A Novel Approach to Optimize Clone Refactoring Activity

Salah Bouktif, Giuliano Antoniol and Ettore Merlo
Départment de Génie Informatique, École Polytechnique de Montréal
C.P. 6079, succ. Centre-ville Montréal (Québec)
H3C 3A7 Canada
{salah.bouktif,ettore.merlo}@polymtl.ca

Markus Neteler
ITC-irst Istituto Trentino Cultura
Via Sommarive 18 - 38050 Povo (Trento), Italy
neteler@itc.it

ABSTRACT

Achieving a high quality and cost-effective tests is a major concern for software buyers and sellers. Using tools and integrating techniques to carry out low cost testing are challenging topics for the testing community. In this work we contribute to alleviate the burden by proposing an architecture support for mutation and coverage criteria based testing. This architecture integrates metaheuristics to derivate effective test sets and uses automated tools to speed up testing activities. In this paper, we describe different components of the testing architecture. We focus on the developed tools performing mutation testing for JAVA code at method-level and the coverage-based testing for C code at function-level. The integration of metaheuristics in these tools is illustrated by using ant colony and tabu search to derive optimal and pruned test sets. By using our testing architecture two cases study to carry out tests respectively for JAVA methods and C-functions are presented and discussed.

Categories and Subject Descriptors

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

General Terms

Algorithms, Measurement, Experimentation, Design

Keywords

Genetic Algorithms, Multi-objective Optimization, Software quality improvement, Refactoring effort, Evolution modeling, Effort prediction.

1. INTRODUCTION

Software quality is the generic term used to represent different software facets, as perceived by different users from different perspectives. From the end-user point of view, usability, reliability, or accuracy are just a few software characteristics falling under the general software quality umbrella. Programmer perspective will likely add other facets such as source code maintainability, evolvability, documentation, and code readability or portability. Regardless of the specifically chosen perspective or facet, software quality depends on an ever changing set of characteristics. Indeed, an intrinsic property of software is its malleability, that is its ease of change and evolution. Changes and evolution characterize the entire life span of any software, from inception to retirement.

Software evolution impacts software quality in many different ways. As software evolves, very often, documentation is not updated and the source code becomes the only reliable source of information. People and technologies turnover may also contribute to create poorly-documented and gray code areas, that are essentially no longer known or maintained. Software evolution and maintenance, sometimes, are carried out in non-disciplined way. As a result, software structure, software architecture and, in general, software quality tend to deteriorate. Often, a system was originally conceived as a single platform application, with a limited number of functionalities and supported devices. Then, it evolved by adding new functionalities and was ported on new product families by adding new devices or target platforms. When writing a device driver or porting an existing application to a new processor, developers may decide to copy an entire working subsystem and to modify the code to cope with the new hardware. This approach increases the chances that programmers' work will not have any unplanned effect on the original piece of code they have just copied. However, this evolving practice promotes the appearing of duplicated code snippets, also said *clones*.

GRASS, Geographic Resources Analysis Support System, commonly referred to as GRASS GIS, is a Geographic Information System (GIS) used for data management, image processing, graphics production, spatial modeling, and visualization of many types of data. GRASS is mostly written in C; early releases date back to 1982 in other words, it is old enough to expose the above outlined evolution phenomena. GRASS underwent and is undergoing major evolutions. Initially developed under Unixes, it was also ported under MS-Windows. GRASS early releases were limited to 2-D, more recent releases added sophisticated 3-D capabilities and many other new features, that were not available in previous releases.

Recently, the GRASS development team started rationalization and redesign actions with the long term goal to improve software structure, modularity and more generally software quality. As part of this goal several key actions were identified. Old Kernighan and Ritchie (K&R) C code will be replaced by ANSI style code; code duplication, perceived as poor design, have to be minimized; coupling between functions reduced and function cohesion improved. Old undocumented gray code areas should be revised, possibly, architecture organization improved so that new developers and maintainers could pick up responsibility for that sections of code.

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.

In the literature, there are many papers proposing various methods for identifying similar code fragments or components in a software system [3, 4, 5, 17, 20, 21, 22]. However, in the best of the authors' knowledge, no effort model, no process, and no methodology to improve quality while carrying out clone refactoring has ever been presented. For a medium or large software system, quality improvement and clone refactoring are difficult and challenging tasks. A suitable order of refactoring actions have to be identified, but, refactoring deals with software architectural and design restructuring. Overall software functionalities must not be affected; effort has to be minimized while quality improvement should be maximized.

This paper proposes to adopt metaheuristic approaches, to schedule quality improvement under constraints and priority. GRASS is an open source, freely available software; development and evolution are carried out on voluntary basis. However, GRASS development is organized with weekly snapshots. At the end of each week, a new working snapshot has to be released. At the time of writing, GRASS contains about 8000 functions; out of which there are about 1000 duplicated and 800 old style K&R functions. Priority should be given to remove duplicate, large, complex, non cohesive K&R functions. Certain sub-system, has to be privileged while other code regions have not to be considered in the current phase. Refactoring actions should be organized into work-packages (WP), so that each WP can be carried out leaving GRASS weekly release in a consistent and fully functional status. Sub-sets of GRASS functions, including the mentioned about 1000 clones, need to be organized into WPs; any WP, subset of functions, constitutes a solution to our problem. In the essence the solution space is huge, for example, searching a subset of clones to be removed means identifying a subset of the 1000 duplicated functions. Unfortunately, there are about 2^{1000} possible subsets. Furthermore, constraints on effort, pressure to increase quality in a shortest possible time frame and priorities, as expressed by developers, have also to be dealt with.

