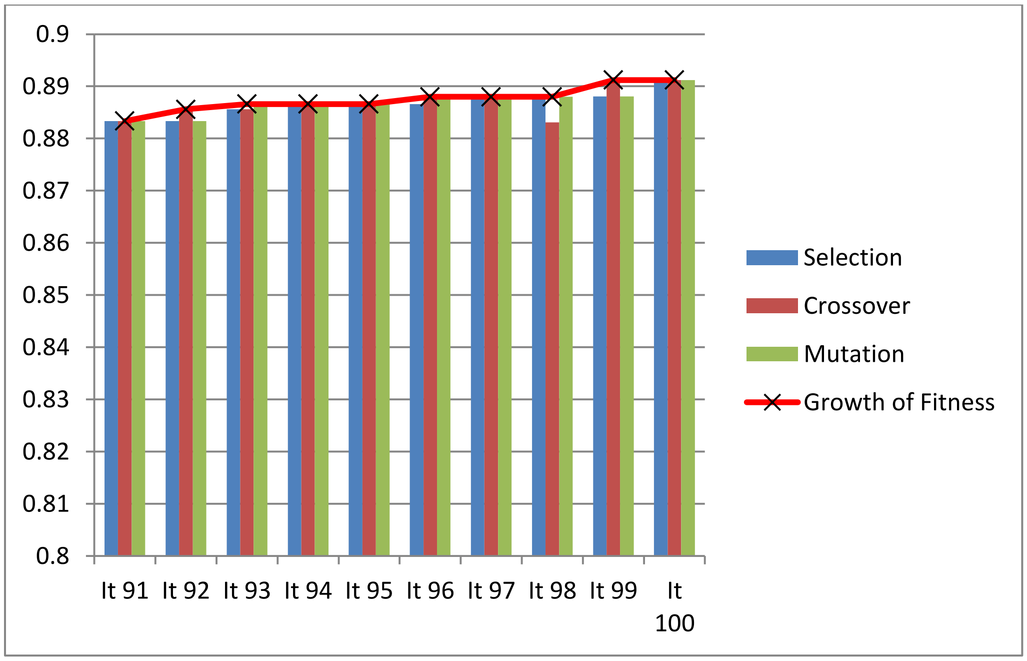


Figure 5-2: Growth of Fitness for Xerces

While Figure 5-2 gives the automatic evaluation results of the different fitness functions it is also necessary to perform manual evaluation of the given solution. Figure 5-3 gives the comparative analysis between AE and ME performed on solutions generated for Xerces. Based on observation that ME fitness is better than AE fitness since alternative solutions are observed to those proposed by the experts.

