Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software xxx (2004) xxx–xxx
The Journal of Systems and Software
www.elsevier.com/locate/jss

A language-independent software renovation framework

M. Di Penta [a,*], M. Neteler [b], G. Antoniol [a], E. Merlo [c]

[a] Department of Engineering, RCOST—Research Centre on Software Technology, University of Sannio, Via Traiano, 1-82100 Benevento, Italy
[b] ITC-irst Istituto Trentino Cultura, Via Sommarive, 18-38050 Povo (Trento), Italy
[c] École Polytechnique de Montréal, Montréal, Quebéc, Canada

Received 1 April 2003; received in revised form 16 July 2003; accepted 2 March 2004

Abstract

One of the undesired effects of software evolution is the proliferation of unused components, which are not used by any application. As a consequence, the size of binaries and libraries tends to grow and system maintainability tends to decrease. At the same time, a major trend of today's software market is the porting of applications on hand-held devices or, in general, on devices which have a limited amount of available resources. Refactoring and, in particular, the miniaturization of libraries and applications are therefore necessary.

We propose a Software Renovation Framework (SRF) and a toolkit covering several aspects of software renovation, such as removing unused objects and code clones, and refactoring existing libraries into smaller more cohesive ones. Refactoring has been implemented in the SRF using a hybrid approach based on hierarchical clustering, on genetic algorithms and hill climbing, also taking into account the developers' feedback. The SRF aims to monitor software system quality in terms of the identified affecting factors, and to perform renovation activities when necessary. Most of the framework activities are language-independent, do not require any kind of source code parsing, and rely on object module analysis.

The SRF has been applied to *GRASS*, which is a large open source Geographical Information System of about one million LOCs in size. It has significantly improved the software organization, has reduced by about 50% the average number of objects linked by each application, and has consequently also reduced the applications' memory requirements.

Keywords: Refactoring; Software renovation; Clustering; Genetic algorithms; Hill climbing

1. Introduction

Software systems evolution often presents several factors that contribute to deteriorate the quality of the system itself (Lehman and Belady, 1985). First, unused components, which have been introduced for testing purposes or which belong to obsolete functionalities, may proliferate. Second, maintenance and evolution activities are likely to introduce clones, while, for example, adding support and drivers for an architecture similar to an already supported one (Antoniol et al., 2002). Third, library sizes tend to increase, because new functionalities are added and refactoring is rarely performed; for the same reasons, also the number of inter-library dependencies, some of which are circular, tends to increase. Finally, sometimes, new functionalities logically related to already existing ones are added in a non-systematic way and they result in sets of modules which are neither organized nor linked into libraries. As a consequence, systems become difficult to maintain. Moreover, unused objects, big libraries, and circular dependencies significantly increase application sizes and memory requirements. This is clearly in contrast

* Corresponding author.
E-mail addresses: dipenta@unisannio.it (M. Di Penta), neteler@itc.it (M. Neteler), antoniol@ieee.org (G. Antoniol), merlo@info.polymtl.ca (E. Merlo).

0164-1212/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2004.03.033

with today's industry hype towards porting existing software applications onto hand-held devices, such as Personal Digital Assistants (PDA), onto wireless devices (e.g., multimedia cell phones), or, in general, onto devices with limited resources.

This paper proposes the SRF to monitor and control some of the quality factors which have been described above. When the number of unused objects and clones increase, or when library sizes become unmanageable, some actions may be taken among the several possible ones. First and foremost, unused code may be removed and clones may be monitored or factored out. Furthermore, some form of restructuring, at library and at object file level, may be required. Together with monitoring and improving maintainability, the SRF eases the miniaturization challenge of porting applications onto limited resources devices.

Most of the SRF activities deal with analyzing dependencies among software artifacts. For any given software system, dependencies among executables and object files may be represented via a dependency graph, which is a graph where nodes represent resources and edges represent dependencies. Each library, in turn, may be thought of as a subgraph in the overall object file dependency graph. Therefore, software miniaturization can be modeled as a graph partitioning problem. Unfortunately, it is well known that graph partitioning is an NP-hard problem (Garey and Johnson, 1979) and thus heuristics have been adopted to find a "good-enough" solution. For example, one may be interested to first examine graph partitions by minimizing cross edges between subgraphs which correspond to libraries. More formally, a cost function describing the restructuring problem has to be defined and heuristics to drive the solution search process must be identified and applied.

We propose a novel approach in which hierarchical clustering and *Silhouette* statistics (Kaufman and Rousseeuw, 1990) are initially used to determine the optimal number of clusters and the starting population of a Software Renovation Genetic Algorithm (SRGA). This initial step is followed by a SRGA search aimed at minimizing a multi-objective function which takes into account, at the same time, both the number of inter-library dependencies and the average number of objects linked by each application. Finally, by letting the SRGA fitness function also consider the experts' suggestions, the SRF becomes a semi-automatic approach composed of multiple refactoring iterations, which are interleaved by developers' feedback. To speed up the search process, heuristics based on a Genetic Algorithm (GA) and a modified GA (Talbi and Bessière, 1991) approach were proposed. Performance improvement was also achieved by means of a hybrid approach, which combines GA strategies with hill climbing techniques.

The SRF has the advantage of being language independent. All activities, except clone detection, rely on information extracted from object files; furthermore, the clone detection algorithm adopted in the SRF is not tied to any specific programming language, provided that a set of metrics can be extracted from the source code.

The SRF has been applied to a large Open Source software system: a Geographical Information System (GIS) named *GRASS* [1] (Geographic Resources Analysis Support System). *GRASS* is a raster/vector GIS combined with integrated image processing and data visualization subsystems (Neteler and Mitasova, 2002) composed of 517 applications and 43 libraries, for a total of over one million LOCs.

The number of team members is small and it is about 7–15 active developers. Decisions are usually taken by the members most capable to solve specific problems. Developers are also *GRASS* users and they often focus on their needs within the general project.

This paper is organized as follows. First, a short review on related work (Section 2) and on main notions of clustering and GAs (Section 3), will be presented. Then, the SRF is presented in Section 4. The case study software system (i.e., *GRASS*) is described in Section 5, while results are presented and discussed in Section 6, and are followed by conclusions and work-in-progress in Section 7.

2. Related work

Many research contributions have been published about software system modules clustering and restructuring, identifying objects, and recovering or building libraries. Most of these work applied clustering or Concept Analysis (CA).

An overview of CA applications to software reengineering problems was published by G. Snelting in his seminal work (Snelting, 2000). Snelting applied CA to several remodularization problems such as exploring configuration spaces (see also Krone and Snelting, 1994), transforming class hierarchies, and remodularizing COBOL systems. Kuipers and Moonen (2000) combined CA and type inference in a semi-automatic approach to find objects in COBOL legacy code. Antoniol et al. (2001a) applied CA to the problem of identifying libraries and of defining new directories and files organizations in software systems with degraded architectures. As according to Krone and Snelting (1994), Kuipers and Moonen (2000), and Antoniol et al. (2001a), we believe that with the present level of technology a programmer-centric approach is required, since programmers are in charge of choosing the proper remodularization strategy based on their knowledge

[1] http://grass.itc.it

and judgment. A comparison between clustering and CA was presented by Kuipers and van Deursen (1999). Our work also applies an agglomerative-nesting clustering to a Boolean usage matrix, although according to Kuipers and van Deursen (1999) the matrix indicated the uses of variables by programs.

Surveys and overviews of cluster analysis applied to software systems have been published in the past, for example, by Wiggerts (1997) and by Tzerpos and Holt (1998). The latter authors (Tzerpos and Holt, 1999) defined a metric to evaluate the similarity of different decompositions of software systems. Tzerpos and Holt (2000a) proposed a novel clustering algorithm which had been specifically conceived to address the peculiarities of the program comprehension; they also addressed the issue of stability of software clustering algorithms (Tzerpos and Holt, 2000b). Applications of clustering to reengineering were suggested by Anquetil and Lethbridge (1998), that devised a method for decomposing complex software systems into independent subsystems. Source files were clustered according to file names and their name decomposition. An approach relying on inter-module and intra-module dependency graphs to refactor software systems was presented by Mancoridis et al. (1998). We share the idea of analyzing dependency graphs and of finding a tradeoff between highly cohesive and little inter-connected libraries, with Mancoridis et al. (1998).