It is well known that in several fields, such as graph theory, artificial intelligence, and pattern recognition, heuristics and opportunistic strategies often allow to reduce the average case complexity when linear solution exists for specific sub-problems. Indeed, the exact solution may not be computational feasible, whereas approximated solutions with low complexity or almost linear complexity may ensure scalability at the price of slightly reducing the result optimality. Given the above outlined constraints we re-formulated the quality schedule improvement, under constraint and priority, as an M-dimensional Knapsack problem. The problem was solved via Genetic Algorithms (GAs) and quality improvement was effectively scheduled by identifying near optimal clone removal actions, set of functions organized into WP. At each week several suitable WP were identified and proposed to GRASS developers; each WP was consistent with available effort and GRASS weekly release strategy while maximizing quality improvement.

The main contribution and novelty of this paper are as follows:

- we propose to solve our schedule quality improvement under constraints and priority as a multi-constrained Knapsack problem;
- we model our clone refactoring actions in term of the schedule quality improvement problem; and
- we propose a novel model to estimate the effort needed for clone refactoring.

The remainder of the paper is organized as follows: Section 2 summarizes previous work; Section 3 presents our problem formulation; the approach i.e., the GA solution to the quality improving problem is presented in Section 4 while GRASS case study is detailed in Section 5, Section 6 draws conclusions and outline future work.

2. RELATED WORK

An application of search–based techniques to project scheduling was done by Davis [10]. A survey of the application of GAs to solve scheduling problems has been presented by Hart et al. in [16]. The mathematical problem encountered is, as described by the author, the classical, NP-hard, bin packing or shop-bag problem. A survey of approximated approaches for the bin packing problem is presented in [9]. More recently Falkenauer published a book devoted to the GA and grouping problems [11]. The theme of the book and the proposed genome encoding are highly relevant to the problem addressed in this paper. Schema interpretation and bin packing genome encoding described in the book were a source of inspiration, although our problem is slightly different. Our problem requires to consider both the work that can be done and, at the same time, the highest possible quality improvement. Therefore, the order on which WPs are presented to the teams is relevant, whereas bin packing doesn't impose a packing order to reach an optimum.

Search heuristics have been applied in the past to solve some related software project management problems. In particular, Kirsopp et al. [19] reported a comparison of random search, hill climbing, and forward sequential selection to select the optimal set of project attributes to use in a search–based approach to estimating project effort.

A comparison of approaches (both analytical and evolutionary) for prioritizing software requirements is proposed in [18], while Greer and Ruhe proposed a GA-based approach for planning software releases [15]. Clark et al. [8] argued a great potential for the exploitation of metaheuristics within software engineering, particularly, in the areas of test data generation, module clustering and cost/effort prediction.

Commonalities can also be found with the work of Antoniol et al. [1]. This work focuses on problem of staffing a software maintenance project using queuing networks and discrete-event simulation. Given an (ordered) distribution of incoming maintenance requests, the goal of Antoniol et al. was to determine the staffing levels for each team.

The present paper has some substantial differences from previous works. We propose an effort model for clone detection refactoring and we incorporate constraints and priorities into our refactoring problem formulation.

3. PROBLEM FORMULATION

Applications based on GAs revealed their effectiveness in finding approximate solutions, when the search space is large or complex, when mathematical analysis or traditional methods are not available, and, in general, when the problem to be solved is NP-complete or NP-hard [13]. Roughly speaking, a GA may be defined as an iterative procedure that searches for the best solution of a given problem among a population, represented by a finite string of symbols, the *genome*. The search is made starting from an initial and often randomly generated population of individuals. At each evolutionary step, individuals are evaluated using a *fitness function*. High-fitness individuals will have the highest probability to reproduce themselves.

The evolution (i.e., the generation of a new population) is made by means of two kinds of operator: the *crossover operator* and the *mutation operator*. The crossover operator takes two individu-

als (the *parents*) of the old generation and exchanges parts of their genomes, producing one or more new individuals (the *offspring*). The mutation operator has been introduced to prevent convergence to local optima; it randomly modifies an individual's genome, for example, by flipping some of its bits, if the genome is represented by a bit string. Crossover and mutation are respectively performed on each individual of the population with probability *pcross* and *pmut* respectively, where $pmut \ll pcross$.

GAs are not guaranteed to converge. The termination condition is often based on a maximum number of generations, on a given value of the fitness function or, also, on a maximum number of generations, without fitness improvement.

In this paper we focus on a general problem of how to promote software quality improvement, while performing evolution and maintenance activities. More precisely, while performing perfective maintenance actions, we would like to select candidate functions to maximize the quality gain under the constraint of a limited amount of available resources i.e., effort.