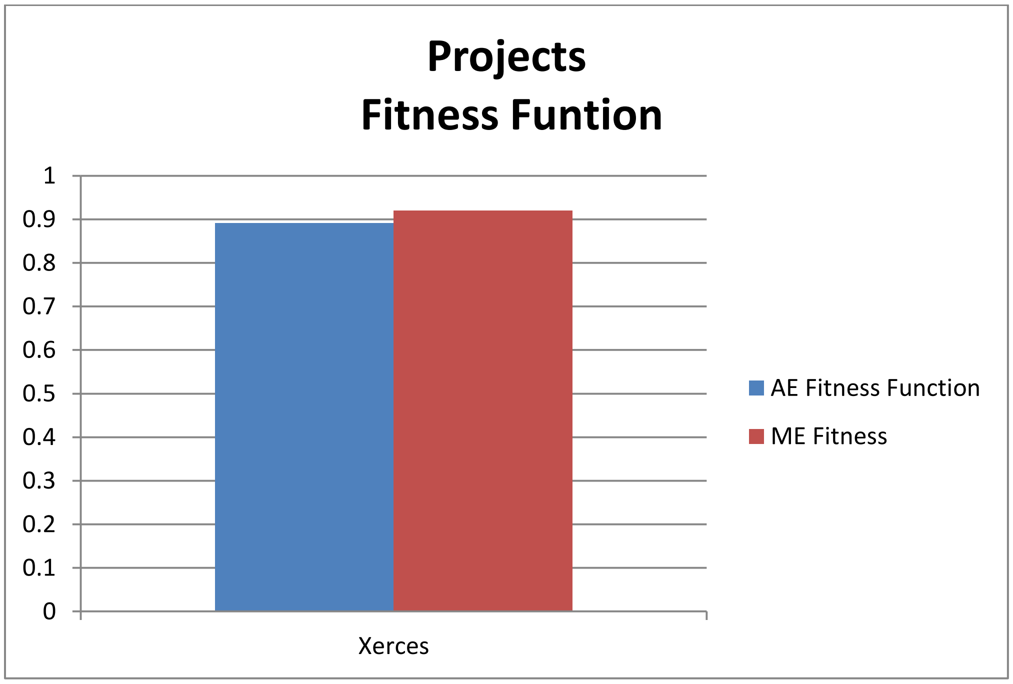


Figure 5-3: Comparative Evaluation for Xerces

100 iterations are performed on Xerces using the GA approach, random search and random search with simulated annealing.

Figure 5-4 allows the means to observe the growth of fitness for different approaches at different iteration level.

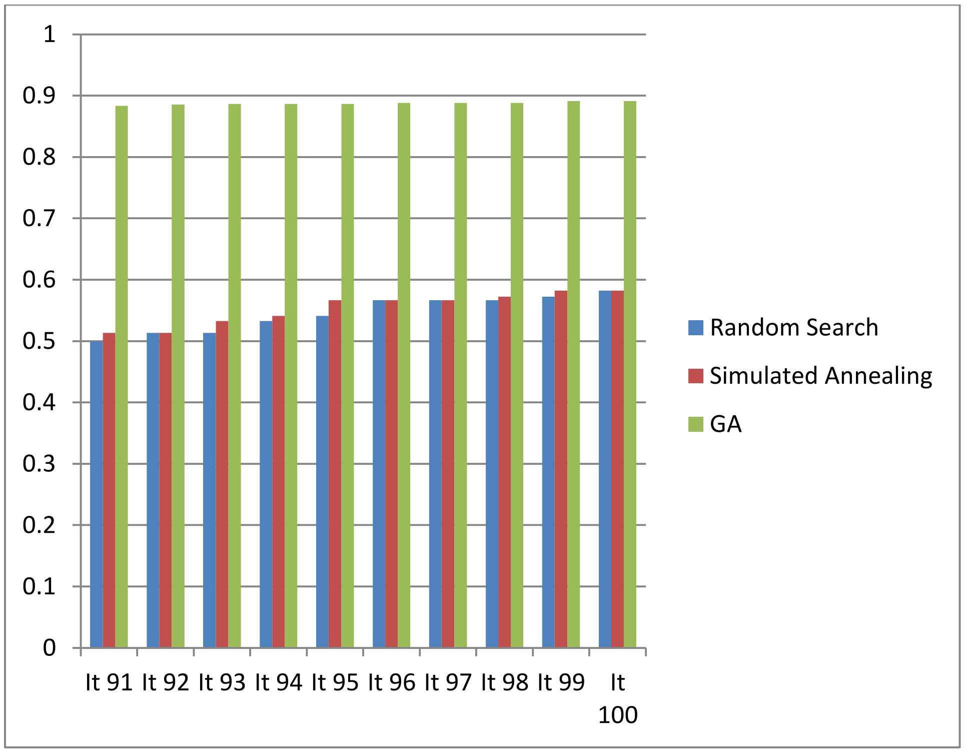


Figure 5-4: Xerces Chart Comparative Growth of Fitness

Figure 5-5 illustrates the efficiency and the effectiveness of the two different approaches. The solution generated by the GA approach is almost twice as better as the solutions produced by random search and simulated annealing.