GAs have been recently applied in different fields of computer science and software engineering. An approach for partitioning a graph using GAs was discussed by Talbi and Bessière (1991). Similar approaches were also published by Shazely et al. (1998), Bui and Moon (1996), and Oommen and de St Croix (1996). Maini et al. (1994) discussed a method to introduce knowledge about the problem in a non-uniform crossover operator and presented some examples of its application. A GA was used by Doval et al. (1999) to identify clusters on software systems. Together with Doval et al., 1999, we share the idea of a software clustering approach which uses a GA and which tries to minimize inter-cluster dependencies. Finally, Harman and et al. (2002) reported experiments of modularization and remodularization by comparing GAs with hill climbing techniques and by introducing a representation and a crossover operator tied to the remodularization problem. Their case studies revealed that hill climbing outperformed GAs. Mahdavi et al. (2003) proposed an approach aimed to combine multiple hill climbs for subsequent searches, thus reducing the search spaces.

Software miniaturization for Java application was recently addressed by *Jax* which is an application extractor for Java software systems (Tip et al., 1999) whose goal is the size reduction of Java programs with particular interest to applets to be transmitted over the network. *Jax* is based on transformations including removal of redundant methods and fields, devirtualization and inlining of method calls, renaming methods, fields, class and packages, and transforming class hierarchies. Another approach, devoted to reduce the size of Java libraries for embedded systems, was proposed by Rayside and Kontogiannis (2002). While the approach proposed by Rayside and Kontogiannis (2002) and *Jax* are tied to a programming language, ours is not. Our approach also differs from *Jax* in philosophy since we do not limit ourselves to reduce the size of the instance application to be executed, but we also support the reorganization of a software system whose structure has been deteriorated because of its evolution. The reduction of memory requirements is thus just one of the effects of the reorganization.

This paper extends preliminary contributions (Di Penta et al., 2002; Antoniol et al., 2003). Together with Di Penta et al. (2002), we share the choice *GRASS* as target application and several activities carried out to refactor libraries.

3. Background notions

The fundamental activity of the SRF is library refactoring. This requires the integration of clustering and GA techniques in a semi-automatic, human-driven process. Clustering deals with the grouping of large amounts of things (*entities*) in groups (*clusters*) of closely related entities (Kaufman and Rousseeuw, 1990; Anderberg, 1973). Clustering is used in different areas, such as business analysis, economics, astronomy, information retrieval, image processing, pattern recognition, biology, and others. GAs come from an idea, born over 30 years ago, of applying the biological principle of evolution to artificial systems. GAs are applied to different domains such as machine and robot learning, economics, operations research, ecology, studies of evolution, learning and social systems (Goldberg, 1989; Mitchell, 1996).

In the following subsections, for sake of completeness, only some essential notions are summarized, because describing the different types of clustering algorithms or the details of GAs is out of the scope of this paper. More details can be found in Anderberg (1973) for clustering and in Goldberg (1989) and Mitchell (1996) for GAs.

3.1. Agglomerative hierarchical clustering

In this paper, the agglomerative-nesting (*Agnes*) algorithm (Kaufman and Rousseeuw, 1990) was applied to build the initial set of *candidate libraries*. *Agnes* is an agglomerative, hierarchical clustering algorithm: it builds a hierarchy of clusters in such way that each level contains the same clusters as the first lower level, except

for two clusters, which are joined to form a single cluster.

3.2. Determining the optimal number of clusters

To determine the actual or optimal number of clusters, people traditionally rely on the plot of an error measure representing the dispersion within a cluster. The error measure decreases as the number of clusters, k, increases, but for some values of k the curve flattens. Kaufman and Rousseeuw (1990) proposed the *Silhouette* statistics for estimating and assessing the optimal number of clusters. For the observation i, let $a(i)$ be the average distance to the other points in its cluster, and $b(i)$ the average distance to points in the nearest cluster. Then the *Silhouette* statistics is defined as

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}. \quad (1)$$

Kaufman and Russeeuw suggested choosing the optimal number of clusters as the value maximizing the average $s(i)$ over the dataset. Traditionally, it is assumed that the error curve *knee* indicates the appropriate number of clusters (Gordon, 1988).

Often, a compromise has to be accepted between maximizing the *Silhouette* (and thus having highly cohesive clusters) and obtaining an excessive number of clusters (that in our application, causes library fragmentation).

3.3. Genetic algorithms

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 (Garey and Johnson, 1979). Roughly speaking, a GA may be defined as an iterative procedure that searches for the best solution of a given problem among a constant-size population, represented by a finite string of symbols, the *genome*. The search is made starting from an initial population of individuals, often randomly generated. 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 individuals (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 and 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 or on a given value of the fitness function.

3.3.1. Hill climbing and GA hybrid approaches

As suggested by Goldberg (1989), hybrid GAs may be advantageous when there is the need for optimization techniques tied to a specific problem structure. The *in-large* perspective of GAs may be combined with the precision of local search. GAs are able to explore large search spaces, but often they reach a solution that is not accurate, or they very slowly converge to an accurate solution. On the other hand, local optimization techniques, such as hill climbing, quickly converge to a local optimum, but they are not very effective for searching large solution spaces because of the possible presence of local maximum or plateaus.

There are at least two different ways to hybridize a GA with hill climbing techniques. The first approach attempts to optimize the best individuals of the last generation, using hill climbing techniques. The second approach uses hill climbing to optimize the best individuals of each generation. Applying hill climbing on each generation could be expensive. However, this technique "inserts" in each generation *high quality individuals*, who are determined by the optimization phase, and therefore reduces the number of generations requested to achieve convergence.

4. The refactoring framework

As highlighted in the introduction, the proposed framework consists of several steps:

- First and foremost, software system applications, libraries, and dependencies among them are identified;
- Unused functions and objects are identified, removed or factored out;
- Duplicated or cloned objects are identified and possibly factored out;
- Circular dependencies among libraries, which cause a library to be linked each time another circularly linked library is needed, are removed, or, at least, reduced;
- Large libraries are refactored into smaller ones and, if possible, transformed into dynamic libraries; and
- Objects which are used by multiple applications, but which are not yet organized into libraries, are grouped into new libraries.

The SRF activities and the adopted representations are detailed in the following subsections.

4.1. Software system graph representation

A graph representation of dependencies between object modules is central to our framework and most of the SRF computations rely on it. Software systems can be represented by an instance of the *System Graph* (*SG*), an example of which is depicted in Fig. 1.

SG is defined as

$$SG \equiv \{O, L, A, D\}, \tag{2}$$

where $O \equiv \{o_1, o_2, \ldots, o_p\}$ is the set of all object modules; $L \equiv \{l_1, l_2, \ldots, l_n\}$, where $l_i \subseteq O$ $i = 1, \ldots, n$, is the set of all software system libraries. Libraries, subsets of objects, are depicted in Fig. 1 as rounded boxes; $A \equiv \{a_1, a_2, \ldots, a_m\}$, where $A \subseteq O$ and $A \cap \{\cup_i l_i\} = \emptyset$, is the set of all software system applications. Applications, i.e. the object modules containing the `main` symbol, are represented in Fig. 1 as squares source nodes; [2] and $D \subseteq O \times O$ is the set of oriented edges $d_{i,j}$ representing dependencies between objects.

We can extract from the *SG* graph two other graphs useful for our refactoring purposes. The first graph is called *Use Graph* and it highlights the uses of objects by applications or by libraries. The *use* relationship is defined as

$$a_x \text{ uses } o_y \iff \exists \text{ path}\{a_x, \ldots, o_y\} \in SG. \tag{3}$$

In other words the *Use Graph* highlights the reachability between applications and library objects in *SGs*. Such reachability can be obtained computing a *k*-fold product on the graph represented by an adjacency matrix.

Similarly, the second graph is called *Dependency Graph* and it is used to represent existing dependencies between two or more libraries, or between to-be-refactored objects contained in a library. The clustering algorithm should avoid inter-cluster dependencies. The *dependency* relationship is defined as

$$o_x \text{ depends\_on } o_y \iff o_x \text{ uses } o_y \wedge o_x \in L \wedge o_y \in L. \tag{4}$$

In particular, a dependency (o_x, o_y) is considered an inter-library dependency, i.e., a dependency that increases the coupling, if $o_x \in l_i$, $o_y \in l_j$, and $i \neq j$.

Given the above definition of *SG*, the SRF activities can be graphically shown in Fig. 2.

4.2. Graph construction

Prior to recover dependencies among applications and libraries, and among libraries themselves, executable applications composing the software system must be identified. In this paper we rely on an approach similar to the one proposed by Antoniol et al. (2001a). However, Antoniol et al. (2001a) identified applications by detecting all source files containing the definition of a `main` function.