3.1 Constrained Knapsack Problem

Let $\mathcal{S}$ be a software system and $\mathcal{F}$ a set of n *problematic* functions in $\mathcal{S}$ (e.g., set of duplicated functions). Each function in $\mathcal{F}$ is described by a set of metrics (e.g., size, cyclomatic complexity, cohesion, coupling) and a level of priority. The higher the priority the higher will be the probability that the function is selected and actually refactored.

The goal is to find the most *valuable* subset of functions in $\mathcal{F}$; a subset that when refactored the maximum quality gain is obtained, while it consumes the minimum of resources. More precisely, the goal is to the maximize the quality improvement, in a given period of time, with an effort not exceeding the maximum amount of available resources. Given this general goal, our problem can be thought of as a *Constrained Knapsack Problem* (KP) [25], which can be standardly formulated as follows:

$$maximize \;\; f(x_1, \ldots, x_n) = \sum_{j=1}^{n} p_j x_j, \tag{1}$$

$$subject\ to \;\; Ct : \sum_{j=1}^{n} w_j x_j \leq b, \tag{2}$$

$$x_j \in \{0,1\};\; j = 1, \ldots, n;\; p_j > 0; w_j > 0 \text{ and } b > 0$$

In this formulation, the KP items are the *problematic* functions ($j = 1, \ldots, n$), the item profit p_j is the estimated quality gain, when the j^{th} function is refactored, and the item weight w_j corresponds to the effort needed to carry out source code modification. The objective function $f(x_1, \ldots, x_n)$ is expressed as the 0-1 weighted sum of profits; each variable $x_j \in \{0, 1\}$ decides if the j^{th} function is included into the subset of functions to be refactored. This is a maximization problem under the constraint C_t called *Effort constraint*; this constraint states that the effort needed should not exceed a maximum amount E_{max}.

Since we are particularly interested in the problem of duplicate code removal, i.e., clone elimination, a second constraint must be considered. We call it *Grouping constraint*. Functions are grouped into clusters according to some distance measures. Each cluster C_k contains functions, i.e., clones c_j, with relative distance below a given threshold. For a threshold value of zero, clusters will contain functions that are exact copies. Details on clone detection and function clustering are beyond the scope of this paper, more details can be found in [2, 17, 20, 21, 22, 23]. The *Grouping constraint* states that if a clone c_j in a cluster C_k belongs to the maximization problem solution, then all the functions in C_k should be in the same solution. This corresponds to an underlying assumption that refactoring effort is composed by two main parts. A first activity is needed to study the system, clones in the cluster, interactions between calling functions and clones, and the overall software architecture. Once performed this program understanding task, the actual incremental cost to remove a single clone of the same cluster is only a fraction of the overall cost. Thus it pays off to get rid of all clones in a cluster, if just one single function is selected. Note that after refactoring, only one function will surrogate all the cluster clones.

With respect to the quality improvement, we are interested in the modularity of the design, which is often expressed in term of coupling and cohesion measures. Our strategy tries to select as refactoring candidates, clones with weak cohesion and strong coupling (low modularity). In other words, since clones with low cohesion and high coupling will be eliminated, the overall quality of $\mathcal{S}$ will be improved, because we will improved the quality of $\mathcal{F}$, which are subset of functions composing the system. Consequently, the problem is a multi-objective problem with two objective functions. While the total cohesion of the selected clones is minimized (or total *lack* of cohesion maximized), the total coupling of them must be maximized.

In addition to the quality improvement of the system $\mathcal{S}$, we have also to consider, for each function, a level of refactoring priority expressed by expert' desires. For example, as a priority, the experts of GRASS maintenance suggested to get rid of the old Kernighan and Ritchie style code, of large functions, and of functions with high cyclomatic complexity. Expert opinion was modeled by a third objective function to be maximized, which would measure the total priority of problem solution.

Overall, the above consideration motivate why our Knapsack problem is in reality Multi-constrained (MKP) and Multi-objective (MOKP). A problem with only one of these two qualifications is known to be an NP-complete combinatorial optimization problem. Our tailored problem formulation is the following:

$$\max_{x_1,\ldots,x_n} \; f1(x_1, \ldots, x_n) = \max_{x_1,\ldots,x_n} \sum_{j=1}^{n} Lcoh(c_j) x_j, \tag{3}$$

$$\max_{x_1,\ldots,x_n} \; f2(x_1, \ldots, x_n) = \max_{x_1,\ldots,x_n} \sum_{j=1}^{n} Coup(c_j) x_j, \tag{4}$$

$$\max_{x_1,\ldots,x_n} \; f3(x_1, \ldots, x_n) = \max_{x_1,\ldots,x_n} \sum_{j=1}^{n} Prio(c_j) x_j, \tag{5}$$

$$subject\ to \quad C_{t1} : \sum_{j=1}^{n} e(c_j) x_j \leq E_{max}, \tag{6}$$

$$C_{t2} : c_j \in C_k | x_j = 1 \Rightarrow \forall c_i \in C_k : x_i = 1 \tag{7}$$

In the above equations, the function $Lcoh(c_j)$ is a measure of a lack of cohesion within a clone c_j; $Coup(c_j)$ measures the in and out coupling between a clone c_j and the other functions of $\mathcal{S}$; $Prio(c_j)$ is a priority refactoring level of a clone c_j; $e(c_j)$ is the effort required to refactor c_j and E_{max} is the maximum amount of effort provided by the available resources for the refactoring activity. In the best of authors' knowledge, no previous contribution published an effort model for clone refactoring problems. In the following paragraphs, we will introduce our novel effort model.