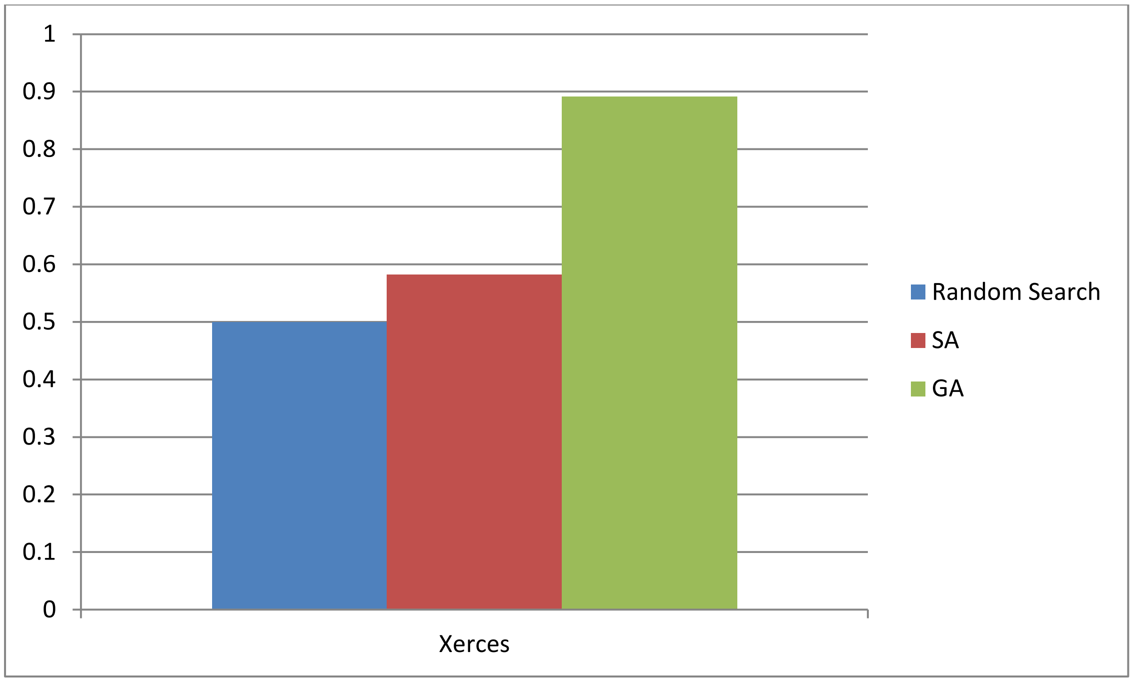


Figure 5-5: Fitness of Solutions using Different Approaches

5.5 RESULTS

Figure 5-6 shows the obtained summary quality results for each of the 6 folds, when using the search-based approach. The generated summaries are generated using a base of examples. The bases of examples are summaries used to detect and correct defects.

Both automatic and manual evaluation (AE and ME) values were high and, as expected, manual evaluation yielded better summary-quality since it considered all the correct summary possibilities and not only the specific alternatives chosen by the experts.

The average fitness function value for the process was 90%, were the lowest fitness function was 89%. The manual validation measure was much greater, with an average value of 92%; this indicates that the proposed summaries were almost as correct as the ones given by experts. The worst summary (for Xerces) had an acceptable AE of 89.9%. The detecting and correcting defects problem is mainly related to the structure of the system to evaluate. For example, some metrics (e.g., number of methods, number of relationships, etc.) could be enough to detect and correct blobs (classes that do or know too much).

To conclude, the generated summaries were in reasonable consonance with human experts. In short, the automated summarizer was able to produce summaries that appear to be similar to what a human may have produced.