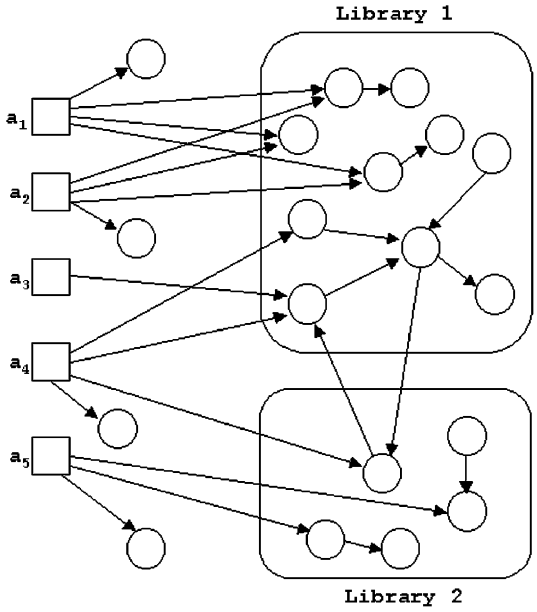

Fig. 1. Example of system graph.

Once applications and existing libraries are identified, the *SG* graph can be built. Given the *use* relationship between an object module requiring a symbol and a module defining it, the corresponding *SG* is built via the *transitive closure* of the *use* relationship, starting from the `main` object of each application and from each library. In other words, for each application, undefined symbols are identified and recursively resolved (possibly adding new undefined symbols to the stack) first inside the objects contained in the same path (i.e., other modules of the application), then inside libraries. A similar process is performed to detect dependencies among libraries. Finally, the *use graph* and the *dependency graph*, represented as adjacency matrices *MU* and *MD*, are extracted from the *SG* graph.

4.3. Handling unused objects

Symbols defined in libraries which are neither used by applications nor by other libraries are likely to represent useless resources. Their presence is often due to utility functions which are inserted in libraries but which are not used by the current set of applications, or it is due to not yet fully implemented features. The objects defining these unused symbols should be removed from the libraries, provided that they do not also export used symbols. In the opposite case such an object should be left into library and its corresponding source file should be restructured. One possible refactoring strategy is to

[2] Applications are not the only source nodes. In fact, as it will detailed later, also unused objects have no incoming edges, even if they can be distinguished from the applications since the latter also define a `main` symbol.

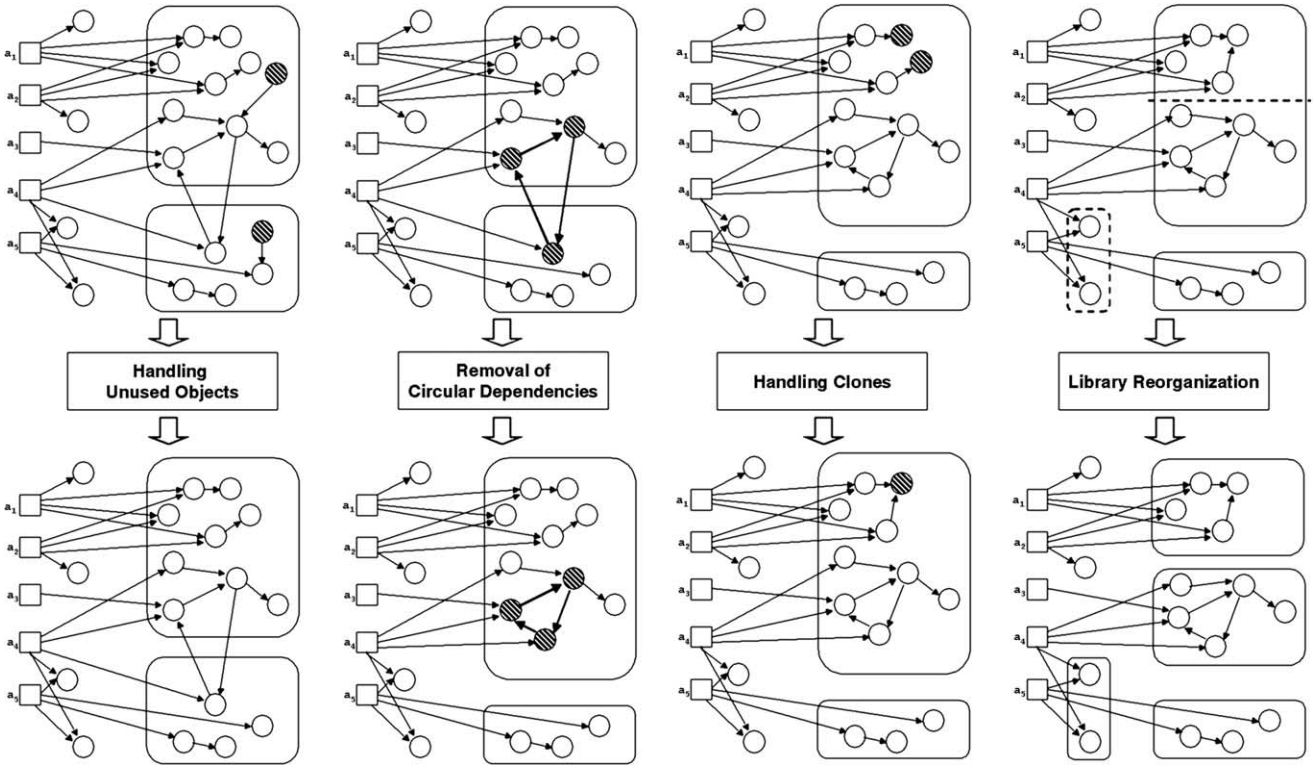

Fig. 2. The framework activities.

create two new libraries from each library, one of which containing all the unused symbols and the other one containing all the used symbols.

4.4. Removal of circular dependencies among libraries

The *DG* introduced in Section 4.1 captures dependencies among the different libraries and allows the identification of strongly connected components. In particular, circular dependencies between libraries cause a library to be linked each time the other one is needed. Once these dependencies are identified, four strategies could be used to remove them:

(1) *Move the object* which causes the circular dependence to another library. This is only feasible if the object does not need resources located in its original library and it is not needed by that library;
(2) *Duplicate the object:* like the previous case, this is appropriate, if the object does not need resources located in the original library but, differently from the previous case, the object is required in that library. Moving the object the library outside will make the situation worse;
(3) *Merge the two libraries:* this strategy should be avoided whenever possible because it increases library sizes; however, it could be the only available solution when the number of objects causing circular and, in general, inter-library dependencies is very high;
(4) *Create dynamic libraries:* instead of merging circularly dependent libraries, one may decide to make them dynamic. Circular dependency problem is not solved, but the average amount of resources needed is reduced, as described in Section 4.6.2.

When the *DG* does not allow the removal of circular dependencies and, when, for performance reasons, options three and four cannot be adopted, a deeper analysis should be performed to identify dependencies at the granularity level of functions rather than objects.

Finally, the existence of a complex dependency relationship between two libraries, if confirmed by developer's feedback, indicates the possibility of a library design which has not been done with miniaturization in mind. In this case, library objects should be merged and then refactored again in new clusters, adopting the process detailed in Section 4.6.

4.5. Identification of duplicate symbols and clones

Examining the list of symbols defined in each library allows the comparison of exported symbol names. It is worth noting that homonym symbols in different librar-

ies may refer to completely different functions, external variable or data structures. On the other hand, two or more symbols may have different names, but they may correspond to duplicated functions. Therefore, clone detection analysis is helpful for library renovation. In this paper a metric-based clone detection process (Antoniol et al., 2001b), aimed at detecting duplicated functions, is adopted. The obtained results suggest different possible actions:

(1) If a whole, duplicated, object module has been detected inside two or more libraries, then it should be left in only one of these, unless it conflicts with circular dependencies removal (see Section 4.4);
(2) If duplicated functions are identified inside different objects, refactoring could be performed by moving them outside their respective objects and by applying considerations similar to the previous case; and
(3) Clone detection may reveal clones outside libraries, since applications may contain duplicated portions of code, in their objects. In some cases, it could be useful to remove such duplicated portions of code and place them into new libraries.

Preliminary to clone refactoring is impact analysis in terms of introduced dependencies, especially circular dependencies, since clone removal may increase dependencies. As explained in Section 4.4 and as it will be shown in Section 4.6, sometimes an object is duplicated to reduce dependencies. In general, it may be preferable to duplicate few objects, rather than introducing a dependence that causes, for a subset of the applications, the linking or the loading of one or more additional libraries. Clearly, if the process duplicates a conspicuous number of objects into two or more libraries, these objects can be refactored, as explained in Section 4.6.2, into a new library on which the old libraries will depend.

Overall, clone removal aims to improve the software system maintainability, although attention should be paid to avoid deteriorating software system reliability, and to reflect the developers' objectives (Cordy, 2003). Clone can also contribute decrease the overall software system size; again a tradeoff should be made: sometimes clone refactoring (especially for very small clones) produces a system bigger than the original one.