As a first level of approximation, we distinguish between two types of clones:

1. *Perfect clones*: clones which are syntactically and semantically identical; they are pure replicas of the same lines of code, but indentation; and

2. *Near-duplicated* clones: clones with some different tokens; they are, for example, code fragments in which a variable has consistently been changed throughout the code.

3.2 Clone Refactoring Effort Model

The effort required to refactor clones is modeled by the four following components: system understanding, clone modification, called context understanding, and calling context adaptation effort.

System understanding effort takes into account the time needed to carry out basic program comprehension actions such as locating the clone, studying software architecture, and deciding which clones to eliminate. We represent this effort as a *system* level effort, which is denoted e_s and which is defined as the average effort required before starting the actual refactoring actions.

Clone modification effort represents the effort needed to modify the necessary tokens in the source code. This term is formulated as:

$$e_0 \cdot DToken(c_j)$$

where e_0 is an average estimated effort for one generic source code change and $DToken(c_j)$ is the number of different tokens in c_j to be changed.

Called context understanding effort expresses the effort required to understand the contexts of the functions called by the clone c_j and, possibly, the code of functions that can be reached from the clone via *caller-callee* relation. It is a function of the size of clone c_j ($LOC(c_j)$) and of the size of the *reached* code $RLOC(c_j)$ from c_j. $RLOC(c_j)$ is the overall size measured as LOC of the functions reached from the clone via *caller-callee* relation. This term is formulated as:

$$\mu \cdot e_0 \cdot \ (LOC(c_j) + RLOC(c_j))$$

We assume that the estimated average effort to understand one line of code be μ times higher than the effort required for one generic source code change.

Calling context adaptation effort represents the effort required to adapt the calling contexts of the modified clones. Adaptation may involve, for example, include files, makefile, or call sites. We assume that the calling context adaptation effort be proportional to the number of functions directly calling the clones c_j and that one generic source code change is performed per calling function. This term is formulated as:

$$e_0 \cdot Calling(c_j)$$

where $Calling(c_j)$ is the number of calling functions.

Therefore, the overall effort $e(c_j)$ required to refactor a clone c_j is expressed as:

$$\begin{cases} e_s + e_0 \cdot Calling(c_j) & \text{if } c_j \text{ is a perfect Clone} \\ e_s + e_0 \cdot (DToken(c_j) + Calling(c_j)) & \\ \qquad + \mu \cdot e_0 \cdot (LOC(c_j) + RLOC(c_j)) & \text{otherwise} \end{cases} \tag{8}$$

As a general rule, any predictive has to be calibrated, i.e., model parameters have to be adjusted to the given organization, process, domain, and so forth. Clearly, to adapt our model to a particular context or a given organization, three parameters must be estimated: effort e_s, e_0 and the multiplicative coefficient μ.

4. A GA SOLUTION TO THE QUALITY IMPROVING PROBLEM

The application of search-based techniques needs a preliminary step in which we define some key ingredients. In particular, to design a genetic algorithm for any specific problem, the following elements are needed:

- a chromosome representation to describe the solutions;
- mutation and crossover operators; and
- a fitness function to guide the search.

Since, the quality schedule problem was reduced to a Knapsack problem (Section 3), in the following paragraphs we will describe how we customize the Knapsack problem i.e., chromosome representation, genetic operator implementation, initial population construction as well as fitness function and constraints implementation.

4.1 Clone Knapsack representation

A simple way to encode our Knapsack problem is to represent the inclusion or exclusion of each of the n *problematic* functions by a bit in a bit string of length n. The j^{th} bit contains the value of the decision variable x_j. Figure 1 illustrates the binary representation of the Knapsack chromosome.

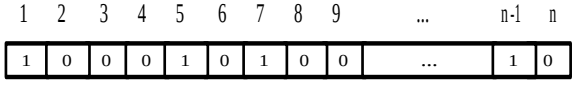

Figure 1: Binary representation of the clone Knapsack solution.

A chromosome encoding may represent an infeasible solution. An infeasible solution is a solution for which at least one of the problem constraints is violated. In our case, an infeasible solution is encountered when the total effort needed exceeds the maximum effort or when not all functions of a clone cluster are included in the Knapsack. According to the literature, handling violations can be done in the following ways:

- by using a representation that automatically preserves solution feasibility [6];
- by applying a penalty method to penalize the fitness of any infeasible solution [14]; and
- by defining a repair operator that transforms infeasible solutions into feasible ones [7].

Clone grouping constraint imposes that if a clone c_j belonging to a cluster C_k is selected then, all the functions contained in C_k must be in Knapsack too. The easiest way to handle this constraint is to use a different bit string encoding in which a bit will represent the inclusion or exclusion of an entire cluster out of the N_c clusters. This representation (N_c-bit string) will be used during the search process in particular by the genetic operators. Solutions are then mapped back into the initial representation (n-bit string) via straightforward one-to-one mapping.