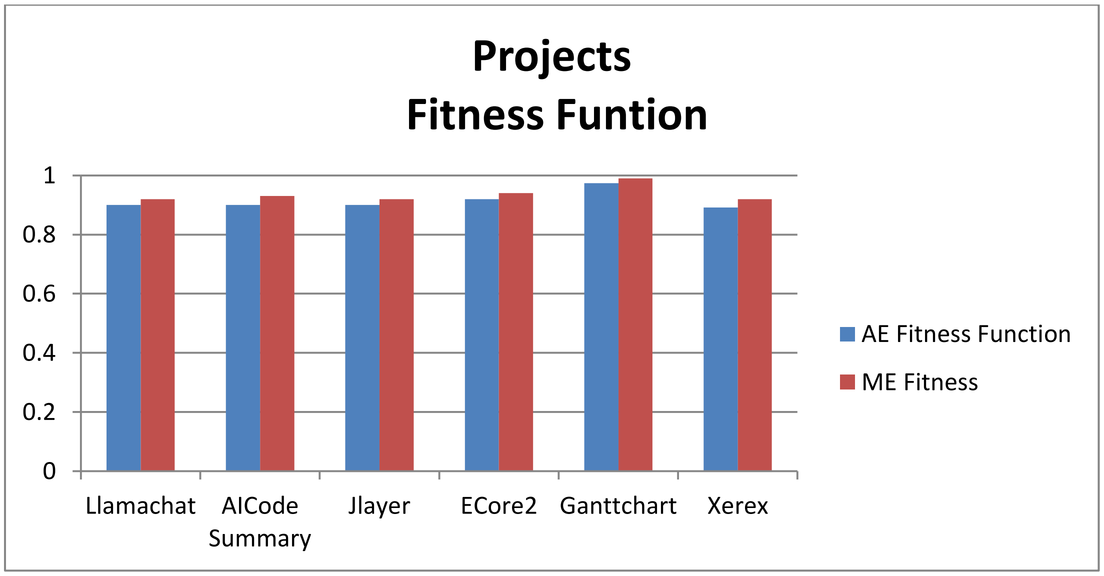


Figure 5-6: Fitness Function of 6 Projects

In Figure 5-7 the candidate solutions generated by the algorithm are compared to those generated by random search and a local search that uses simulated annealing. The local search starts with a single solution that is generated randomly. Mutation is applied on the solution for a set of iterations to produce a better candidate solution.

The GA performs almost twice as better as random search and the local search algorithm that employs SA. Heuristics search have better performance in large search spaces. The use of a good base of examples also allows the GA to converge to better candidate summaries at a much faster rate compared to a random start approach. Even under manual evaluation the GA approach outpaces the random search and simulated annealing approaches.