4.6. Library refactoring

The last phase of the SRF is devoted to splitting existing, large libraries into smaller clusters of objects. Basically, the idea is similar to that proposed by Antoniol et al. (2001a) to identify libraries. To minimize the average number of libraries required by each program, objects used by a common set of programs should be grouped together. Antoniol et al. (2001a) used a *concept lattice* to group objects into libraries. Although the lattice gives useful information, it becomes unmanageable when a large number of applications and libraries must be handled (Anquetil, 2000), as in our case study. Instead of pruning information on a *concept lattice* like Siff and Reps (1999) and Tonella (2001), clustering analysis was performed, similar to Anquetil and Lethbridge (1998), Mancoridis et al. (1998), and Merlo et al. (1993).

The library refactoring process, as shown in Fig. 3, consists of the following steps:

(1) Determine the optimal number of clusters and an initial solution;
(2) Determine the new candidate libraries using a GA; and
(3) Ask developers for feedback and, possibly, iterate through step 2.

4.6.1. Determining the optimal number of clusters and a suboptimal solution

As explained in Section 3.2, the optimal number of clusters is determined by inspecting the *Silhouette* statistics computed on the suboptimal clusters which are determined using agglomerative-nesting clustering. Given the curve of the average *Silhouette* values obtained from Eq. (1) for different numbers k of clusters, we choose for some libraries the *knee* of that curve (Kaufman and Rousseeuw, 1990) as the optimal number of clusters, instead of considering the maximum of the curve because that is often too high for our refactoring purpose.

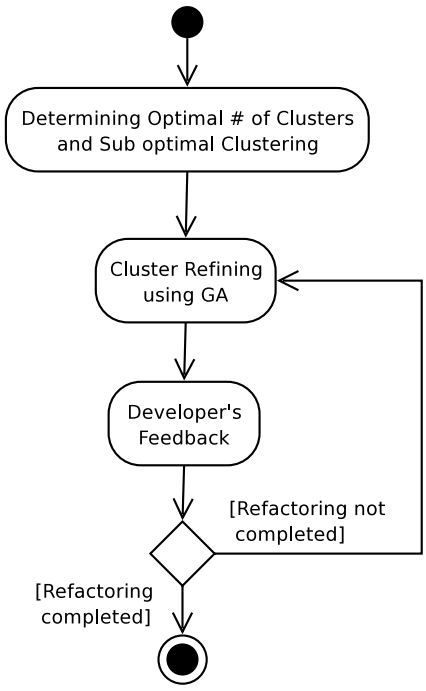

Fig. 3. Activity diagram of the library refactoring process.

We have also incorporated experts' knowledge in the choice of the optimal number of clusters and we have considered a tradeoff between excessive fragmentation produced by too many clusters and excessive library size produced by fewer clusters. The suboptimal solution for the chosen value of k is then used as the starting point of the application of a GA, which is the subsequent framework step.

The effectiveness of the refactoring process is evaluated by a quality measure of the new library organization. Let k be the number of clusters $l_{x_1},\ldots,l_{x_k}$ obtained from a library l_x. The *Partitioning Ratio* (*PR*) is defined as

$$PR(x) = 100 \times \sum_{i=1}^{m} \frac{\sum_{j=1}^{k} |l_{x_j}| \times mu_{i,x_j}}{|l_x| \times mu_{i,x}}, \quad (5)$$

where $|l_x|$ is the number of objects archived into library l_x. The smaller is the *PR*, the more effective is the partitioning since the average number of objects linked or loaded by each application is smaller than using the whole old library.

4.6.2. Refining the solution using genetic algorithms

The solution determined by the previous step presents two main drawbacks:

(1) The number of dependencies between the new libraries may be high. Each time a symbol from a library is needed, another library may also need to be loaded, therefore reducing the advantage of having new smaller libraries; and
(2) New libraries may not be meaningful with respect to developers' intentions whose feedback has to be incorporated in the refactoring process.

Of course, as shown by Di Penta et al., 2002, an important step to perform is the conversion of static libraries into dynamically-loadable libraries (DLL), so that each and possibly small library is loaded at runtime only when needed, and it is unloaded when it is no longer useful. However, the DLL approach presents a main drawback: loading and unloading libraries may be cause of a significant decrease in performance and its use should be limited, when performance constitutes an essential requirement, and, whenever possible, it should be accompanied by dependency minimization.

The genome has been encoded using a bit-matrix encoding. The genome matrix *GM* for each library to refactor corresponds to a matrix of k rows and $|l_x|$ columns, where $gm_{i,j} = 1$ if the object j is contained into cluster i, 0 otherwise. Clearly, the presence of the same object in more libraries is indicated by more "1" in the same column (this is not possible using the array genome, widely used for graph partitioning problems). As already stated, instead of randomly generating the initial population (i.e., the initial libraries), the GA is initialized with the encoding of the set of libraries obtained in the previous step.

The fitness function has been conceived to balance four factors:

(1) The number of inter-library dependencies at a given generation;
(2) The total number of objects linked to each application which should be as small as possible;
(3) The size of the new libraries; and
(4) The feedback given by the developers.

Overall, the fitness function F is defined in terms of four factors which are the *Dependency Factor* (*DF*), the *Partitioning Ratio* (*PR*) defined by Eq. (5), the *Standard Deviation Factor* (*SDF*), and the *Feedback Factor* (*FF*).

DF is defined as:

$$DF(g) = \sum_{i=0}^{|l_x|-1} \sum_{j=0}^{m-1} gm_{i,j} \sum_{k=0}^{k=m-1} md_{j,k}(1 - gm_{i,k}) \times [1 - \delta(k,j)], \quad (6)$$

where $\delta(x,y)$ is the well-known Kronecker delta function:

$$\delta(x,y) = \begin{cases} 1 & x = y, \\ 0 & x \neq y, \end{cases}$$

$gm_{i,j}$ is the genome encoding i.e., the $GM[i,j]$ bit matrix entry. As shown in Eq. (6), the $DF(g)$ is incremented each time an object (i.e., a high bit in the genome) depends from another object not contained in the same cluster. *SDF* can be thought of as the difference between the initial library sizes standard deviation and the one at the current generation. Without taking *SDF* into account, the SRGA may attempt to reduce dependencies by grouping a large fraction of the objects in the same library and it may negatively affect the *PR*. A similar factor was also applied by Talbi and Bessière (1991). Given the arrays of library sizes S_0 and S_g, respectively for the initial population and for the gth generation, *SDF* is

$$SDF(g) = |\sigma_{S_0} - \sigma_{S_g}|. \quad (7)$$

The fourth factor takes into account the developers' feedback. After a first execution of the SRGA without considering *FF*, developers are asked to provide a feedback on the proposed new libraries. Developers' feedback is stored in a bit-matrix *FM*, which has the same structure of the genome matrix and which incorporates those changes to the libraries that developers suggested. After this feedback, the SRGA is run again taking into account, this time, the *feedback factor FF*, based on the difference between the genome and the *FM* matrix:

$$FF = \sum_{i=1}^{k} \sum_{j=1}^{|l_x|} |gm_{i,j} - fm_{i,j}|. \quad (8)$$

In other words, the *FF* counts the number of differences between the genome and the refactoring proposed by developers.

The fitness function F is formally defined as

$$F(g) = DF(g) + w_1 PR(g) + w_2 SDF(g) + w_3 FF(g), \quad (9)$$

where w_1, w_2 and w_3 are real, positive weighting factors for the *PR*, *SDF*, and *FF* contribution to the overall fitness function. The higher is w_1, the smaller will be the overall number of objects linked by applications at the expense of dependency reduction. Similarly, the higher is w_2, the more similar will be the result to the starting set of library, again, at the expense of a satisfactory dependency reduction. After the first preliminary run of the SRGA which must be performed with $w_3 = 0$, w_3 should be properly sized to weight the influence of developers' feedback. As stated in (9), our fitness function is multi-objective (Deb, 1999). Notice that, since we aim to give maximum priority to dependency reduction, the *DF* weight is set to 1. Successively, w_1, w_2 and w_3 are selected using a trial-and-error, iterative procedure, adjusting them each time until the *DF*, *PR*, *SDF*, and *FF* obtained at the final step were satisfactory. The process is guided by computing each time the average values for *DF*, *PR*, *SDF*, and *FF*, and by plotting their evolution, to determine the 3D space region in which the population should evolve.