With respect to the effort constraint, a Knapsack solution that needs more effort than E_{max} has to be transformed to lower the

needed resources. This strategy is more recommended in the literature than the approaches based on penalty method, specially when the optimization problem is multi-objective. Of course, a repair operator is needed to transform infeasible Knapsack into feasible ones (see Section 4.3).

4.2 Crossover and mutation operators

Since the chromosome representation that we adopted for the Knapsack problem is a bit string, several choices are possible to perform the crossover and the mutation. We adopted two ways to perform crossover. The first is the standard crossover with two cut points: two parent chromosomes (P_1 and P_2) are selected according to their fitness function values. Two substrings (SS_1 and SS_2), having the same length and beginning at the same position, are extracted respectively, from P_1 and P_2. Then SS_1 takes the place of SS_2 in P_2 and vice versa for SS_2 to obtain two offspring O_1 and O_2.

The crossover operator adopted is the uniform crossover in which two parent chromosomes give birth to a single offspring. Each bit of the offspring is copied from one of the parent. For each offspring position, a binary random number is generated. If this number is 0, the j^{th} bit of the offspring is a copy of the j^{th} bit in the first parent. If it is equal to 1 the bit is copied from the second parent.

Once the offspring are generated by the standard or the uniform crossover, a mutation operator is applied. The mutation consists in complementing a small number of bits equivalent in size to roughly 2 or 3% of the string length, by changing them from 0 to 1 and vice versa.

4.3 Repair operator

This operator is mainly needed to ensure feasible solution under the effort constraint. The offspring generated by the crossover and mutation may violate the effort constraint. In order to make all the offspring feasible, we developed a greedy algorithm inspired by the works of Raidl [25] and of Chu and Beasley [7].

```
REPAIRKNAPSACK(S)
 1  Arrange S by respecting an ascending order of qp_k ratios
 2  E ← Σ_{k=1}^{N_c} e(C_k)S[k]
 3  j ← 0                    ▷ Considering the lowest qp_k
 4  while (E > MaxE and k < N_c) do
 5     if (S[k] == 1) then
 6        S[k] ← 0
 7        E ← E − e(C_k)
 8     end if
 9     k ← k + 1
10     end do
11  k ← N_c                  ▷ Considering the highest qp_k
12  while (E + e(C_k)S[k] < MaxE and k > N_c) do
13     if (S[k] == 0) then
14        S[k] ← 1
15        E ← E + e(C_k)
16     end if
17     k ← k − 1
18     end do
19  SFEASIBLE ← S
20  return SFEASIBLE
```

Figure 2: Summary of Repair operator algorithm.

The general idea is to use the notion of pseudo-utility ratios for the Knapsack. Since we are handling only one constraint in this algorithm, a pseudo-utility ratio for a clone cluster C_k is defined as $qp_k = \frac{q(C_k)}{e(C_k)}$, where $q(C_k)$ is quality gain associated to C_k and $e(C_k)$ is the effort needed to eliminate the clones in C_k. We call this ratio *quality-price ratio*. Once the quality-price ratios are calculated for each variable x_j, the repair process consists of two steps. The first step checks the decision variables in ascending order of qp_k's and mutates them from 1 to 0, if the feasibility is still violated. As result, the variables with lowest quality-price ratio are being considered first and the corresponding clone clusters are removed from the Knapsack. The second step examines the the decision variables in descending order of qp_k's and mutates them from 0 to 1 as long as the feasibility is not violated. As result, the variables with highest quality-price ratio are being considered first and the corresponding clone clusters are added to the Knapsack while the feasibility is not violated. Figure 2 summarizes the repair operator.

4.4 Initial Population

Two objectives have to be pursued when constructing the initial population: individuals must be diversified and must represent feasible solutions. Using the bit string representation, the inclusion and the exclusion (decision variable x_k) of a randomly selected clone cluster C_k is decided in the following way. First, a binary number is randomly generated. If the number is equal to one, the selected clone cluster is added to the solution, if this one remains feasible. This process continues until no clone cluster can be added without violating E_{max}. The construction of the initial population is shown in the algorithm of Figure 3.

```
INITIALPOPULATION(P_Size)
 1  for k ← 1 to N_c do S[k] ← 0 end do
 2  E ← 0
 3  for k ← 1 to P_Size do
 4     S_k ← S
 5     R ← k          ▷ R is a set of non examined ranks
 6     select randomly j from R
 7     R ← R − {k}
 8     while (E + e(C_k) < MaxE) do
 9        S_k[k] ← 1
10        E ← E + e(C_k)
11        select randomly k from R
12        R ← R − {k}
13     end do
14  end do
```

Figure 3: Algorithm of Initial Population Construction, P_{Size} is its size .

4.5 Fitness function and selection methods

In single-objective optimization problems using a GA fitness function and objective function are often identical. However in Multi-objective Optimization Problems (MOP), fitness assignment and consequently the selection operation must take into account multi-criteria of optimization. Basically, three strategies dealing with MOP resolution are reported and used in the literature [27]: Aggregation-based, criterion-based, and Pareto-based strategies. These three strategies propose different approaches of fitness assignment and individual selection. In the remainder of this section, we summarize the three strategies devoting more details to the one adopted in this paper: the aggregation-based strategy.

