

# A Language-Independent Framework for Software Miniaturization

M. Di Penta\*, M. Neteler\*\*, G. Antoniol\*, E. Merlo\*\*\*

dipenta@unisannio.it, neteler@itc.it, antoniol@ieee.org, merlo@polymtl.ca

\*RCOST - Research Centre on Software Technology

University of Sannio, Department of Engineering

Palazzo Bosco Lucarelli, Piazza Roma, 82100 Benevento, Italy

\*\*ITC-irst Istituto Trentino Cultura

Via Sommarive 18

38050 Povo (Trento), Italy

\*\*\*École Polytechnique de Montréal

Montréal, Canada

## Abstract

*One of the undesired effects of software evolution is the proliferation of unused components, or components unlikely to be used by a given subset of the applications. As a consequence, the size of binaries and libraries tends to grow. One of the major trends of today's software market is the porting of applications on hand-held devices or, in general, on devices having a limited amount of resources available. Several forms of refactoring and, in particular, the miniaturization of libraries and applications, are therefore necessary.*

*We propose a framework and a toolkit covering several aspects of software miniaturization, such as removing unused objects and code clones, as well as creating small, cohesive libraries refactoring the existing ones. The last step has been implemented using a hybrid approach based on hierarchical clustering, genetic algorithms and hill climbing, also incorporating the developer's knowledge. Most of the framework activities are language independent (relying on object module analysis) thus they do not require any kind of source code parsing and are applicable to software systems developed with*

*different programming languages.*

*The proposed framework has been applied to GRASS, an over 1 million LOCs open source Geographical Information System, reducing of 50% the average number of objects linked by each application, and thus application's memory requirements.*

**Keywords: refactoring, software miniaturization, clustering, genetic algorithms, hill climbing**

## **1. Introduction**

Porting existing software applications on hand-held devices, such as Personal Digital Assistants (PDA), or on wireless devices (e.g., multimedia cellphones) is one of the new software trends and industry hip. Wireless communication based on GSM, iDEN, GPRS (part of the so called 2.5G technology [1]) will be further ameliorated by the new third generation protocols promising higher bandwidths. Moreover, the 2.5G technologies offer enormous possibilities; as an example, GPRS provides up to 171 kbits/sec.; TCP/IP (PPP) over GPRS allows to connect via cellular phone a laptop or a PDA to information providers, exchange data, download music, images, update databases, or multimedia documents.

Although wireless technology has reached such a turning point, where complex applications can be ported on small devices, resource available is often limited, if compared to those for which the applications were conceived. To go wireless applications face the miniaturization challenge: the extra fat must be eliminated to ensure space and bandwidth optimization. Clearly, several actions may be taken. First and foremost, dead code should be removed and clones should be factored out. Furthermore, some form of restructuring, at library and at object file level, may be required. The latest intervention must deal with dependencies among software artifacts.

For any given software system, dependencies among executables and object files may be represented via a dependency graph, a graph where nodes represent resources and edges the resource dependencies. Each library, in turn, may be thought of as a sub-graph 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 [2] and thus often heuristics have been adopted to find a sub-optimal solution. For example, one may be interested to first examine graph partitions minimizing cross edges between sub-graphs corresponding to libraries. More formally, a cost function describing the restructuring problem has to be defined, and heuristics driving the solution search process must be identified and applied.

In [3], a process to miniaturize software systems has been proposed. The central idea is to apply clustering techniques to identify software libraries minimizing the average executable size. Doval et al. [4], applied Genetic Algorithms (GAs) to find what they called *meaningful partitions* in a graph representing dependencies among software components. Other communities, such as the optimization community, addressed the graph partitioning related problems in several ways. For example, constraints were incorporated by modifying the problem definition. To speed up the search process, heuristics based on GAs and modified GAs [5] were proposed. Performance improvement was also achieved by means of a hybrid approach, obtained combining a GA with hill climbing techniques.

Our framework stems from the observation that previously proposed approaches to software miniaturization were not completely satisfactory. For example, it is not obvious if pruning clones may be beneficial to reduce the memory requirements of executables. Moreover, a library refactoring approach based solely on clustering may be unable to find solutions easily identified by GA. Conversely, GA requires a starting population; choosing a random solution may not be very effective, or it may lead to a local sub-optimal solution. To overcome the aforementioned limitations, we propose a novel approach where an initial sub-optimal solution of library refactoring (i.e., a set of graph partitions) is determined via traditional clustering approaches, followed by a GA search aimed at reducing the inter-library dependencies. GA is applied to a newly defined problem encoding, where genetic mutation may lead sometimes to generate clones, clones that do indeed reduce the overall amount of resources required by the executables, in that they remove inter-library dependencies.

Moreover, our framework exploits a multi-objective fitness function, trying to keep low, at the same time, both the number of inter-library dependencies and the average number of objects linked by each application. Finally, the fitness function keeps into account the expert suggestions, leading to a semi-automatic approach composed by multiple iterations of the refactoring algorithm, interleaved by expert feedback.

In summary, this paper proposes a framework for software system miniaturization, composed by the different kinds of above mentioned activities: removing unused objects and clones, minimizing dependencies, moving to smaller libraries, and identifying new libraries. The main advantage of our framework is its language independence: all activities (except clone detection) rely on information extracted from object files; furthermore the clone detection algorithm adopted is not tied to specific programming languages (provided that a set of metrics can be extracted from the source code).

The framework has been applied to a large (517 applications and 43 libraries, for a total of over 1 million LOCs) Open Source software system: a Geographical Information System (GIS) named *GRASS* (Geographic Resources Analysis Support

System, <http://grass.itc.it>). *GRASS* is a raster/vector GIS combined with integrated image processing and data visualization subsystems [6]. The current *GRASS* development model can be considered as “council type”. The number of team members is small (7-15 active developers), decisions are usually taken by the members most capable of a certain problem. The developers are also users of the system, often focusing on their needs within the general project framework.

The paper is organized as follows. First a short review on related work (Section 2) and on main notions of clustering and GA (Section 3), will be presented. Then, the refactoring framework is described 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, before conclusions and work-in-progress.

## 2. Related Work

Many works are reported in literature concerning with software system modules clustering and/or restructuring, identifying objects, and recovering or building libraries. Most of these works applied clustering or concept analysis (CA).

An overview of CA applied to software reengineering problems was shown by G. Snelting in his seminal work [7], where he used CA in several re-modularization problems such as exploring configuration spaces (see also [8]), transforming class hierarchies, and re-modularizing COBOL systems. In [9] Kuipers and Moonen combined CA and type inference in a semi-automatic approach to find objects in Cobol legacy code. In [10] authors proposed the idea of recovering libraries and creating a source file directory structure using CA. As in [8, 9, 10, 11] we believe that with the present level of technology a programmer-centric approach is required: the user is left in charge to choose the proper re-modularization based on his/her knowledge.

A comparison between clustering and CA was presented in [11]. Our work shares with [11] the idea of applying an agglomerative-nesting clustering to a Boolean usage matrix, although in [11] the matrix indicated the uses of variables by programs.

A survey of clustering techniques applied on software engineering was presented by Tzerpos and Holt in [12]. The same authors presented in [13] a metric to evaluate the similarity of different decompositions of software systems, in [14] a clustering algorithm oriented to program comprehension, and they discussed in [15] the problem of stability of software clustering algorithms. Another overview of cluster analysis applied on software systems has been presented in [16].