The crossover operator used in this paper is the *one point crossover* which exchanges the content of two genome matrices around the same random column (see Fig. 4a). The mutation operator works in two modes:

(1) with probability *pmut*, it takes a random column and randomly swaps two bits: this means that, if the two swapped bits are different then an object is moved from a library to another (see Fig. 4b); or
(2) with probability *pclone* < *pmut*, it takes a random position in the matrix: if it is zero and the library is dependent on it, then the mutation operator clones the object into the current library (Fig. 4c).

Noticeably, cloning an object increases both *PR* and *SDF*, and therefore it must be minimized. The SRGA heuristically activates the cloning only for the final part of the evolution (after 66% of generations in our case study). Our strategy favors dependency minimization by moving objects between libraries. At the end, we attempt to remove remaining dependencies by cloning objects. Obviously, at the end of the refactoring process cloned objects should be factored out again. For example, if objects o_a and o_b are contained in both l_i and l_j, then o_a and o_b should be moved into a third library on which l_i and l_j depend.

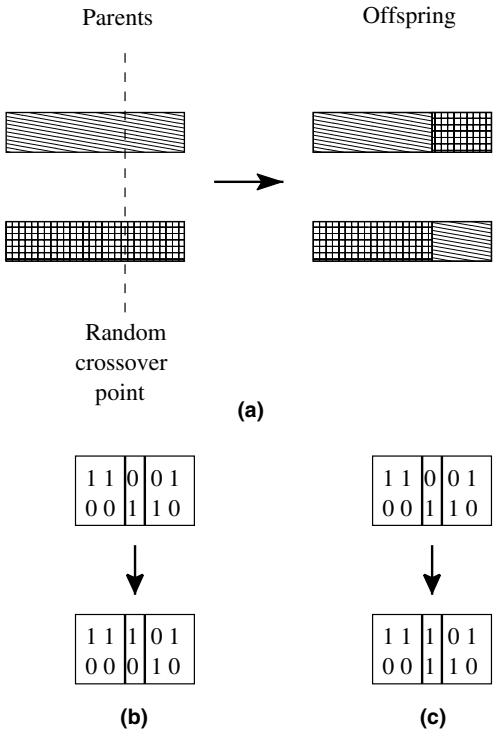

Fig. 4. Genetic operators: (a) crossover, (b) mutation (move an object), and (c) mutation (clone an object).

Finally, we have introduced the *Lock Matrix* (*LM*) as a further, stronger level of developers' feedback. When developers strongly believe that an object should belong to a cluster, *LM* matrix gives them the possibility to enforce such a constraint. The mutation operator does not perform any action that would bring a genome in a inconsistent state with respect to the *Lock Matrix*.

The population size and the number of generations are determined by using an iterative procedure, which doubles both of them each time until the obtained *DF*, *PR* and *FF* are equal to those obtained at the previous iterative step.

The SRGA suffers from slow convergence. To improve its performance, is has been hybridized with hill climbing techniques. In our experience, applying hill climbing only to the last generation significantly improves neither the performance nor the results. On the opposite, applying hill climbing to the best individuals of each generation makes the SRGA converge significantly faster.

4.7. Identification of new libraries

Due to its evolution, a software system tends to contain objects that, even if used by a common set of applications, are not contained in any library. Their identification and organization into libraries should therefore be desirable. The factoring process is quite similar to that described in the previous section. In par-

ticular, a *MU* matrix is built on a subgraph of the *use graph* obtained by removing all the already existing libraries. Then, a first set of new *candidate libraries* is built by analyzing the dendrogram and the *Silhouette* statistics. These libraries are then refined with the aid of the SRGA and of developers' feedback.

4.8. Tool support

To support the refactoring process, different tools have been conceived:

(1) *The application identifier* identifies the list of object modules containing the `main` symbol by using the `nm` Unix tool;
(2) *The graph extractor*, which is also based on the `nm` tool, which produces the *System Graph*, the *Use Graph*, and the *Dependency Graph*. The *graph extractor* also exports data in `.DOT` format, to allow visualization and analysis using the `Dotty` graph visualization tool; [3]
(3) *The unused symbol identifier* produces, for each library, the list of the symbols which are not used by any application or library together with the object names in which those symbols are contained;
(4) *The circular dependency identifier* produces the list of all circular paths among libraries;
(5) *The duplicated symbol identifier* identifies the list of duplicated and defined external symbols. It is used in conjunction with the metric-based *clone detector* (see Antoniol et al., 2001b, for details) and with the *dependency graph extractor* to minimize the presence of clones inside libraries;
(6) *The number of clusters identifier* implements the *Silhouette* statistics. In particular, implementations available in the *cluster* package of the *R Statistical Environment* [4] have been used;
(7) *The library refactoring tool* supports the process of splitting libraries in smaller clusters. Cluster analysis is performed by the *Agnes* function available in the *cluster* package of *R Statistical Environment*;
(8) *The GA library refiner* is implemented in C++ using the *GAlib*; [5] and
(9) *The developers' feedback collector* is a web application that allows developers to post their feedback about the produced libraries on an appropriate web site.

The SRF works under any standard Unix operating system, or under any operating system which supports the GNU tool set. In particular, the SRF uses the standard Bourne shell (or the new Bash), the Perl interpreter, the *R statistical environment* and a C++ compiler for the GA library refiner. To collect the programmers' feedback, the SRF relies on a PHP web application (the *developers' feedback collector*). Since the required infrastructure is available under several operating systems (both Unixes and Windows) the SRF is widely portable.

5. Case study

As mentioned in the introduction, the SRF has been applied to *GRASS*, which is a large open source GIS. In particular, the *GRASS* CVS development snapshot of April 5, 2002 [6] was used as a case study. Its characteristics are summarized in Table 1.

GRASS modules, which correspond to applications and which represent commands, are organized by name, based on their function class such as display, general, imagery, raster, vector or site, etc. The first letter of a module name refers to a function class 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 "front.end". If there are no command-line arguments entered by a user, the "front.end" module calls the interactive version of a command. Otherwise, it will start the command-line version. If only one version of the specific command exists, i.e., if there is only one command-line version available, the command is executed. 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 several hundreds 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 the "*GRASS* 5.0 Programmer's Manual" (Neteler, 2001).

6. Case study results

This section presents the results obtained by applying the SRF, which has been described in Section 4, to *GRASS*.

6.1. Handling unused objects

Out of 921 objects composing *GRASS* libraries, 89 were not used by any application, nor by other libraries. When refactoring libraries with the SRF, those objects will be moved and organized into a separate cluster, thought of as a sort of repository to be "frozen" for fu-

[3] http://www.research.att.com/sw/tools/graphviz/
[4] http://www.r-project.org
[5] http://lancet.mit.edu/ga/
[6] Downloadable from http://grass.itc.it

Table 1
GRASS key characteristics

Pre-existing libraries	43
Library objects	921
Applications	517
C source files	7107
C KLOC	1014

ture uses. A deeper analysis revealed that some functions, which are contained in unused objects, wrap lower level *GRASS* functions such as `db_create_index`, wrap standard library and system call functions such as `scan_dbl`, `scan_int`, `whoami`, and, in general, provide some simple functionalities using lower level functions such as `datetime_is_same`, that compares two `DateTime` structures. An interesting example is the library `libdbmi` (see also Section 6.4): out of 97 objects, 19 were not used at all. In all cases, the unused functions corresponded to one or more wrapped, lower level functions, that have been directly used by applications.

6.2. Removal of circular dependencies among libraries

Three cases of circular dependencies among libraries were found. The first dependency was between `libstubs.a` and `libdbmi.a`. In particular, we discovered that `libstubs.a` required one symbol, located inside the `error.o` module which belonged to `libdbmi.a`. On the other hand, `libdbmi.a` required 27 symbols from `libstubs.a`. The obvious solution was to move `error.o` into `libstubs.a`: this required moving in that library also the module `alloc.o`, since it depends from `error.o`.

The second circular dependency was found between `libgis.a` and `libcoorcnv.a`. In particular, `libgis.a` required three symbols. Such symbols were located in the module `datum.o` from `libcoorcnv.a`. In the other direction, `libcoorcnv.a` dependencies involved 13 symbols from `libgis.a`. Moving `datum.o` into `libgis.a` resolved the problem.

Finally, circular dependencies were found between `libvect.a` and `libdig2.a`. They involved 13 symbols in one direction and 31 symbols in the other direction. Symbols involved in the dependencies were located in several different objects. The links present in the dependency graph excluded the possibility of resolving circular dependencies between `libvect.a` and `libdig2.a` by simply moving or duplicating objects. The decision taken together with *GRASS* developers was initially to merge the two libraries which, in effect, have been designed to work together, and then try to refactor the new library (see Section 6.4).