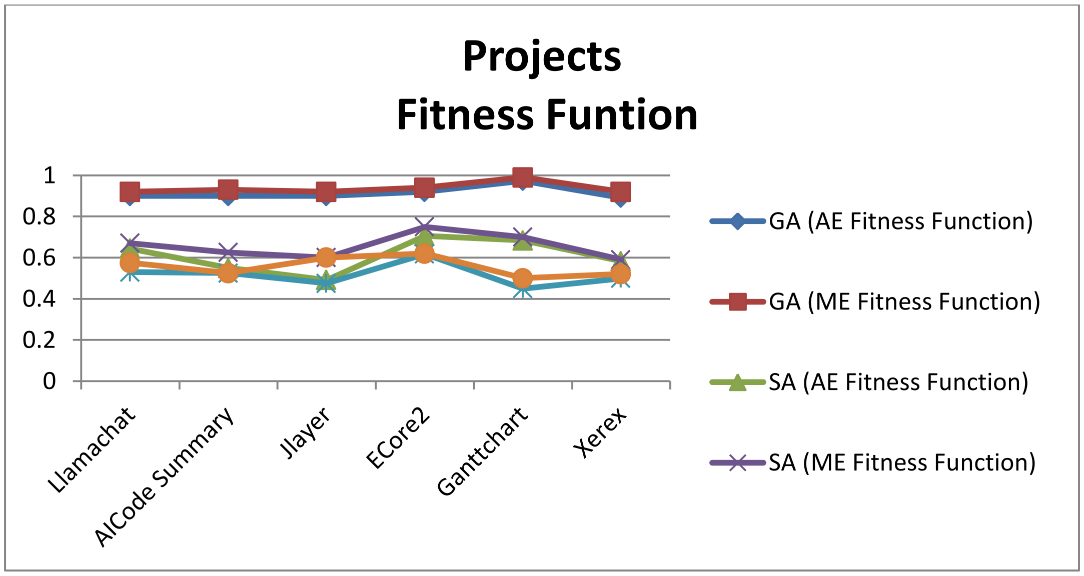


Figure 5-7: Comparative Fitness Function

5.6 DISCUSSION

An important consideration is the impact of the example base size on summary quality. Drawn for Llamachat, the results of Figure 5-8 shows that the approach also proposes good summary precisions in situations where only few examples are available. When using the technique, AE and ME seem to grow steadily and linearly with the number of examples. The precision seems to follow an exponential curve; it rapidly grows to acceptable values and then slows down. Indeed, AE and ME improved from roughly 30 to 70% as the example base size went from 1 to 8 examples. Then, it grew only by an additional 9% as the size varied from 8 to 11 examples.

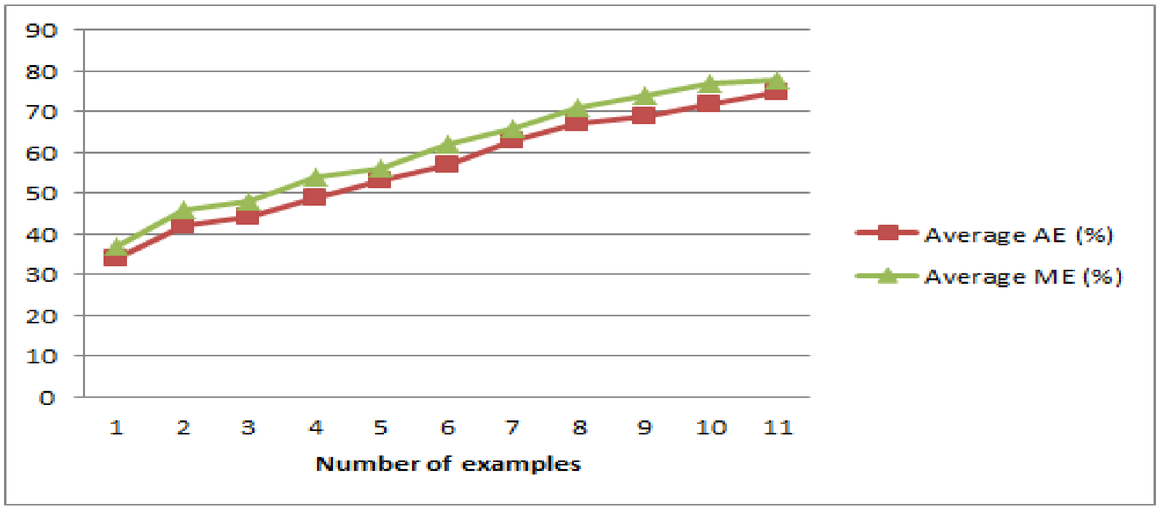


Figure 5-8: Example Size Variation

The reliability of the proposed approach requires an example set of good summaries. In the study, it was proved that by using some open source summary examples, the technique can be used out of the box and this will produce good summaries for the systems studied. In an industrial setting, companies could be expected to start with some open-source summary examples, and gradually migrate its set of good examples to include context-specific data.

Another metric was used for objective evaluation of the quality of model summaries. The metric is based on the average time required by two groups of students (having approximately the same qualifications) to performs some software engineering tasks related to the detection and correction of a specific defects (blobs and functional decomposition), and adding a new functionality to the system String Search (a new file-conversion functionality). One group performs these tasks without the use of summary and the second group used the proposed summaries. Thus, a naive approach is used by the first group that to scan through all the code elements until the ones of interest are found.

As showed in Figure 5-9, a model summary provides support for performing the software engineering tasks by presenting early on, to the programmer, code elements that are more likely to be used or analyzed, and elements that are more closely related to other code elements likely to be modified. As a result, users can locate code fragments of interest easier.