Applications of clustering to re-engineering were suggested in [17] and [18]. In [17] a method for decomposing complex

software systems into independent subsystems was proposed by Anquetil and Lethbridge. Source files were clustered according to file names and their decomposition. Merlo et al. [18] exploited comments, as well as variable and function names to cluster files.

An approach relying on inter-module and intra-module dependency graphs to refactor software systems was presented in [19]: we share with [19] the idea of analyzing dependency graphs, finding a tradeoff between having highly cohesive libraries and a low inter-connectivity.

GAs have been recently applied in different fields of computer science and software engineering. An approach for partitioning a graph using GA was discussed in [5]. Similar approaches were also shown in [20, 21, 22]. Maini et al. [23] discussed a method to introduce the problem knowledge in a non-uniform crossover operator, and presented some examples of its application. GA was used by Doval et al. [4] to identify clusters on software systems. We share with [4] the idea of a software clustering approach using GA, trying to minimize inter-cluster dependencies. Finally, Harman et al. [24] reported experiments of modularization/remodularization, comparing GA with hill climbing techniques, and introducing a representation and a crossover operator tied to the remodularization problem. Their case studies revealed that hill climbing outperformed GA.

Preliminary results from the *GRASS* refactoring were proposed in [3], where several activities were carried out in order to refactor *GRASS* libraries. In particular, unused symbols were identified and pruned, clones were removed, and a preliminary work aimed at splitting the largest libraries was performed. The library refactoring process was then improved in [25], determining the optimal number of clusters with the Silhouette statistics and minimizing the number of inter-cluster dependencies using GA. While commonalities can be found with previous works, this paper contributes with a new framework encoding developer's knowledge in the GA fitness function, and improving refactoring performance via a hybrid approach based on both GA and hill climbing.

### 3. Background notions

The fundamental activity of our framework is the library refactoring. This requires to integrate 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 [26, 27, 28]. 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 [29].

In the following sub-sections, for sake of completeness, some essential notions are summarized. Describing the different types of clustering algorithms or the details of GA is out of the scope of this paper. More details can be found in [26, 27, 28] for clustering and in [29] for GA.

### 3.1. Clustering

In this paper, the agglomerative-nesting (*agnes*) algorithm [30] 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. In particular, agglomerative algorithms start building the dendrogram from the bottom of the hierarchy (where each one of the  $N$  entities represents a cluster), until at the  $N - 1$  level all entities are grouped in a single cluster.

The key point of hierarchical clustering is determining the *cut point*, i.e., the level to be considered in order to determine the actual clusters. As it will be shown in Section 3.2, in this work such operation was supported by the Silhouette statistics.

### 3.2. Determining the Optimal Number of Clusters

To determine the actual or optimal number of clusters, traditionally, people rely on the plot of an error measure representing the within cluster dispersion. The error measure decreases as the number of cluster  $k$  increases, but for some  $k$  the curve flattens. Traditionally, it is assumed that the error curve *elbow* indicates the appropriate number of clusters [31]. To overcome the limitation of such a heuristic approach, several methods have been proposed, see [31] for a comprehensive summary.

Kaufman and Rousseeuw [30] 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 (but its own), 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. Notice that the Silhouette statistics, as most of the methods described in [31], has the disadvantage that it is undefined for one cluster, and thus it offers no indication of whether the current dataset already represents a good cluster. Since our purpose is to split the original libraries into smaller ones, this does not constitute a problem.

### 3.3. Genetic Algorithms

GAs revealed their effectiveness in finding approximate solutions for problems where:

- The search space is large or complex;
- No mathematical analysis is available;
- Traditional search methods did not work; and, above all
- The problem is NP-complete or NP-hard [2, 5].

Roughly speaking, a GA may be defined as an iterative procedure that searches 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 kind 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, in that it randomly modifies an individual's genome (e.g., 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  $pmut \ll pcross$ .

The GA does not guarantee to converge: the termination condition is often specified as a maximal number of generations, or as a given value of the fitness function.

### 3.3.1 Hill Climbing and GA Hybrid Approaches

As suggested in literature [29], hybrid GAs may reveal advantageous when there is the need of optimization techniques tied to specific problem structure. The *in-large* perspective of GA may be combined with the precision of local search. GAs are able to explore a large search space, but often they reach a solution that is not accurate, or they converge to an accurate solution very slowly. On the other hand, local optimization techniques (such as hill climbing) quickly converge to a local optimum, but they are not very effective to search in large solution spaces (in that they suffer of problems such as local maximum or plateaus).

There are different ways to hybridize a GA with hill climbing techniques. The first approach tries to optimize, using hill climbing techniques, the best individuals of the last generation. 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 an *high quality individual*, obtained from the optimization, reducing therefore the number of generations requested to ensure GA convergence.

## 4. The Refactoring Framework

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

- First and foremost, software system applications, libraries and dependencies among them should be identified;
- Unused functions/objects should be identified and, if needed, removed from the system or, possibly, stored in an appropriate repository;
- Duplicated (cloned) objects must be identified and, whenever possible, factored out;
- Circular dependencies among libraries must be removed, or, at least, minimized. In fact, these dependencies cause a library to be linked each time another one (circularly linked to it) is needed;
- Large libraries should be refactored into smaller ones and, if possible, transformed into dynamic libraries; and
- Objects used by multiple applications (not yet organized into libraries) should be grouped into new libraries.

The framework activities and the representations used are detailed in the following subsections.

## 4.1 Software System Graph Representation

Central to our framework is the software system representation; most of the computation activities rely on a graph highlighting dependencies between object modules  $O \equiv \{o_1, o_2, \dots, o_p\}$ . The software system may be, in fact, represented by the *System Graph* ( $SG$ ) defined as:

$$SG \equiv \{O, D\} \quad (2)$$

where  $D \subseteq O \times O$  the set of oriented edges  $d_{i,j}$  representing dependencies between objects. An example of  $SG$  graph is depicted in Figure 1. Nodes of the  $SG$  graph may be classified in two categories:

1. Source<sup>1</sup> nodes containing the `main` symbol, drawn in Figure 1 as squares, representing the set of the  $m$  software system applications  $A \equiv \{a_1, a_2, \dots, a_m\}$ ; and
2. Other nodes, indicated as circles, representing objects needed by applications. Some of these objects may be contained into libraries (depicted as rounded boxes). The set of  $n$  system libraries is indicated as  $L \equiv \{l_1, l_2, \dots, l_n\}$ .

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

$$a_x \text{ uses } o_y \iff \exists \text{ path } \{a_x, \dots, o_y\} \in SD \quad (3)$$

$$l_x \text{ uses } o_y \iff \exists o_j \in l_x \mid \exists \text{ path } \{o_j, \dots, o_y\} \in SD \quad (4)$$

The *Use Graph* can be represented by a matrix of uses  $MU$ :

---

<sup>1</sup>It is worth noting that applications are not the only source nodes. In fact, as it will be detailed later, also unused objects satisfy this property.