6.3. Identification of clones

Clone detection was performed at two different levels of the software system architecture: within libraries and on the whole system. In the first case, clone detection aimed at library renovation; in the second case, the objective was to identify portions of duplicated code that could be potentially re-organized into new libraries.

Table 2 reports results obtained from clone analysis in terms of the total number of analyzed functions, the number of clone clusters (Antoniol et al., 2002) detected, and the number and the percentage of cloned functions. Finally, clones were computed while filtering out the shortest functions; for example, two functions that simply return a value are clones by definition, but they are not significant and should not be taken into consideration. Results are reported considering two thresholds of function size: functions longer than five and than 10 LOCs.

As shown in Table 2, the overall percentage of clones is not negligible (26.04%), even considering only functions longer than five LOCs (16.38%) and it suggests a potential for reduction in the number of the cloned functions. Clearly, the actual reduction rate depends on the number of *false positives* which typically include functions that simply contain a list of calls to other functions (where the number of calls and of parameters match), functions that print different error messages and, in general, any other function that shares the same metrics while being different.

The number of clones contained inside libraries is low, indicating that the developers accurately factored functions and objects to avoid duplicates. Finally, we investigated the set of clones between libraries and objects outside libraries in the perspective of possible refactoring. The analysis of clones inside libraries revealed an

Table 2
Results of clone detection

	Total number of functions	Number of clone clusters	Number of cloned functions	Percent of cloned functions (%)	Threshold (LOCs)
Overall	22,229	2019	5789	26.04	5
		1404	3641	16.38	10
Within libraries	5271	72	180	3.41	5
		41	101	1.92	10
Outside libraries	16,958	1817	4974	29.33	5
		1290	3268	19.27	10
Libraries vs. outside	22,229	130	635	2.86	5
		73	272	1.22	10

interesting situation: 16 functions from library `libortho`, were cloned across `libimage_sup`, `libgmath` and `libtrans`. Nine of the cloned functions were devoted to performing matrix algebra. By analyzing the *Dependency Graph* of `libortho` (see Fig. 5), a subgraph composed of such functions was identified and depicted in the box on the right. On the other hand, seven of the functions in the box on the left were cloned in `libimage_sup`. In particular, the entire structure enclosed in the rounded-dashed-box was replicated in that library. `libortho` was split `libortho` in two libraries, shown in the two boxes in Fig. 5:

(1) A library (`libmatrix`) to handle matrices; and
(2) A library (`libcamera`) to handle photogrammetric computations for aerial cameras.

Cloned functions contained in these two libraries were removed from `libimage_sup`, `libgmath` and `libtrans`.

Several "interesting" clones were also found outside libraries. In particular, the `r.mapcalc3` application contains four clusters of cloned functions, spanning from 27 to 59 LOCs in size. These cluster contain mathematical functions, cloned to handle different data types. In this case, refactoring is clearly possible by generalizing the operations and by abstracting types.

Finally, we analyzed clones between applications and libraries. In most cases clones were revealed to be part of *legacy* applications developed before the corresponding functions were added into a library. Unfortunately, the application was never changed afterwards. A relevant fraction of about 20% of these clones was discovered in the *contrib* subsystems, which had often been developed by third parties and therefore which were not always properly aligned with respect to the rest of the system.

6.4. Library refactoring

Refactoring was performed on libraries which were composed of a large number of objects (see Table 3), by following the process described in Section 4.6 and depicted in Fig. 3. As suggested by developers, `libproj` was not refactored, because it was under development by a different team. As explained in Section 6.2, `libvect-new` library was obtained by merging `libvect.a` and `libdig2.a`.

Table 3
GRASS largest libraries

Library	Objects
libgis	184
libdbmi	97
libproj	119
libvect-new	54

Silhouette statistics was used to determine the optimal number of clusters for each library. Values of such statistics, are plotted in Fig. 6, for different number of clusters. We decided to split `libgis` into four clusters (instead of the six proposed in Di Penta et al., 2002), and to divide `libvect-new` and `libdbmi` into three clusters. It is worth noting that, for `libgis`, the number of clusters was chosen in correspondence of the *Silhouette* maximum; for the other two libraries, a

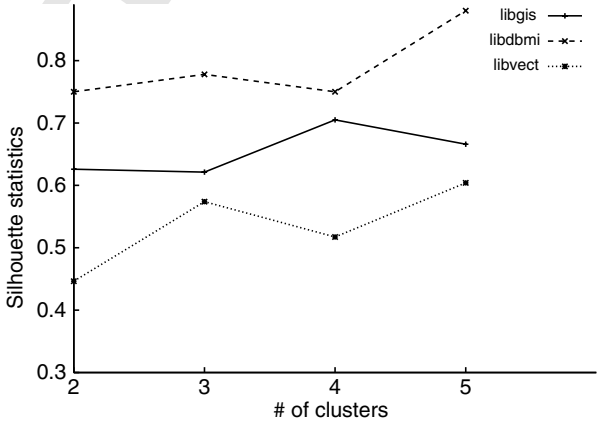

Fig. 6. Silhouette statistics for different number of clusters.

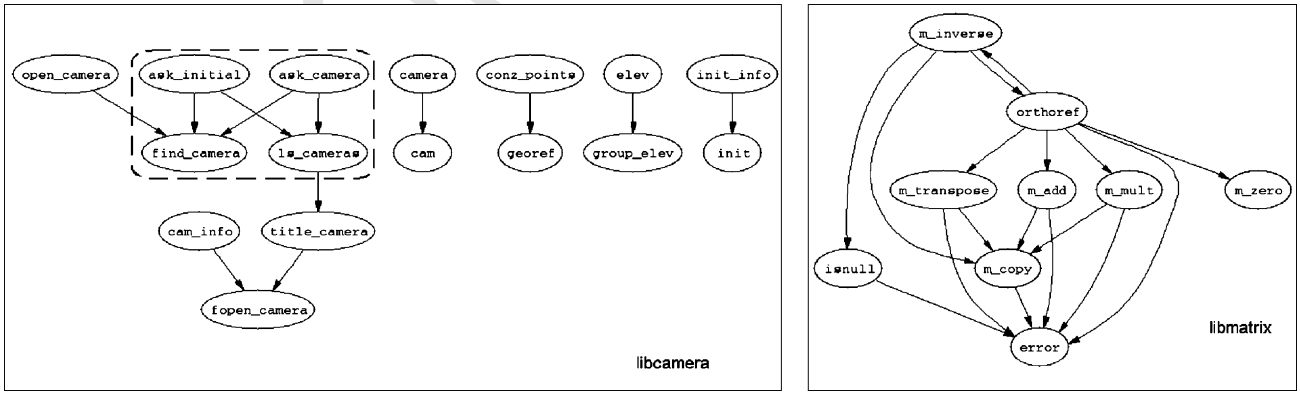

Fig. 5. Splitting library libortho.

compromise was accepted between maximizing the *Silhouette* and avoiding excessive fragmentation.

Subsequently, a preliminary clustering was performed and it was refined by an initial execution of the SRGA, which had been performed without considering any developers' feedback and by setting $w_3 = 0$. Table 4 reports for each library:

- The number of objects composing the library;
- The number of *candidate libraries* the original library is refactored into and the corresponding *Silhouette* statistics value;
- The number of inter-library dependencies and *PR* before applying the SRGA; and
- The number of inter-library dependencies and *PR* after applying the SRGA.

As shown, the SRGA reduced `libgis` dependencies from 579 to 26, while keeping *PR* almost constant (from 51% to 48%). A significant reduction of inter-library dependency was obtained (from 237 to 4 for `libdbmi` and from 66 to 3 for `libvect`), while slightly reducing *PR*, except for `libdbmi`, where it increased to 46% and it was worse than the preliminary solution.

The first refactored architecture of the *candidate libraries* was submitted to *GRASS* developers to seek their feedback. For `libgis`, manual analysis indicated that the first cluster should contain "utility" and "allocation" functions, the second "area" and "geodesic" functions, the third "color-related" functions, and the fourth "raster" functions. For `libvect-new`, developers indicated that the first cluster should contain basic file-system operations and the other two clusters should include all other functions without any further distinction. The feedback for `libdbmi` was quite different with respect to the other two libraries. In this case, developers confirmed that the solution suggested by the hierarchical clustering performed before applying the SRGA reflected their own conception of libraries. A manual graph analysis via *Dotty* graph visualization agreed, too. In fact, as also reported by Di Penta et al. (2002), the library was split into the three following clusters:

- `libdbmi-1` contains (19) unused objects;
- `libdbmi-2` contains (30) objects which are directly used by applications; and

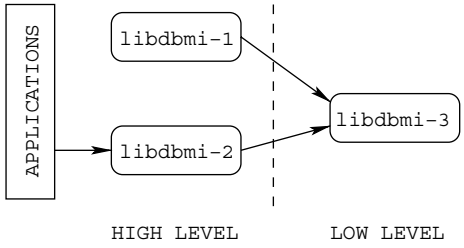

Fig. 7. New libdbmi layering structure.