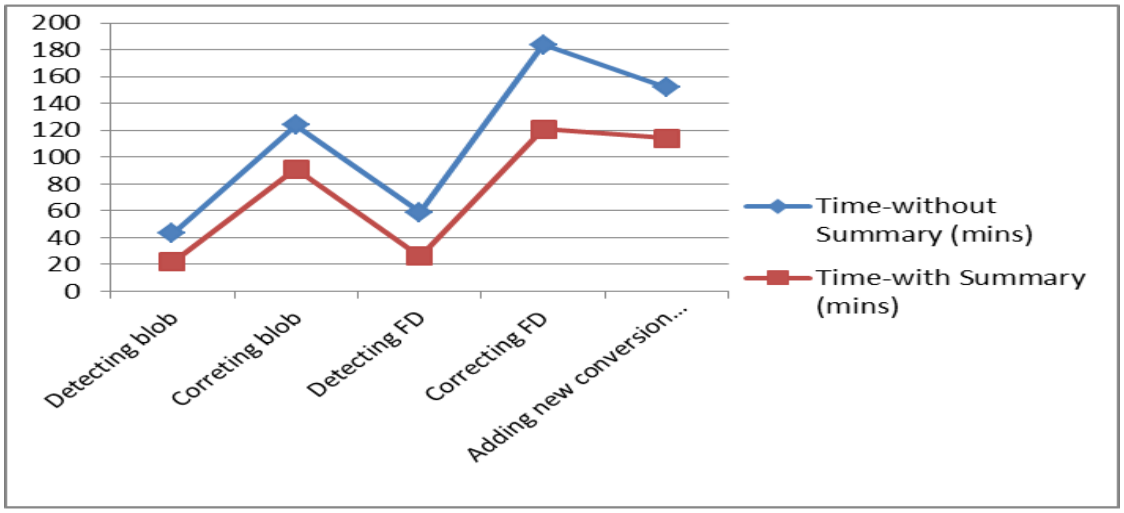


Figure 5-9: Summary Benefits

The summaries results might vary depending on the rules used, which are randomly generated, though guided by a meta-heuristic. To ensure that the results are relatively stable, the results of multiple executions for rules generation using genetic programming were compared as shown in Figure 5-10. Consequently the approach tends to be stable, since the precision scores are approximately the same for different executions.

Finally, since the summarization problem was viewed as a combinatorial problem addressed with heuristic search, it is important to contrast the results with the execution time. The algorithm was executed on a standard desktop computer (Pentium CPU running at 2GHz with 3GB of RAM). The execution time is shown in Figure 5-11. The figure reveals that greater execution times are lower than the one required for the manual process. In any case, the approach is meant to be applied to situations where manual solutions are normally not readily available.