Aggregation-based strategy central idea is to transform MOP into a single-objective problem. The different objective functions f_i of the problem are combined into a single function F with an affine transformation:

$$F(x) = \sum_{i=1}^{T} \lambda_i f_i(x),$$

where $\mathbf{x}$ is a solution, the weights $\lambda_i \in [0,1]$ and $\sum_{i=1}^{T} \lambda_i = 1$ and T is the number ob objectives of the MOP. The derived solutions by linear aggregation method for the MOP strongly depend on the choice of the λ vector. The weights λ_i are recommended to be chosen according to some preferences associated to the objectives [27]; λ_i selection is a trial-and-error process. Thus, in general, the problem is solved several times with different λ vectors and the best values are chosen using some evaluation criteria such as, for example, speed of convergence, sub-optimality of solution, and so forth. If the different objective are not commensurable, they can be brought in the same range by the following equation:

$$F(x) = \sum_{i=1}^{T} \beta_i \lambda_i f_i(x), \tag{9}$$

where β_i are constants initialized generally to $\frac{1}{f_i(x^*)}$ and $f_i(x^*)$ is the optimal solution according to the i^{th} objective. Several blind strategies are used to generate randomly the weights λ_i as follows:

$$\lambda_i = \frac{random_i}{random_1 + \ldots + random_T.} \tag{10}$$

Using the aggregation-based method, the fitness function of our GA for MKP is defined by the equation 11.

$$\begin{aligned} F(x_1, \ldots, x_n) &= \beta_1 \lambda_1 \sum_{j=1}^{n} Lcoh(c_j) x_j \\ &+ \beta_2 \lambda_2 \sum_{j=1}^{n} Coup(c_j) x_j \\ &+ \beta_3 \lambda_3 \sum_{j=1}^{n} Prio(c_j) x_j \end{aligned} \tag{11}$$

where λ_1, λ_2 and λ_3 are generated by applying three times equation 10, with:

$$T = 3$$

$$\beta_1 = \frac{1}{\sum_{j=1}^{n} Lcoh(c_j)}$$

$$\beta_2 = \frac{1}{\sum_{j=1}^{n} Coup(c_j)} \tag{12}$$

$$\beta_3 = \frac{1}{\sum_{j=1}^{n} Prio(c_j)}$$

Here, the denominators take the respective values of the objective functions when $x_j = 1 \forall j = 1, \ldots, n$.

With the aggregation based method, individual selection is performed via methods like the *Roulette Wheel Selection*, the *Rank Selection*, or the *Steady-State Selection* [12]. Having no particular reason to prefer one approach over the other, in this paper, individual selection was performed by applying the *Roulette Wheel Selection* method.

Criterion-based methods are founded on the idea of switching between the objectives, during the selection phase. Each selection of an individual is potentially based on a different objective. The set of selected individuals contains in our case three sub-sets, portions or fractions. Each sub-set contains the "fittest" individuals, according to a different objective function. Some researchers, for example, propose to constitute equal portions [26].

Pareto-based methods are based on computing the fitness values by using Pareto dominance [14]. The idea is to exploit the partial order on the population. The most known way of such fitness calculating are referred as ranking methods like Non-dominated Sorting Genetic Algorithm (NSGA), Non Dominated Sorting (NDS), and Weighted Average Ranking (WAR). For example, NSGA, proposed initially by Goldberg [14], assigns the rank 1 to all the non dominated individuals. In a maximization problem, a solution (individual) s_1 dominates a solution s_2 *iff* $\forall i \in [1..T] f_i(s1) \geq f_i(s2)$ *and* $\exists i \in [1..T] | f_i(s1) > f_i(s2)$. Then, these individuals are removed from the population; the next new non dominated individuals are identified and the rank 2 is assigned to them. The process continues until all the population individuals are ranked. Afterward, the probability of an individual to be selected is computed on basis of its rank. More detail about the others methods can be found in [27].

5. CASE STUDY

As mentioned in the introduction, refactoring has been performed to *GRASS*, which is a large open source GIS. In particular, the *GRASS* 6.1-CVS development snapshot of December 5, 2005[1] was used as a case study. Its characteristics are summarized in Table 1. *GRASS* modules correspond to applications and represent commands. Applications are organized by name, based on their functionality group such as map display, general file operations, image processing, raster, vector operations, etc. The first letter of a module name refers to a functionality group and is followed by one dot and one or two other dot-separated words, which describe specific tasks. All *GRASS* modules are linked with an internal parser. If there are no command-line arguments entered by a user, the parser calls a Tcl/TK based graphical user interface, for an interactive version of a command. Otherwise, it will start the command-line version. Code parameters and flags are defined within each module. They are used to ask user to define map names and other options. *GRASS* provides an ANSI C language API with nearly one thousand of GIS functions, which are used by *GRASS* modules to read and write maps, to compute areas and distances for georeferenced data and to visualize attributes and maps. Details of *GRASS* programming are covered in [24].

Directories	541
C Files	2486
Header Files	591
C KLOC	510
Libraries	45
Applications	580
Functions	8054
K&R Functions	820
Perfect Clones	274
Near Duplicated Clones	720

Table 1: GRASS 6.1-CVS key characteristics (Dec 5, 2005).