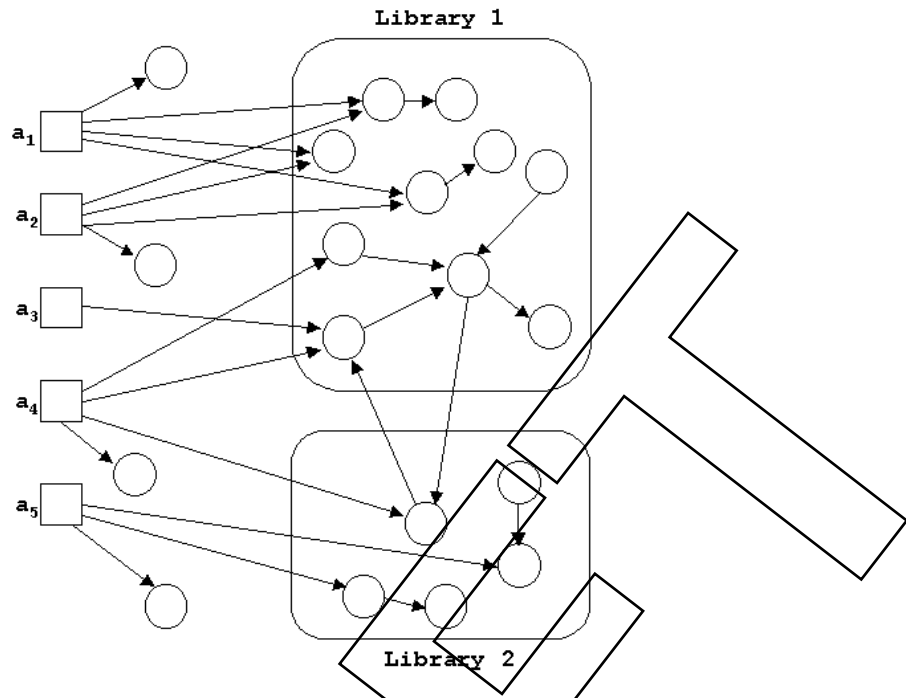
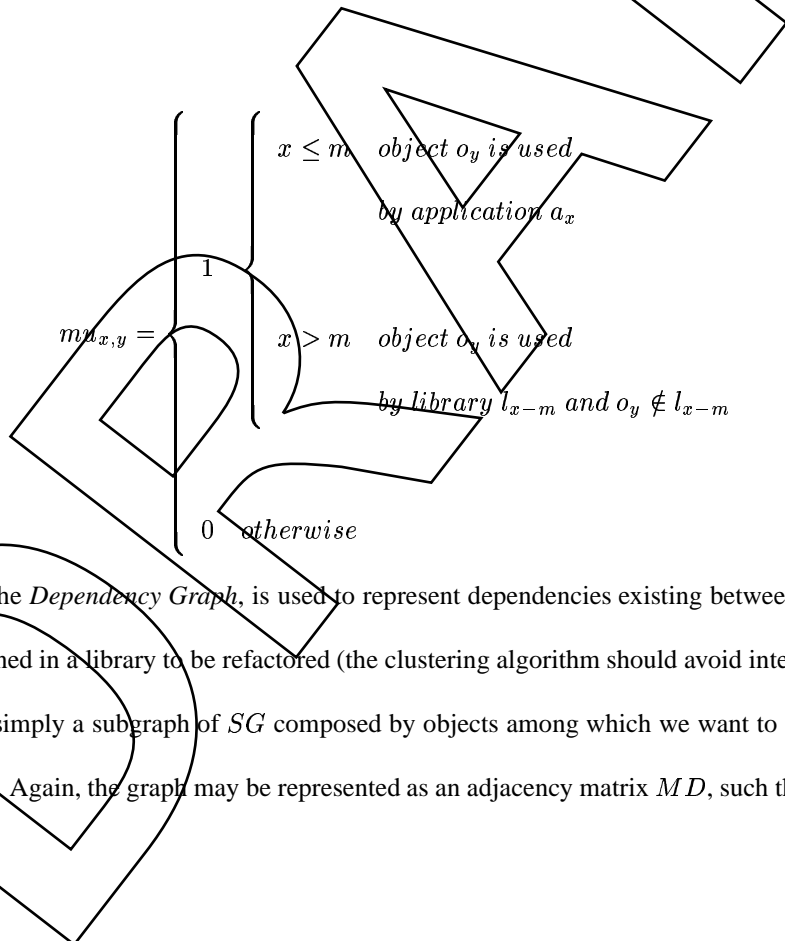


Figure 1. Example of System Graph.



The second graph, the *Dependency Graph*, is used to represent dependencies existing between two or more libraries, or between objects contained in a library to be refactored (the clustering algorithm should avoid inter-cluster dependencies). A *Dependency Graph* is simply a subgraph of *SG* composed by objects among which we want to analyze dependencies, and edges connecting them. Again, the graph may be represented as an adjacency matrix  $MD$ , such that:

$$md_{x,y} = \begin{cases} 1 & o_x \text{ depends from } o_y \\ 0 & \text{otherwise} \end{cases}$$

## 4.2 Graph Construction

Prior to recover dependencies among applications and libraries, and among libraries themselves, applications (i.e., executables) composing the software system must be identified. In this paper an approach similar to the one used in [10] was used. However, in [10] applications were identified detecting all source (.c) files containing the definition of a main function. We experienced that in *GRASS* there was not always a one-to-one correspondence between source and object files, therefore objects defining the main symbol were directly searched and mapped to applications.

Once applications and existing libraries were identified (for the latter the identification process was trivial, in that it consisted in simply searching for .a files), the *SG* graph was built. Given the *use* relation between an object module requiring a symbol and a module defining it, the graph was built via the *transitive closure* of the *use* relation, starting from the main object of each application and from each library. In other words, for each application, undefined symbols were identified and recursively (in that new undefined symbols were added to the stack) resolved firstly inside the objects contained in the same path (i.e., other modules of the application), then inside libraries. A similar process was performed to detect dependencies among libraries. Finally, the *use graph* and the *dependency graph* (and therefore the *MU* and *MD* matrices) were extracted from the *SG* graph.

## 4.3 Handling Unused Objects

Symbols defined in libraries, but not used by applications nor by other libraries, are likely to constitute useless resources, thus they should be identified. Their presence is often due to utility functions inserted in libraries (not used by current set of applications), or to features not yet fully implemented.

The objects defining these symbols should be removed from the libraries, providing that objects do not export also used symbols (in that case the object should be left into library, or the corresponding source file restructured). One possible refactoring solution is to create, from each library, two new libraries, one of which containing all the unused symbols.

In any case, even if developers decide to leave these objects in their position, maybe because they can potentially be used

by future applications (or new versions of old applications), their presence and their impact on the software system size should be highlighted.

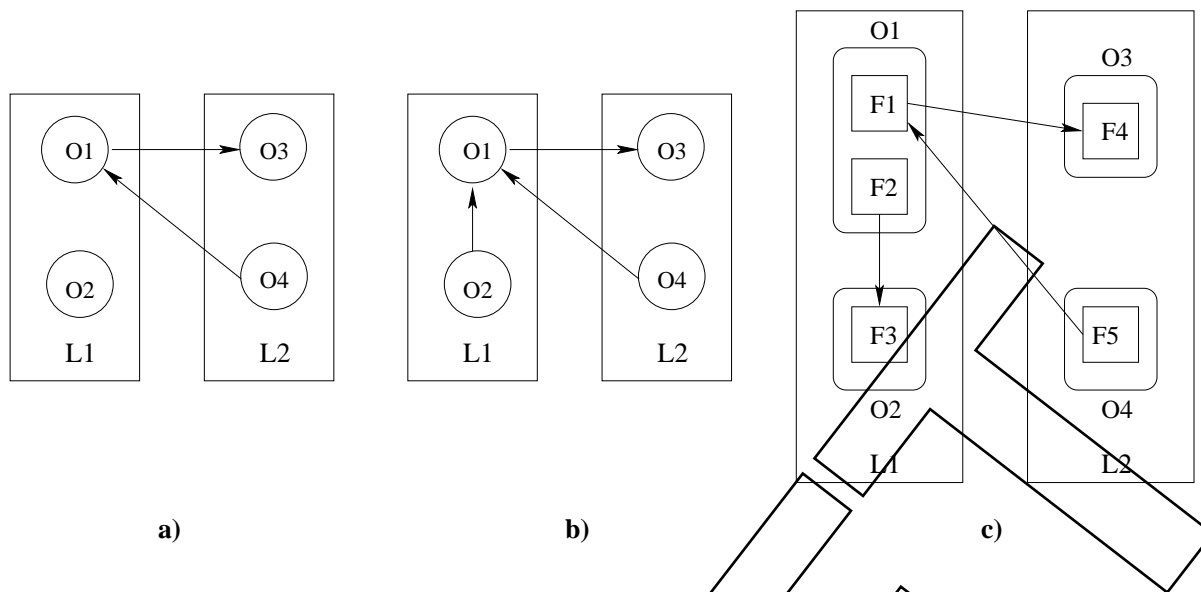
#### 4.4 Removal of Circular Dependencies among Libraries