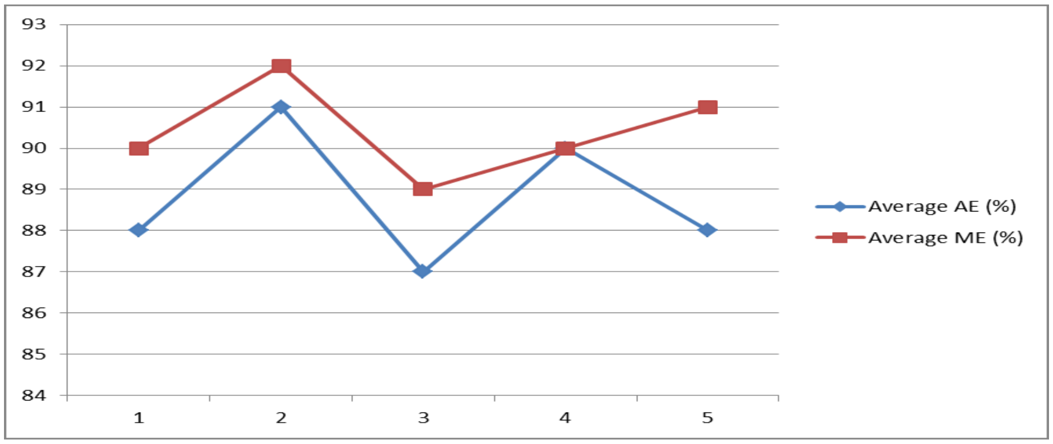


Figure 5-10: Multiple Executions

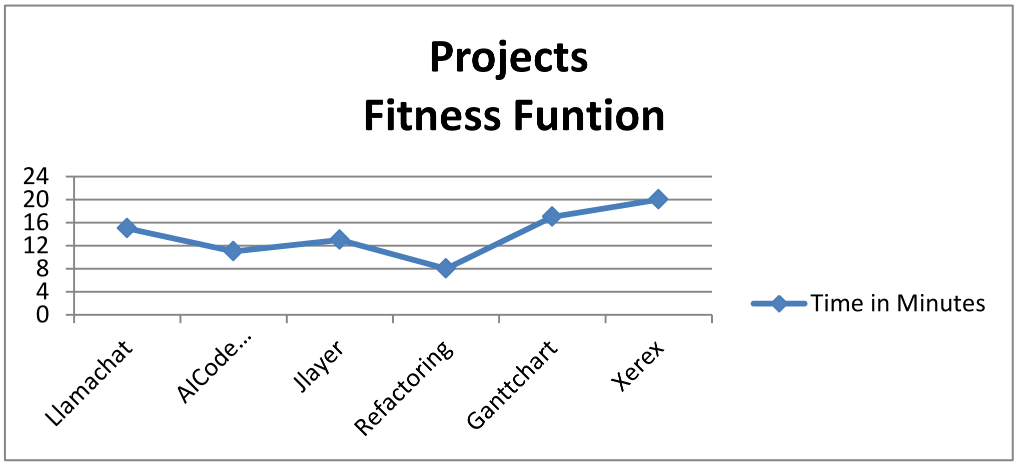


Figure 5-11: Execution Time

6. CONCLUSION

Real systems can often be extremely complex. A software engineer wishing to perform some maintenance or evolution tasks has the fastidious task of understanding and exploring the whole source code before being able to modify it. An automated solution for model summarization can help by providing an overview of the entire source code, and making it possible to explore by selecting only relevant model components.

A novel approach was proposed to automate model summarization using heuristic search. The approach uses a set of good summary examples to summarize a model of a software system. The summarization process is seen as an optimization problem where different summarization possibilities are evaluated and, for each possibility, a quality is associated depending on its conformance with both structural and lexical information.

The search space is explored using two heuristic search algorithms. Genetic programming is used to translate summary examples into metrics-based rules covering the structure of good summaries. Then, Latent Semantic Indexing (LSI) is adapted to find the important words in the source code. Finally, a summary solutions population is conglomerated based on combinations of model elements using simulated annealing algorithm. An evaluation function calculates, based on the two first steps, an average score between the two criteria's of structural information (rules coverage) and lexical information (relevant words coverage).

The proposed approach was tested on 6 open-source systems and the results are promising. In fact, an experimental assessment of the summaries, both subjectively, and objectively shows that the algorithm is able to find good summaries for a given system.

7. LIMITATIONS AND FUTURE WORK

The proposed work also has limitations. First, the approach's performance depends on the availability of summary examples, which could be difficult to collect. Second, due to the nature of the solution, i.e., an optimization technique, the summarization process can be time consuming for large systems. Finally, as heuristic algorithms are used, different execution for the same source code may lead to different summaries. Nevertheless, this is close to what happens in the real world where different experts could propose different summaries.

As part of future work, base of examples could be extended in order to take into consideration more programming contexts and other software engineering tasks to perform. In addition, another technique is being developed to reduce summary size when maximizing the quality dimension.

BIBILOGRAPHY

[1] A. Mehta and G. T. Heineman. “Evolving legacy system features into fine-grained components.” In Proceedings of the 24th International Conference on Software Engineering, pages 417–427. ACM Press, 2002.