[1]Downloadable from `http://grass.itc.it`

This programming API is organized in about forty libraries, and higher priority was given to eliminate old style K&R functions and to refactor clones. Details on number of function, number of K&R functions and clones are summarized in Table 1.

The process to determine model constants was perceived as crucial to produce a meaningful quality improvement planning. Moreover, although GRASS quality improvement is priority, the reliable organization of refactoring activity into WPs compatible with the current GRASS development practice was perceived as essential to obtain programmers' cooperation and support. Perfect clones are a special case of near duplicated clones and they have been the target of model calibration, since the effort to refactor them is lower than that needed to refactor near duplicated clones.

Model tuning was thus organized into the following sub-steps:

- Initial subjective model constant determination;
- Qualitative result assessment; and
- Preliminary constant recalibration.

Initial subjective constant determination was carried out via a consensus based approach. Three software engineers knowledgeable of refactoring, and C programming idioms were asked to reach a consensus on e_s and e_0 values.

Once a consensus was reached, a qualitative assessment was done. This step aimed at verifying that no major error in constant value guessing was made. Values were used in the effort model to produce via GA two sets of ten WPs. For the first set E_{max} was fixed at one hour and thus the overall estimated refactoring effort for each WP was of about one hour. The second set contained, the same number of WPs, but for groups of clones requiring a refactoring activity of about two hours. Two GRASS developers were asked to qualitatively assess the WPs. More precisely they were asked to judge, in their best knowledge, how long it would have been taken to refactor one randomly chosen WP in each of the two sets. They were also asked to quantify the average expected ratio of the time requires in the two cases. GRASS developers were neither aware of the effort model, nor of the imposed different E_{max} values. Qualitative results substantially confirmed e_s and e_0 initial values.

GRASS developers were asked to provide guidelines to set up GA priority constraints. Discussion lead to the decision to separate the problem of K&R code *ansification* from the more long term quality improvement goal. Unfortunately public domain *ansification* tools don't provide an accurate code transformation. Thus we developed our own *ansification* toolkit; information collected while parsing C code on K&R function location is very accurate and it was used to perform precise (including saving comment position) and semantic preserving code transformation. The overall time required in the process was of about six hours of an expert in parsing and code transformation and eight hours of an expert GRASS programmer. Out of the six hours, about five were needed to develop the *ansification* script. *Ansification* was performed on a per-directory basis; GRASS developer time was needed to semi-automatically verify performed transformations, to recompile the system, perform basic tests, and commit changes in the central CVS repository. In summary, all 820 K&R functions were translated into ANSI style in less than two working days. At the time of writing, no error has been discovered related to such a major change. It is however worth noticing the relation with quality issues: in K&R style function parameter declaration, at function definition, is optional. Indeed out of the 820 functions about 20 had undeclared parameters; most of these dangerous missing declarations were unknown to developers. The type of these parameters is assumed by the compiler, however, there is no guarantee on assumptions made by different compilers, thus the simple rebuild of an application with different compiler may lead crashes or, in the worst scenario, to undetected data corruption.

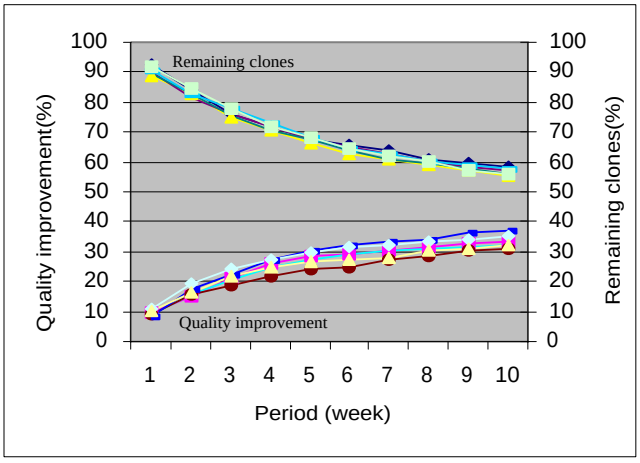

Figure 4: Planned quality improvement.

GRASS developers set up priority on a code region perceived as critical. Moreover, preference was expressed to start with the largest and more complex function. Complexity was in turn represented as number of parameter and cyclomatic complexity. These new priority values together with the selected model parameter values were then used exactly as above to produce two new sets of WPs different in term of involved functions from the previous one. The same GRASS developers were asked to randomly select one WP and to perform the actual refactoring, while recording effort data.

As already underlined, effort data for the GRASS case study are needed to ensure that the refactoring effort will be compatible with the GRASS development process and will not interfere with GRASS planned milestones. Empirical observation demonstrated an optimist attitude of people involved in the project. Guessed data were not substantially different. Nevertheless, reported data for the two programmers, were about 25 % higher. Two WPs of one hour requires about 75 and 76 minutes respectively; out of this time in both cases about 15 minutes were needed to initially study the task. Collected data were fed into a recalibration algorithm to update e_s and e_0 estimates. Effort to refactor near duplicated clones was assumed as being three fold higher than the effort required for perfect clones, i.e. μ value was set to three. With these estimated values, simulations to schedule and study quality improvement were carried out on a ten weeks temporal horizon. Figure 4 reports data obtained out of five simulations. Data points are quite closed together so that it is difficult to distinguish one from the other. Notice also that there are 274 perfect clones in the system. Therefore, under the assumption of an available effort corresponding to one GRASS developer working full time on refactoring, i.e., E_{max} equal to 40 hours, the first month of activity is substantially devoted to removing most of these clones.