*DG* captures dependencies among the different libraries and allows to identify 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 were identified, four choices could be taken to remove them:

1. *Move the object* causing the circular dependence from a library to another. This is only feasible if the object does not need resources located in its original library, nor it is needed by that library. For example, in Figure 2-a, object  $o_1$  can be moved from library  $L_1$  to library  $L_2$ ;
2. *Duplicate the object*: like the previous case, this is feasible if the object does not need resources located in the original library but, differently from the previous case, it is required in that library (therefore moving it outside worsen the situation). In Figure 2-b, object  $o_1$  should be duplicated in library  $L_2$  (it cannot be moved, in that  $o_2$  depends on it);
3. *Merge the two libraries*: this strategy should be avoided whenever possible; 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. *Make 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 (see details in Section 4.7.1).

When the *DG* do not allow to remove circular dependencies and, for performance reasons, options three and four cannot be adopted, a deeper analysis should be performed, identifying dependencies at function grain-level instead of object grain-level. This should ease the removal of some critical dependencies. Let us consider the example in Figure 2-c: object  $o_1$  cannot be moved, nor duplicated in  $L_2$  (in that it depends from the object  $o_2$ ). However, splitting  $o_1$  into two chunks, leaving the function  $f_2$  in library  $L_1$ , and moving  $f_1$  in  $L_2$ , would solve the problem.

Finally, the existence of a complex dependency relationship between two libraries indicates the possibility (to be confirmed by developer's feedback) of poor library design. In this case, library objects should be merged and then refactored again in new clusters, adopting the process detailed in Section 4.6.



**Figure 2. Examples of dependencies among libraries.**

#### 4.5 Identification of Duplicates Symbols and Clones

Comparing the list of symbols defined in each library allows detecting the list of duplicate exported symbols. It is worth noting that homonym symbols in different libraries may refer to completely different functions, external variable or data structures. On the other hand, two or more symbols, although having different names may correspond to cloned functions.

Therefore, a cloning analysis is necessary. In this paper a metric-based clone detection process [32], aiming at detecting duplicated functions, was adopted. The results obtained may 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;
2. If duplicated functions are identified inside different objects, refactoring could be performed moving them outside, applying the considerations similar to the previous case; and
3. Finally, clone detection may reveal clones outside libraries, in that several applications may contain, in their objects, duplicated portions of code. On some cases, it should be useful to factor such duplicated code in a library.

Preliminary to the above described actions is an analysis of the impact in terms of dependencies (and, above all, circular dependencies) introduced. 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 (if using DLLs) of one or more additional libraries.

## 4.6 Library Refactoring

The last, and most relevant point of the proposed process is devoted to split existing, large libraries into smaller archives, thus reducing the memory footprint of applications. Basically, the idea is similar to those proposed in [10] to identify libraries: objects used by a common set of programs should be grouped together, trying to minimize the average number of libraries required by each program.

In [10] a *concept lattice* was used to group objects into libraries: although the lattice gives useful information (often good libraries are the sets of objects located on top nodes, or concepts retaining large percentages of objects), it becomes unmanageable when a large number of applications and libraries (as in our case study) must be handled [33]. Instead of pruning information on *concept lattice* like [34, 35], a clustering analysis was performed, similarly to [17, 18, 19].

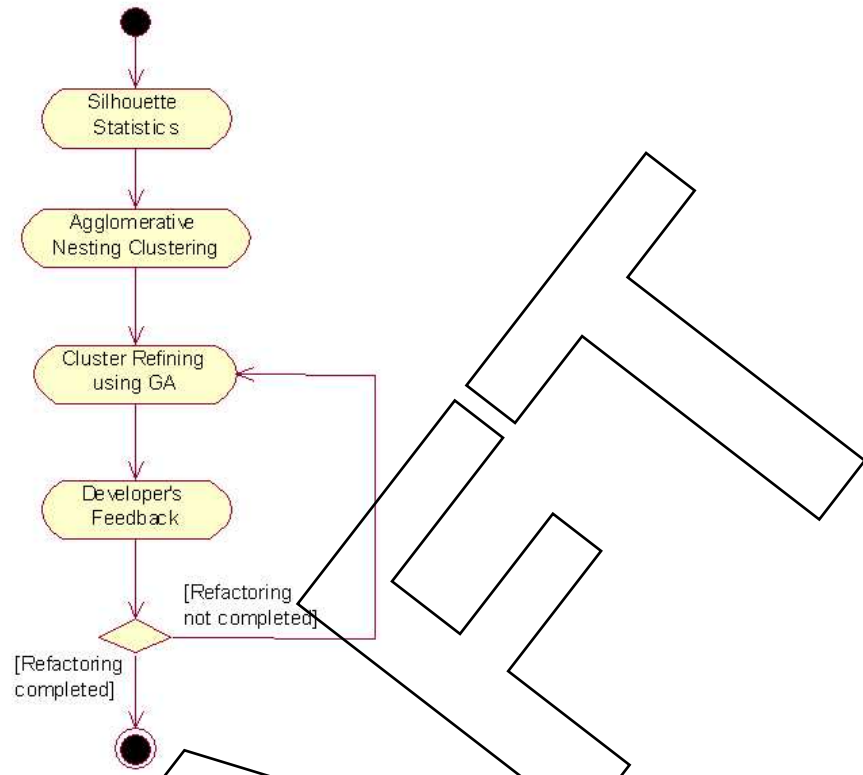
For each library  $l_x$  to refactor, we built a  $MU$  matrix from the subgraph of the *use graph* representing uses of  $l_x$  objects from applications (and other libraries), and a  $MD$  matrix representing dependencies between all library objects.

The library refactoring process, as shown in Figure 3, consists of four steps:

1. Determine optimal number of clusters;
2. Determine a sub-optimal refactoring by hierarchical clustering; and
3. Determine the new candidate libraries using GA;
4. Ask feedback to developers and, eventually, iterate through step 3.

### 4.6.1 Determining the Optimal Number of Clusters

As explained in Section 3.2, the optimal number of clusters was computed on each  $MU$  matrix inspecting the Silhouette statistics. Giving the curve of the average Silhouette values for different numbers  $k$  of clusters, instead of considering the maximum (often too high for our refactoring purpose), we chose for some libraries the knee of that curve [30] as optimal



**Figure 3. Activity diagram of the library refactoring process.**

number of clusters. We also incorporated in the choice experts' knowledge, and we considered a tradeoff between excessive fragmentation and library size. Examples of Silhouette statistics are shown in Figure 6.