[2] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, “Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis,” 1st ed. John Wiley and Sons, March 1998.

[3] S. Haiduc, J. Aponte, and A. Marcus, "Supporting Program Comprehension with Source Code Summarization," in 32nd ACM/IEEE International Conference on Software Engineering - NIER track, Capetown, South Africa, 2010, pp. 223-226.

[4] S. Haiduc, J. Aponte, A. Marcus and L. Moreno, "On the Use of Automated Text Summarization Techniques for Summarizing Source Code,” in the Proceedings of the 17th IEEE Working Conference on Reverese Engineering (WCRE2010), Beverly, MA, October 13-16, 2010, pp. 35-44

[5] Y. Gong and X. Liu, “Generic text summarization using relevance measure and latent semantic analysis,” in 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, , 2001, pp. 19-25.

[6] G. Salton, “Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer.” AddisonWesley, 1989.

[7] J. R. Koza. 1992. “Genetic Programming: On the Programming of Computers by Means of Natural Selection.” MIT Press, Cambridge, MA, USA.

[8] S. Kirkpatrick, C.D. Jr. Gelatt, M.P. Vecchi, “Optimization by simulated annealing.” Sciences 220(4598), 671–680, 1983.

[9] K. Sparck-Jones, "Automatic summarising: The state of the art," Information Processing and Management: An International Journal, vol. 43, pp. 1449-1481, 2007.

[10] N. Fenton and S. L. Pfleeger, “Software Metrics: A Rigorous and Practical Approach,” 2nd ed. London, UK: International Thomson Computer Press, 1997.

[11] D. E. Goldberg, “Genetic Algorithms in Search, Optimization and Machine Learning.” Boston, MA, USA: Addison-Wesley Longman Publishing Co., 1989.

[12] T. K. Landauer, P. W. Foltz, and D. Laham, "An Introduction to Latent Semantic Analysis," Discourse Processes, vol. 25, pp. 259-284, 1998.

[13] Open Source Software in Java. “http://java-source.net/.” March 2, 2013.

[14] U. Hahn, and I. Mani, “The challenges of automatic summarization.” IEEE Computer, 33, 29–36, 2000.

[15] G. Murphy, "Lightweight Structural Summarization as an Aid to Software Evolution," PhD Thesis, University of Washington, 1996.

[16] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing Software Artifacts: A Case Study of Bug Reports," in International Conference on Software Engineering, Cape Town, South Africa, 2010, pp. 505-514.

[17] S. Rastkar, "Summarizing software concerns," in International Conference on Software Engineering, Cape Town, South Africa, 2010, pp. 527-528.

[18] A. Kuhn, S. Ducasse, and T. Girba, "Semantic Clustering: Identifying Topics in Source Code," Information and Software Technology, vol. 49, pp. 230-243, 2007.

[19] D. Poshyvanyk and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," in 15th IEEE International Conference on Program Comprehension, 2007, pp. 37-46.

[20] A. Marcus, A. Sergeyev, V. Rajlich, J. Maletic, “An Information Retrieval Approach to Concept Location in Source Code" in the Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE2004), Delft, The Netherlands, November 9-12, pp. 214-223

[21] M. Harman, B.F Jones, “Search-based software engineering. Inf. Softw. Technol.” 43(14), 833–839, 2001.

[22] M. Harman, “The current state and future of search based software engineering.” In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), 20–26 May, Minneapolis, USA , 2007.

VITA

Lokesh Krishna Ravichandran was born in Chennai, Tamil Nadu, India. He graduated with his Central Board of Secondary Education high school degree from Chettinad Vidyasharam , Chennai, Tamil Nadu, India in the summer of 2007. He pursued his bachelor of engineering degree in the field of computer science at Jerusalem College of Engineering affiliated to Anna University Chennai, Tamil Nadu, India and graduated in the spring of 2011. He enrolled for the Master of Science program in Computer Science at the Missouri University of Science and Technology in the fall of 2011. He received his Master's degree in August 2013.