- `libdbmi-3` contains 48 objects, which are only internally used by `libdbmi`, and represents some sort of "low-level" library.

Fig. 7 reports the layering structure of the clusters extracted from `libdbmi`. To avoid circular dependencies, one object was moved from `libdbmi-3` to `libdbmi-1`. Clearly, when refactoring a large software system such as *GRASS*, a compromise should be accepted between having small and decoupled clusters like those generated by applying the SRGA and having clusters that are not totally decoupled, but are conceptually cohesive, since they contain functions which implement closely-related tasks. In the latter case, memory optimization is possible adopting, as noted, dynamically loadable libraries (in spite, however, of performances, as explained in Section 4.6.2). We decided to leave `libdbmi` clusters as they were after hierarchical clustering and to perform a "second iteration" of the SRGA refactoring on `libgis` and `libvect-new`, while taking into consideration also the *Feedback Factor FF*, this time. For sake of completeness, we also reported results for `libdbmi`. By varying the w_1, w_2 and w_3 thresholds, we obtained different results. As shown in Table 5, it was never possible to achieve a complete cluster decoupling and to obtain, at the same time, libraries which were very close to the structure proposed by developers.

In Table 5, the comparison of the first three columns with the last three highlights that, after the first SRGA iteration, the coupling between clusters remained low. On the other hand, as highlighted by a high *FF* value before the second iteration, identified libraries tend to have a structure which is somehow different with respect to developers' intention. The second iteration of the SRGA tried to decrease *FF*, while, unfortunately, coupling increased. At such a stage, in the authors' opinion, developers may decide either to produce *meaningful* libraries

Table 4
Results of the library refactoring process before considering feedback ($w_3 = 0$)

Library	Number of objects	Candidate libraries (k)	Silhouette statistics	Before GA		After GA	
				DF	PR (%)	DF	PR (%)
`libgis`	184	4	0.70	579	51	26	48
`libdbmi`	97	3	0.78	237	35	4	46
`libvect`	54	3	0.57	66	46	3	40

Table 5
Results of the second round of the library refactoring process ($w_3 \neq 0$)

Library	Number of objects	Candidate libraries (k)	Before second round			After second round		
			FF	DF	PR (%)	FF	DF	PR (%)
libgis	184	4	203	26	48	128	60	52
libdbmi	97	3	97	4	46	23	43	39
libvect	54	3	72	3	40	30	6	52

Table 6
Performance comparison between pure GA and hybrid GA with hill climbing

Library	Pure GA		Hybrid GA		Fitness difference (%)	Time difference (%)
	Fitness function	Time (s)	Fitness function	Time (s)		
libgis	3119	9113	3239	4524	1	49
libdbmi	77	509	83	190	7	37
libvect	195	96	198	41	3	43

and to reduce the memory requirements using dynamically-loadable libraries, or to obtain independent clusters, which may not always conceptually group objects as related as expected. Although it is counterintuitive, the latter result is not surprising, since experts classified functions according to the intended purpose or semantic. This seldom ensure high cohesion and low coupling, because the improvement of the latter attributes produces a final partitioning which somehow differs from what it was expected.

The addition of hill climbing into the SRGA did not improve the fitness function, since the SRGA also converged to similar results, when it was executed on an increased number of generations and increased population size. Noticeably, performing hill climbing on the best individuals of each generation produced a drastic reduction of convergence times. Comparing both strategies when the difference between values of the fitness function was below 10% highlighted that a hybrid strategy allowed on average to reduce the execution time of 43%. Convergence times for a *Compaq Proliant*™ with Dual Xeon™ 900 MHz processor, 2 MB Cache and 4 GB of RAM are reported in Table 6.

6.5. Extraction of new libraries

To identify new candidate libraries, the final step of the SRF is devoted to the analysis of the *Use Graph* obtained by subtracting the already existing libraries. Sometimes there are groups of objects used by a common set of applications, but they have not yet been organized into libraries. Clustering was performed on objects used by, at least, two applications.

Results revealed the presence of four clusters which were all located in the *orthophoto* subsystem. The number of dependencies between clusters was low and it was possible to solve them simply moving a couple of objects between clusters. Besides, all clusters had a considerable number of dependencies to external objects which belonged to the same set of applications. To eliminate these dependencies, it would have been necessary to increase the size of each cluster by 100%, clearly in contradiction with respect to the intended objective of reducing applications' memory requirements and size. Consequently, it was decided not to cluster these objects into libraries. In the authors' opinion, this is not a negative result, but it constitutes a quality indicator of the system showing that developers had carefully created and maintained libraries.

7. Conclusions

This paper has presented a framework for software system renovation (SRF) and the results of its application to *GRASS* Geographical Information System, which is over one million LOCs in size.

The SRF has allowed us to remove several structural problems from *GRASS*. In particular, unused objects were identified and factored out; clones were identified and, especially for those inside libraries, refactoring was performed. The SRF incorporates a novel library refactoring process, in which a suboptimal solution is first identified by hierarchical clustering and then refined by the SRGA. The proposed SRGA fitness function takes into account different factors: minimizing the number of dependencies, the average number of objects linked by each application, and the feedback of developers. Although the approach has been applied on C and C++ systems (*GRASS* and others reported by Antoniol et al., 2003), it is not tied to any specific programming language, provided that object modules, which contain a list of defined and required symbols, be available. However, for applications to be executed on a virtual machine, such as Java, Smalltalk programs, other approaches such as those of Tip et al. (1999) and Rayside and Kontogiannis (2002) may be preferable.

Overall, the SRF helps to monitor and improve the quality of a software system, which tends inevitably to deteriorate during the evolution. Unused objects, clones, library coupling, library sizes, and poor object organization are in fact significant quality indicators. For instance, the absence of new libraries identified by the SRF in *GRASS* indicates a careful design and a controlled evolution. Moreover, the SRF also addresses the miniaturization problem, which is relevant to port applications on limited-resource devices. The SRF has allowed us to reduce *GRASS* memory requirements and to improve its performance. The average number of library objects linked by each application was indeed reduced of about 50%. At the time of writing, *GRASS* has successfully been ported on a PDA (i.e., a CompaQ iPAQ). Given the size of the application and the available resources, a brute force automatic approach would not be feasible, since developers' suggestions were an essential component for the miniaturization process.

Clone detection performed on *GRASS* revealed that the cloning level outside libraries was not negligible and suggested further clone refactoring. Besides, the cloning level inside libraries was in general low, except for the mentioned cases. The cloning between libraries and the rest of the system was in most cases due to third party applications. Most of the system reorganization work described in the paper was incorporated in the subsequent releases of *GRASS* by removing unused objects and some clones, and by reorganizing some libraries. The latter reorganization, as pointed out in the paper, was carried out with minor modifications with respect to the result of the SRF.

Our in-progress work is devoted to investigate the feasibility of integrating other sources of knowledge into the SRF with special regards to dynamic information and in-field user profiles (Antoniol and Di Penta, 2003), obtained by instrumenting the source code.

Acknowledgments

We are grateful to the *GRASS* development team for the support, the information provided, and the feedback on the refactored artifacts. Giuliano Antoniol and Massimiliano Di Penta were partially supported by the ASI grant I/R/091/00. Markus Neteler was partially supported by the FUR-PAT Project WEBFAQ. Ettore Merlo was partially supported by the National Sciences and Engineering Research Council of Canada (NSERC).

References

Anderberg, M.R., 1973. Cluster Analysis for Applications. Academic Press Inc.

Anquetil, N., 2000. A comparison of graphs of concept for reverse engineering. In: Proceedings of the IEEE International Workshop on Program Comprehension. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 231–240.

Anquetil, N., Lethbridge, T., 1998. Extracting concepts from file names; a new file clustering criterion. In: Proceedings of the International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 84–93.

Antoniol, G., Di Penta, M., 2003. Library miniaturization using static and dynamic information. In: Proceedings of IEEE International Conference on Software Maintenance, Amsterdam, The Netherlands. pp. 235–244.

Antoniol, G., Casazza, G., Di Penta, M., Merlo, E., 2001a. A method to re-organize legacy systems via concept analysis. In: Proceedings of the IEEE International Workshop on Program Comprehension. IEEE Computer Society Press, Los Alamitos, CA, USA, Toronto, ON, Canada, pp. 281–290.