#### **4.7. Determining the Sub-Optima Libraries by Hierarchical Clustering**

Once known the number of clusters for each “old library”, agglomerative-nesting clustering was performed on each  $MU$  matrix. This builds a *dendrogram* and a vector of heights, that allow identifying  $k$  clusters. These clusters are the new *candidate libraries*.

To assess the effectiveness of the refactoring process, a measure of quality for the new libraries should be defined. Let  $k$  the number of clusters  $l_{x_1}, \dots, l_{x_k}$  obtained from a library  $l_x$ . Then, the *Partitioning Ratio*  $PR_x$  can be defined as:

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

where  $|l_x|$  is the number of objects archived into library  $l_x$ . The smaller is the  $PR$ , the most effective is the partitioning, in that the average number of objects linked (or loaded) by each application is smaller than using the whole old library.

#### 4.7.1 Reducing Dependencies using Genetic Algorithms

The solution reached at the previous step presents two main drawbacks:

1. The number of dependencies between the new libraries could be high, forcing to load another library each time a symbol from that library is needed, therefore wasting the advantage of having new smaller libraries; and
2. New libraries may not be meaningful with respect to developer's intentions: their feedback has to be incorporated in the refactoring process.

Of course, as shown in [3], an important step to perform is to convert static libraries to dynamic-loadable libraries, so that each (small) library is loaded at run-time only when needed, and then unloaded when it is no longer useful. In this case, even if there are dependencies among libraries, the average number of libraries in memory is considerably reduced with respect to the original system.

The removal of inter-library dependencies can be brought back to a graph partitioning problem that, as shown in [5], is NP-hard, and a GA was used to reach an approximate solution of the problem (i.e., minimize the number of dependencies).

A GA requires the specification of:

1. The genome encoding;
2. The initial population;
3. The fitness function;
4. The crossover operator;
5. The mutation operator; and
6. All GA parameters, such as the crossover and mutation probability, the population size and the number of generations.

An approach of clustering functions using GA was discussed in [4]. However, as shown below, in our case the genome encoding, the initial population, the mutation operator and the fitness function are different.

The encoding schema widely adopted in literature [4, 5] indicates each partition with an integer  $p$  such that  $0 \leq p \leq k - 1$  (where  $k$  is the number of *candidate libraries*), and represents the genome as a  $|l_x|$ -size array  $GV$ , where the integer  $p$  in position  $q$  means that the object  $q$  is contained into partition  $p$ . However, our purpose is the reduction of memory requirements for each application, therefore sometimes “cloning” an object in different libraries may help reducing the number of linked libraries. Unfortunately, the encoding schema above mentioned does not allow an object to be contained in more than one library.

We therefore adopted a bit-matrix encoding, where the genome  $GM$  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” on the same column.

Instead of randomly generating the initial population (i.e., the initial libraries), the GA was initialized with the encoding of the set of libraries obtained in the previous step.

The fitness function was constructed 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 that, as said, should be as small as possible;
3. The size of the new libraries; and
4. The feedback given by the developers.

The first factor, the *Dependency Factor* ( $DF(g)$ ) was defined as:

$$DF(g) = \sum_{i=0}^{|l_x|-2} \sum_{j=i+1}^{|l_x|-1} md_{i,j} \delta(G[i], G[j]) \quad (6)$$

where

$$\delta(G[i], G[j]) = \begin{cases} 0 & G[i] = G[j] \\ 1 & G[i] \neq G[j] \end{cases}$$

The second factor is the *PR* shown in equation 5. The third factor, the *Standard Deviation Factor* ( $SF(g)$ ) can be thought of as the difference between the initial library sizes standard deviation and the actual (at the current generation) standard

deviation. Without taking into account the last item, it could happen that the GA, in the attempt to reduce dependencies, groups a large fraction of the objects in the same library, negatively affecting the  $PR$ . A similar factor was also applied in [5]. Given  $S_0$  the array of library sizes for the initial population, and  $S_g$  the same for the  $g$ -th generation:

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

The fourth factor keeps into account the developer feedback. After a first execution of GA, without considering this factor, developers were asked to provide a feedback on the proposed new libraries. The result of developer's feedback is a bit matrix  $FM$ , having the same structure of the genome matrix, and incorporating changes developers suggested with respect to libraries proposed by GA.

After this feedback, the GA is ran again keeping into account this time the *feedback factor*  $FF$ , accounting 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 each time there is a difference between the genome and the refactoring proposed by developers.

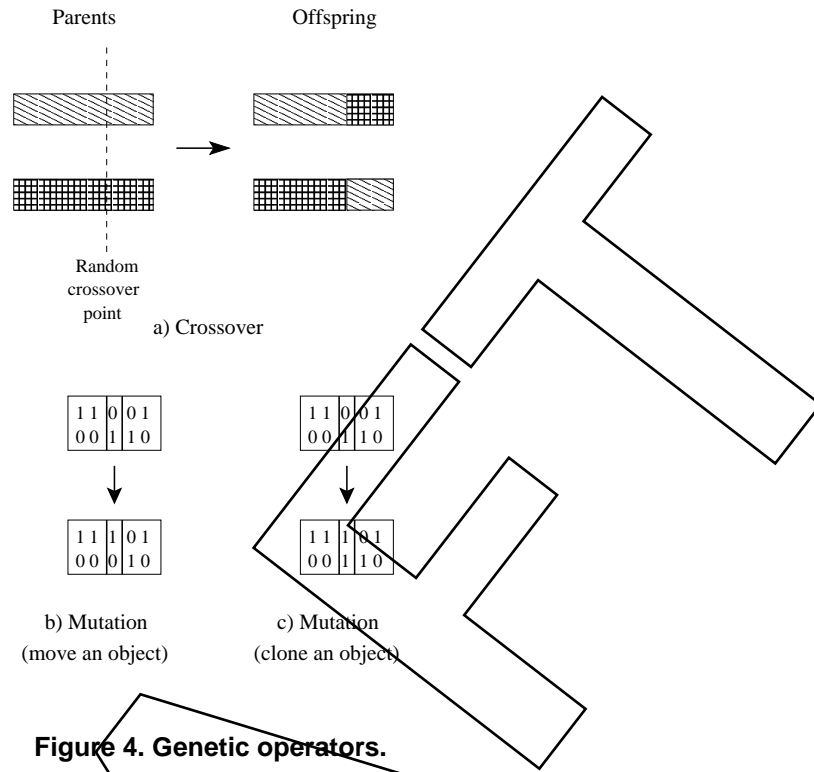
Overall, the fitness function  $F$  was defined as:

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

where  $w_1$ ,  $w_2$  and  $w_3$  are real, positive weighting factors for the  $PR$ ,  $SF$  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; on the other hand, increasing too much  $w_1$  decreases dependency reduction. Similarly, the higher is  $w_2$ , the more similar will be the result to the starting set of library, while an excessively higher  $w_2$  could not allow a satisfactory dependency reduction. Finally,  $w_3$  should be properly sized to weight the influence of developer's feedback. As explained, before asking developers for a feedback, a preliminary run of the GA must be performed with  $w_3 = 0$ .

As stated in (9), our fitness function is multi-objective [36, 37, 38]. Notice that, since we aimed to give maximum priority to dependency reduction, the  $DF$  weight was set to 1. Successively, we selected  $w_1$ ,  $w_2$  and  $w_3$  using a trial-and-error, iterative procedure, adjusting them each time until the  $DF$ ,  $PR$ ,  $SF$  and  $FF$  obtained at the final step were not satisfactory.