Clearly, new data and new empirical evidence are needed to verify our conjecture that μ has an approximate value of three. New data will also help us to better calibrate the model and to quantify model accuracy. Our preliminary results are very encouraging. However it pays to be cautious in that only a few data points were available to calibrate the model. Furthermore, the open source na-

ture of the GRASS project and the GRASS programmers' motivations and skills may or may not be similar to other open source projects or to other industrial development environments.

6. CONCLUSION

We have presented how a general problem to schedule quality improvement under constraint and priority can be solved via metaheuristic search. We reported how the problem can be instantiated into an approach to schedule refactoring actions aiming at removing duplicated code under constraints and priorities.

By working in tight contact with the GRASS development team, we were able to obtain a first tuning of the effort model parameters and, thus, to effectively provide people with WPs, which are sets of functions to be refactored, that maximized quality improvement under imposed constraints. WPs were identified via GAs. Presently work is undergoing to apply refactoring actions to the GRASS system. Priority in this early phase was given to eliminate K&R code and large non cohesive exactly duplicated functions. In the paper, simulation was carried out and reported to better understand the time frame under which clone removal will take place.

Future work will be devoted to better calibrate constant parameters of the effort model, to incorporate library structure into the quality schedule problem, and to provide a detailed taxonomy of clones so that semi-automatic clone removal will be feasible.

7. ACKNOWLEDGMENTS

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (Research Chair in Software Evolution #950-202658).

8. REFERENCES

[1] G. Antoniol, A. Cimitile, G. A. Di Lucca, and M. Di Penta. Assessing staffing needs for a software maintenance project through queuing simulation. *IEEE Transactions on Software Engineering*, 30(1):43–58, Jan 2004.

[2] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44:755–765, October 2002.

[3] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of IEEE Working Conference on Reverse Engineering*, July 1995.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 368–377, 1998.

[5] E. Buss, R. D. Mori, W. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. Muller, J. M. S. Paul, A. Prakash, M. Stanley, S. Tilley, J. Troster, and K. Wong. Investigating reverse engineeering technologies for the CAS program understanding project. *IBM Systems Journal*, 33(3):477–500, 1994.

[6] P. C. Chu and J. E. Beasley. Constraint handling in genetic algorithms: The set partitioning problem. *Journal of Heuristics*, 4(4):323–357, June 1998.

[7] P. C. Chu and J. E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4(1):63–86, December 1998.

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

[9] E. J. Coffman, M. Garey, and D. Johnson. Approximation algorithms for bin-packing. *In Algorithm Design for Computer System Design*, 1984.

[10] L. Davis. Job-shop scheduling with genetic algorithms. In *International Conference on GAs*, pages 136–140. Lawrence Erlbaum, 1985.

[11] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. Wiley-Inter Science, Wiley - NY, 1998.

[12] E. Falkenauer. *Genetic algorithms and grouping problems*. John Wiley and Sons, 1998.

[13] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[14] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[15] D. Greer and G. Ruhe. Software release planning: an evolutionary and iterative approach. *Information and Software Technology*, 46(4):243–253, 2004.

[16] E. Hart, D. Corne, and P. Ross. The state of the art in evolutionary scheduling. *Genetic Programming and Evolvable Machines*, 2004 (to appear).

[17] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON*, pages 171–183, October 1993.

[18] J. Karlsson, C. Wohlin, and B. Regnell. An evaluation of methods for priorizing software requirements. *Information and Software Technology*, 39:939–947, 1998.

[19] C. Kirsopp, M. Sheppard, and J. Hart. Search heuristics, case-based reasoning and software project effort prediction. In *Proceedings of Genetic and Evolutionary Computation Conference*. Springer-Verlag, 2002.

[20] K. Kontogiannis, R. D. Mori, R. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, March 1996.

[21] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 244–253, Monterey CA, Nov 1996.

[22] T. J. McCabe. Reverse engineering reusability redundancy: the connection. *American Programmer*, 3:8–13, Oct 1990.

[23] E. Merlo, G. Antoniol, M. DiPenta, and F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analysis. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 412–416. IEEE Computer Society Press, 2004.

[24] M. Neteler, editor. *GRASS 6.1 Programmer's Manual. Geographic Resources Analysis Support System*. ITC-irst, Italy, `http://grass.itc.it/devel/`, 2005.

[25] G. R. Raidl. An improved genetic algorithm for the multiconstrained 0–1 knapsack problem. In *Proceedings of International Conference on Evolutionary Computation*, pages 207–211, 1998.

[26] J. D. Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of International Conference on Genetic Algorithms and Their Applications*, pages 193–100, Pittsburgh, PAL, 1985.

[27] E. Talbi. Métaheuristiques pour l'optimisation combinatoire muli-objectif : État de l'art. TR 98-757.33, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, 2001.