Antoniol, G., Casazza, G., Di Penta, M., Merlo, E., 2001b. Modeling clones evolution through time series. In: Proceedings of IEEE International Conference on Software Maintenance. pp. 273–280.

Antoniol, G., Villano, U., Merlo, E., Di Penta, M., 2002. Analyzing cloning evolution in the Linux Kernel. In: SCAM 2002 Special Issue. Information and Software Technology 44, 755–765.

Antoniol, G., Di Penta, M., Neteler, M., 2003. Moving to smaller libraries via clustering and genetic algorithms. In: European Conference on Software Maintenance and Reengineering. IEEE Computer Society Press, Los Alamitos, CA, USA, Benevento, Italy, pp. 307–316.

Bui, T.N., Moon, B.R., 1996. Genetic algorithm and graph partitioning. IEEE Transactions on Computers 45 (7), 841–855.

Cordy, J., 2003. Comprehending reality—practical barriers to industrial adoption of software maintenance automation. In: Proceedings of the IEEE International Workshop on Program Comprehension, Portland, OR, USA. pp. 196–205.

Deb, K., 1999. Multi-objective genetic algorithms: problem difficulties and construction of test problems. Evolutionary Computation 7 (3), 205–230.

Di Penta, M., Neteler, M., Antoniol, G., Merlo, E., 2002. Knowledge-based library re-factoring for an open source project. In: Proceedings of IEEE Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, Richmond, VA, pp. 128–137.

Doval, D., Mancoridis, S., Mitchell, B., 1999. Automatic clustering of software systems using a genetic algorithm. In: Software Technology and Engineering Practice (STEP), Pittsburgh, PA. pp. 73–91.

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

Goldberg, D.E., 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Pub. Co.

Gordon, A.., 1988. Classification, 2nd ed. Chapman and Hall, London.

Harman, M., Hierons, R., Proctor, M., 2002. A new representation and crossover operator for search-based optimization of software modularization. In: AAAI Genetic and Evolutionary Computation COnference (GECCO). Springer-Verlag, New York, USA, pp. 82–87.

Kaufman, L., Rousseeuw, P., 1990. Finding Groups in Data: An Introduction to Cluster Analysis. Wiley-Inter Science, Wiley, NY.

Krone, M., Snelting, G., 1994. On the inference of configuration structures from source code. In: Proceedings of the 16th International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, Sorrento, Italy, pp. 49–57.

Kuipers, T., Moonen, L., 2000. Types and concept analysis for legacy systems. In: Proceedings of the IEEE International Workshop on Program Comprehension. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 221–230.

Kuipers, T., van Deursen, A., 1999. Identifying objects using cluster and concept analysis. In: Proceedings of the International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 246–255.

Lehman, M.M., Belady, L.A., 1985. Software Evolution—Processes of Software Change. Academic Press, London.

Mahdavi, K., Harman, M., Hierons, R.M., 2003. A multiple hill climbing approach to software module clustering. In: Proceedings of IEEE International Conference on Software Maintenance, Amsterdam, The Netherlands. pp. 315–324.

Maini, H., Mehrotra, K., Mohan, C., Ranka, S., 1994. Knowledge-based nonuniform crossover. In: IEEE World Congress on Computational Intelligence. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 22–27.

Mancoridis, S., Mitchell, B.S., Rorres, C., Chen, Y., Gansner, E.R., 1998. Using automatic clustering to produce high-level system organizations of source code. In: Proceedings of the IEEE International Workshop on Program Comprehension. IEEE Computer Society Press, Los Alamitos, CA, USA.

Merlo, E., McAdam, I., De Mori, R., 1993. Source code informal information analysis using connectionist model. In: Proceedings of the International Joint Conference on Artificial Intelligence. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 1339–1344.

Mitchell, M., 1996. An Introduction to Genetic Algorithms. MIT Press, Cambridge, MA, USA.

Neteler, M. (Ed.), 2001. GRASS 5.0 Programmer's Manual. Geographic Resources Analysis Support System. ITC-irst, Italy, Available from: <http://grass.itc.it/grassdevel.html>.

Neteler, M., Mitasova, H., 2002. Open Source CIS: A GRASS CIS Approach. Kluwer Academic Publishers, Boston/USA; Dordrecht/Holland; London/UK.

Oommen, B., de St Croix, E., 1996. Graph partitioning using learning automata. IEEE Transactions on Computers 45 (2), 195–208.

Rayside, D., Kontogiannis, K., 2002. Extracting Java library subsets for deployment on embedded systems. Science of Computer Programming 45 (2–3), 245–270.

Shazely, S., Baraka, H., Abdel-Wahab, A., 1998. Solving graph partitioning problem using genetic algorithms. In: Midwest Symposium on Circuits and Systems. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 302–305.

Siff, M., Reps, T., 1999. Identifying modules via concept analysis. IEEE Transactions on Software Engineering 25, 749–768.

Snelting, G., 2000. Software reengineering based on concept lattices. In: Proceedings of IEEE International Conference on Software Maintenance. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 3–10.

Talbi, E., Bessière, P., 1991. A parallel genetic algorithm for the graph partitioning problem. In: ACM International Conference on Supercomputing. ACM Press, New York, USA, Cologne, Germany.

Tip, F., Laffra, C., Sweeney, P.F., Streeter, D., 1999. Practical experience with an application extractor for Java. ACM SIGPLAN Notices 34 (10), 292–305.

Tonella, P., 2001. Concept analysis for module restructuring. IEEE Transactions on Software Engineering 27 (4), 351–363.

Tzerpos, V., Holt, R.C., 1998. Software botryology: automatic clustering of software systems. In: DEXA Workshop. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 811–818.

Tzerpos, V., Holt, R.C., 1999. MoJo: A distance metric for software clusterings. In: Proceedings of IEEE Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 187–195.

Tzerpos, V., Holt, R.C., 2000a. ACDC: An algorithm for comprehension-driven clustering. In: Proceedings of IEEE Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 258–267.

Tzerpos, V., Holt, R., 2000b. The stability of software clustering algorithms. In: Proceedings of the IEEE International Workshop on Program Comprehension. IEEE Computer Society Press, Los Alamitos, CA, USA.

Wiggerts, T.A., 1997. Using clustering algorithms in legacy systems remodularization. In: Proceedings of IEEE Working Conference on Reverse Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA.

Massimiliano Di Penta received his laurea degree in Computer Engineering in 1999 and his PhD in Computer Science Engineering in 2003 at the University of Sannio in Benevento, Italy. Currently he is with RCOST—Research Centre On Software Technology in the same University. His main research interests include software maintenance, software quality, reverse engineering, program comprehension and search-based software engineering. He is author of about 30 papers appeared in international journals, conferences and workshops. He serves the program and organizing committees of workshops and conferences in the software maintenance field, such as the International Conference on Software Maintenance, the International Workshop on Program Comprehension, the Workshop on Source code Analysis and Manipulation.

Markus Neteler received his M.Sc. degree in Physical Geography and Landscape Ecology from the University of Hanover in Germany in 1999. He worked at the Institute of Geography as Research Scientist and teaching associate for two years. Since 2001 he is researcher at ITC-irst (Centre for Scientific and Technological research), Trento, Italy since 2001. His main research interest is remote sensing for environmental risk assessment and Free Software GIS development. He is author of two books on the Open Source Geographical Information System GRASS and various papers applications in GIS.

Giuliano Antoniol received his doctoral degree in Electronic Engineering from the University of Padua in 1982. He worked at Irst for 10 years were he led the Irst Program Understanding and Reverse Engineering (PURE) Project team. Giuliano Antoniol published more than 60 papers in journals and international conferences. He served as a member of the Program Committee of international conferences and workshops such as the International Conference on Software Maintenance, the International Workshop on Program Comprehension, the International Symposium on Software Metrics. He is presently member of the Editorial Board of the Journal Software Testing Verification & Reliability, the Journal Information and Software Technology, the Empirical Software Engineering and the Journal of Software Quality. He is currently Associate Professor the University of Sannio, Faculty of Engineering, where he works in the area of software metrics, process modeling, software evolution and maintenance.

Ettore Merlo received his Ph.D. in computer science from McGill University (Montreal) in 1989 and his Laurea degree—summa cum laude—from University of Turin (Italy) in 1983. He has been the lead researcher of the software engineering group at Computer Research Institute of Montreal (CRIM) until 1993 when he joined Ecole Polytechnique de Montreal where he is currently an associate professor. His research interests are in software analysis, software reengineering, user interfaces, software maintenance, artificial intelligence and bio-informatics. He has collaborated with several industries and research centers in particular on software reengineering, clone detection, software quality assessment, software evolution analysis, testing, architectural reverse engineering and bio-informatics.