The process was guided by computing each time the average values for  $DF$ ,  $PR$ ,  $SF$ , and  $FF$ , and by plotting their evolution, to determine the 3D space region in which the population should evolve.



**Figure 4. Genetic operators.**

The crossover operator used in this paper is the *one point crossover*: given two matrices, both are cut at the same random column, and the two portions are exchanged (Figure 4a). The mutation operator works in two modes:

1. Normally, it takes a random column and randomly swaps two rows: this means that, if the two swapped bits are different then an object is moved from a library to another (Figure 4b); or
2. With probability  $p_{clone} < p_{mut}$ , 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 (Figure 4c).

Noticeably, the cloning of an object increases both  $LF$  and  $SF$ , therefore it must be minimized. Our GA activates the cloning only for the final part of the evolution (after 66% of generations in our case studies). Our strategy favors dependency minimization by moving objects between libraries; then, 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: if, for example, 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 from which  $l_i$  and  $l_j$  depend on.

Finally, we introduced the *Lock Matrix* as a further, stronger level of feedback: this matrix gives to developers the possibility of locking an object in a cluster, when they strongly believed the object should belong to that cluster.

Given this, if a column of the lock matrix contains at least a not-null item, then the mutation operator does not perform any action bringing a genome in a inconsistent state with respect to the *Lock Matrix*. In other words, if the lock matrix indicates that object  $j$  belongs to cluster  $i$ , any mutation removing object  $j$  from cluster  $i$  is avoided. It is worth noting that the crossover operator simply mixes columns from two genomes, thus it cannot perform any action in contrasts with respect to the *Lock Matrix*. Similarly, a *cloning* mutation duplicates an object, without removing it from the cluster(s) where it was locked, thus even in this case no particular inconsistency check is necessary.

The population size and the number of generations were chosen by an iterative procedure, doubling both each time until the obtained *DF*, *PR* and *FF* were equal to those obtained at the previous step.

GA suffers from slow convergence: to improve performances, the GA was hybridized with hill climbing techniques. As reported in Section 3.3.1, hill climbing may be applied on individuals of the last generation, or on the best individual(s) of each generation. We experienced that, for our optimization problem applying hill climbing only to the last generation does not significantly improve performances nor results. On the contrary, applying hill climbing to the best individuals of each generation makes the GA convergence significantly faster. In particular, hill climbing was applied by repeatedly moving a random object from its source cluster to another cluster, and accepting the change only if the new genome increased the fitness function.

#### 4.8 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 into any library. Their identification and organization into libraries should be therefore desirable. The factoring process is quite similar to that described in the previous section. In particular, a *MU* matrix is built on a subgraph of the *use graph* obtained removing all existing libraries from it. 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 GA and developer's feedback.

## 4.9 Tool Support

To support the refactoring process, different tools were needed, some of which already described in [3]. The following tools were conceived:

- *The application identifier* that, using the `nm` Unix tool, identifies the list of object modules containing the main symbol;
- *The graph extractor*, also based on the `nm` tool, that produces the *System Graph*, the *Use Graph*, and the *Dependency Graph*. The *graph extractor* also allow exporting data in `.DOT` format [39], to allow visualization and analysis using the `DotTy` [40] graph visualization tool;
- *The unused symbol identifier*: it produces, for each library, the list of the symbols (and, for each one, the object containing it) not used by any application or library;
- *The circular dependency identifier*: it produces the list of all circular path among libraries;
- *The duplicated symbol identifier*: it identifies the list of duplicated-defined external symbols. It is used in conjunction with the metric-based *clone detector* (see [32] for details) and with the *dependency graph extractor* to minimize the presence of clones inside libraries; and
- *The number of clusters identifier*, implementing the Silhouette statistics. In particular, implementations available in the *cluster* package of the *R Statistical Environment* [41, 42] were used;
- *The library refactoring tool*: it supports the process of splitting libraries in smaller clusters. The cluster analysis is performed by the *agnes* function available under the *cluster* package of the *R Statistical Environment*;
- *The GA library refiner*: implemented in C++ using the *GAlib* [43]; and
- *The developer's feedback collector*: it is a web application that allowed developer posting on our web site a feedback on the produced libraries.

## 5. Case Study

The modular structure of *GRASS* allows it to run with a very small memory overhead, therefore the hardware requirements are quite moderate. When running *GRASS* on a PC workstation or a notebook, standard equipment is generally sufficient.

Pre-existing libraries	43
Library objects	1056
Applications	517
C source files	7107
C KLOC	1014

**Table 1. GRASS key characteristics.**

The GRASS CVS development snapshot of April 5, 2002, downloadable from <http://grass.itc.it> was used as a case study. Its characteristics are summarized in Table 1.

Supported platforms at the date of writing comprise Linux/PC, SUN, HP/UX, MacOSX, MS-Windows/Cygwin, IPAQ/Linux and others. For geospatial data, more RAM is generally more effective than a faster CPU. GRASS modules (commands) are organized by name, based on their function class (display, general, imagery, raster, vector or site, etc.). The first letter refers to the function class, followed by a dot and one or two other words, again separated by dots, describing the specific task performed by the module.

GRASS modules are invoked within a shell environment (or from the graphical user interface). The GRASS parser is a collection of subroutines which allow the programmer to define options (parameters) and flags that make up the valid command line input of a GRASS command. The parser routines behave in the following ways: If no command line arguments are entered by the user, the parser searches for a completely interactive version of the command which may differ from the command line version. If the interactive version is found, control is passed over to this version. If not, the parser will prompt the user for all programmer-defined options and flags. If all necessary options and flags are entered on the command line by the user, the parser executes the command with the given options and flags, otherwise it will pass an error message to the user indicating which required options and/or flags were missing from the command line and cancel execution of the command.

The GRASS modules are linked against an internal "front.end". The "front.end" module will call the interactive version of the command if there are no command-line arguments entered by the user. Otherwise, it will run the command-line version. If only one version of the specific command exists (for example, if there is only a command-line version available) the existing 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 utilized in the *GRASS* modules, from reading and writing maps to area and distance calculations for georeferenced data as well as attribute handling and map visualization. Details of *GRASS* programming are covered in the “*GRASS 5.0 Programmer’s Manual*” [44]. This programming API are organized as follows (typical function name prefixes for related library functions are listed in squared brackets):

- GIS library: database routines (*GRASS* file management), memory management, parser (parameter identification on command line), projections, raster data management etc. [G\_], e.g. `G_read_raster_row()`;
- vector library: management of area, line, and point vector data [Vect\_, V2\_, dig\_], e.g. `V2_read_line()`;
- image data library: image processing file management [I\_], e.g. `I_georef()`;
- site data library: site data management [G\_sites\_], e.g. `G_site_new_struct()`;
- display library: graphical output to the monitor [D\_], e.g. `D_new_window()`;
- raster graphics library: display raster graphics on devices [R\_], e.g. `R_open_driver()`;
- segment library: segmented data management [segment\_], e.g. `segment_get()`;
- vask library: control of cursor keys etc. [V\_], e.g. `V_ques()`;
- rowio library: for parallel row analysis of raster data [rowio\_], e.g. `rowio_get()`.

## 6. Case Study Results

This section presents the results obtained applying to *GRASS* the refactoring framework described in Section 4.

### 6.1 Handling Unused Objects

Out of 921 objects composing libraries, 89 were not used by any application, nor by other libraries. According to our framework, when refactoring libraries, those objects will be organized into a separate cluster, thought of as a sort of repository to be “frozen” for future uses. A deeper analysis revealed that some functions contained into unused objects wrap lower level *GRASS* functions (e.g., `db_create_index`) standard library/system call functions (e.g., `scan_dbl`, `scan_int`,

whoami), and, in general, provide some simple functionalities using lower level functions (e.g., `datetime_is_same`, that compares two `DateTime` structures). An interesting example (see also Section 6.4) is the library `libdbmi`: out of 97 objects, 19 were not used at all. In all cases, the unused functions correspond 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, (contained in `libdbmi.a`). On the contrary, `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 from `libcoorcnv.a`, symbols located in the module `datum.o` (while the inverse dependency involved 13 symbols). Moving `datum.o` into `libgis.a` resolved the problem.

Finally, circular dependencies were found between `libvect.a` and `libdig2.a`. It involved 13 symbols in a direction, 31 in the other, symbols located in several different objects. The links present in the dependency graph excluded the possibility of resolving circular dependencies simply moving (or duplicating) objects. The decision taken (supported from system's developers) was to initially merge the two libraries (in effect designed to work together) and then try to refactor the new library (see Section 6.4).

## 6.3 Identification of Clones

The search for duplicated symbols followed by 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 refactoring and miniaturization; in the second case, the objective was to identify portions of duplicated code that can be potentially organized into new libraries.

Table 2 reports results obtained from clone analysis: the total number of functions analyzed, the number of clone clusters [45] detected, the number and the percentage of cloned functions. Finally, clones were computed filtering out the shortest

functions: for example, two functions that simply return a value should not be considered as clones. Results were presented considering two thresholds: functions longer than five and ten LOCs.

	Total # of Functions	# of Clone Clusters	# of Cloned Functions	% of Cloned Functions	Threshold (LOCs)
Overall	22229	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	16958	1817	4974	29.33%	5
		1290	3268	19.27%	10
Libraries vs. Outside	22229	130	635	2.86%	5
		73	272	1.22%	10

**Table 2. Results of Clone Detection.**

As shown, the overall percentage of clones is not negligible (26%), even considering only functions longer than five LOCs (16.38%). Data in Table 2 suggests a potential reduction up to 17% in the number of the functions; clearly, the actual reduction rate will be lower because of *false positives*. The number of clones contained inside libraries is low, indicating that developer factored functions/objects accurately, avoiding duplicates. Finally, we investigated the set of clones between libraries and objects outside libraries, a situation where there could often be the possibility of a refactoring.

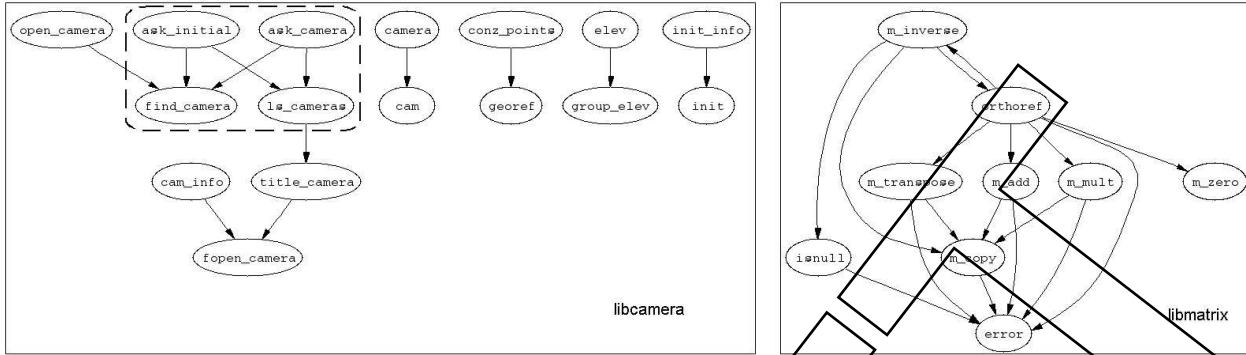
The analysis of clones inside libraries revealed an interesting situation: 16 symbols of library `libortho`, were cloned across `libimage_sup`, `libgmsh` and `libtrans`. Nine of the cloned functions were devoted to perform matrix algebra and, analyzing the *Dependency Graph* `libortho` (see Figure 5), a subgraph composed by such functions was identified (i.e., 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. The decision taken was to split `libortho` in two libraries, corresponding to the two boxes in Figure 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`.



**Figure 5. Splitting library libortho.**

Several “interesting” clones were also found outside libraries. In particular, the `r_mapcalc3` application contains four clusters of cloned, large functions (spanning from 27 to 59 LOCs). The first group contained functions called `f_add`, `f_mul`, `f_sub`, the second group `f_cos`, `f_sin` and `f_tan`, the third `f_and` and `f_or`, the fourth `f_double`, `f_int`, `f_float`. In all case, refactoring is clearly possible generalizing the operations and abstracting the types. We also found a function `weisemberg_bingham` cloned from `shapiro_francia` because (as from developer’s comment) the former statistics was not yet available in the system and it was temporarily replaced by the latter.

Finally we analyzed clones between applications and libraries. On most cases clones revealed to be part of *legacy* applications developed before the function was added into a library, and successively the application was never changed. A relevant (about 20%) fraction of these clones were discovered into the *contrib* sub systems, often developed by third-parties and therefore not always properly aligned with respect to the rest of the system.

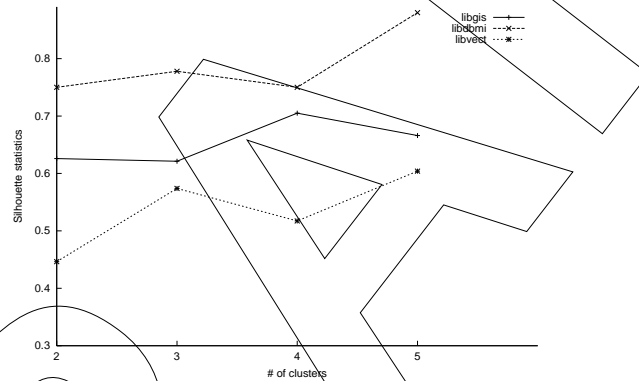
## 6.4 Library Refactoring

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

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

**Table 3. GRASS largest libraries.**

First and foremost, Silhouette statistics was used to determine the optimal number of clusters for each library. Values of the statistics, for different number of clusters, are plotted in Figure 6. Given this, we decided to split `libgis` into four clusters (instead of the six proposed in [3], `libvect-new` and `libdbmi` into three clusters. It is worth noting that, while for `libgis` the number of clusters was chosen in correspondence of the Silhouette maximum, for the other two libraries a compromise was pursued between maximizing the Silhouette and avoiding excessive fragmentation.



**Figure 6. Silhouette statistics for different number of clusters.**

Subsequently, a preliminary clustering was performed and, then, results were refined with a first execution of GA, without considering any developer's feedback (i.e., 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 the *PR* before applying the GA; and

- The number of inter-library dependencies and the *PR* after applying the GA.

Library	# 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 4. Results of the preliminary refactoring process.**

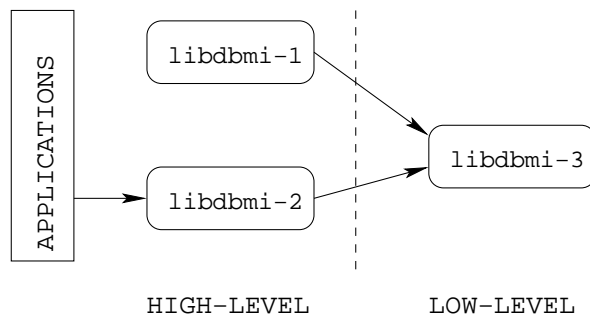
As shown, GA reduced `libgis` dependencies from 579 to 26 keeping the *PR* almost constant (from 51% to 48%). A significant reduction of inter-library dependency was obtained (from 237 to 4 for `libdbmi` and from 66 to 3 for `libvect`), also slightly reducing the *PR* (except that for `libdbmi`, where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

The first refactored architecture of the *candidate libraries* was shown to developers to collect their feedback. For `libgis` the 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 developer indicated that the first cluster should contain basic filesystem operations, and the other two all other functions (a specific distinction was not supplied in this case).

The feedback for `libdbmi` was pretty different with respect to the other two libraries. In this case developer (and also a manual graph analysis we did using *Dotty*) revealed that the solution suggested by the hierarchical clustering reflected programmer’s way to conceive the library. In fact, as also reported in [3], the library was split into three clusters:

- A cluster (`libdbmi-1`) containing (19) unused objects;
- A cluster (`libdbmi-2`) containing (30) objects directly used by applications; and
- A “low-level” library (`libdbmi-3`), containing 48 objects, used only internally to `libdbmi`.

Figure 7 reports the layering structure of 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 like *GRASS*, a compromise should be pursued between having smaller, decoupled clusters (like those generated applying GA) and clusters that, even if



**Figure 7. New libdbmi layering structure.**

not totally decoupled, are conceptually cohesed (i.e., they contain functions implementing closely-related tasks). In the latter case, memory optimization is even possible adopting, as said, dynamically loadable libraries.

Given this, we decided to leave libdbmi clusters as they were after hierarchical clustering, and to perform a “second round” of GA refactoring on libgis and libvect-new, considering, this time, also the *Feedback Factor FF*. For sake of completeness we also reported results for libdbmi. Varying the  $w_1$ ,  $w_2$  and  $w_3$  threshold we obtained different results. As shown in Table 5, it was never possible to pursue cluster complete decoupling and obtaining, at the same time, libraries very close to the structure proposed by developers (i.e., a small *FF*).

Library	# 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 5. Results of the second round of the GA refactoring process ( $w_3 \neq 0$ ).**

The comparison of the first three columns with the last three highlights that, after the first GA round, the coupling between clusters was kept low. On the other hand, the libraries produced tend to have a meaning different to what intended by developers (this was highlighted by the high *FF* before second round). The second round of GA tried to reduce the *FF*: this however increased coupling. At this stage, in the opinion of the authors, the developers may decide to produce *meaningful* libraries and reducing the memory requirements using dynamic-loadable libraries, or to obtain independent cluster, even if these clusters did not always group conceptually related objects.

The adoption of a hybrid GA approach, as mentioned before, did not allow us to improve accuracy, in that, increasing the number of generations and the population size, pure GA also converged to similar results. 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 to reduce, on average, the execution time of 43%. Convergence times for a *Compaq Proliant<sup>TM</sup>* (Dual Xeon<sup>TM</sup> 900 MHz processor, 2MB Cache and 4GB of RAM) are reported in Table 6.

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

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

## 6.5 Extraction of new Libraries

The final step of our framework is devoted to analyze the *Use Graph* obtained removing all existing libraries, to investigate the existence of new candidate libraries: sometimes there are groups of objects used by a common set of application, but they have not yet been clustered into libraries. To perform this task, clustering was performed on objects used by, at least, two applications.

Results revealed the presence of four clusters, all located in the *orthophoto* subsystem. The number of dependencies between clusters was small, and it was possible to solve them simply moving between clusters a couple of objects. Besides, all clusters had a considerable number of dependencies to external objects (i.e., other objects belonging to their same set of application). To eliminate these dependencies, it would be necessary to increase the size of each cluster of over 100%. This is clearly contradictory with respect to the miniaturization principles our miniaturization framework relies to. Consequently, it was decided not to cluster these objects into libraries. In the opinion of the authors, this is not a negative results; on contrary, this constitutes a quality indicator of the system: developer carefully created and maintained libraries.

## 7. Conclusions

This paper presented a framework for software miniaturization, as well as results from its application to an over 1 Million LOCs software system like the *GRASS* Geographical Information System. At the time of writing the application was successfully ported on a PDA (i.e., a CompaQ iPAQ). Given the size of the application and the available resources, a brute force automatic approach was not feasible: the developer's suggestions revealed to be an essential component for the miniaturization process.

The framework allowed to remove several, physiological problems from the analyzed software. In particular, unused objects were identified and factored out; clones were identified and, especially for clone inside libraries, refactoring was performed. Cloning level outside libraries was non negligible, suggesting further activity of clone refactoring. Besides, the cloning level inside libraries was low, except some situations like those highlighted, and the cloning between libraries and rest of the system was on most cases due to third party applications.

We defined a multi-steps library refactoring process, where hierarchical clustering identified a first, sub-optimal solution then refined by GA. The proposed fitness function kept into account different factors: minimizing dependencies, the average number of objects linked by each application and, finally, the feedback given by the developers. To produce results, compromise had to be pursued between the different factors.

The framework allowed to effectively refactor *GRASS*, reducing its memory requirements and improving its performances. In our case study, the average number of library objects linked by each application was reduced of about 50%.

Finally, the proposed framework can also be used as a quality metric profile: unused objects, clones, library coupling, library sizes and bad object refactoring are in fact significant quality indicators. For instance, the identification of no new libraries in *GRASS* indicated a good refactoring performed by developers. Work-in progress is devoted to incorporate dynamic information, exploited from instrumenting source code, in the library refactoring process.

## 8. 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.

## References

- [1] L. Garber, "Will 3G really be the next big wireless technology.," *IEEE Computer*, vol. 35, pp. 26–32, January 2002.
- [2] M. Garey and D. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.
- [3] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo, "Knowledge-based library re-factoring for an open source project," in *Proceedings of IEEE Working Conference on Reverse Engineering*, (Richmond - VA), Oct 2002 (to appear).
- [4] D. Doval, S. Mancoridis, and B. Mitchell, "Automatic clustering of software systems using a genetic algorithm," in *Software Technology and Engineering Practice (STEP)*, (Pittsburgh, PA), pp. 73–91, 1999.
- [5] E. Talbi and P. Bessière, "A parallel genetic algorithm for the graph partitioning problem," in *ACM International Conference on Supercomputing*, (Cologne, Germany), 1991.
- [6] M. Neteler and H. Mitasova, *Open Source GIS: A GRASS GIS Approach*. Boston/U.S.A; Dordrecht/Holland; London/U.K.: Kluwer Academic Publishers, 2002.
- [7] G. Snelting, "Software reengineering based on concept lattices," in *Proceedings of IEEE International Conference on Software Maintenance*, pp. 3–10, March 2000.
- [8] M. Krone and G. Snelting, "On the inference of configuration structures from source code," in *Proc. of the 16th International Conference on Software Engineering*, (Sorrento Italy), pp. 49–57, May 1994.
- [9] T. Kuipers and L. Moonen, "Types and concept analysis for legacy systems," in *Proceedings of the IEEE International Workshop on Program Comprehension*, pp. 221–230, June 2000.
- [10] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo, "A method to re-organize legacy systems via concept analysis," in *Proceedings of the IEEE International Workshop on Program Comprehension*, (Toronto, ON, Canada), pp. 281–290, IEEE Press, May 2001.
- [11] T. Kuipers and A. van Deursen, "Identifying objects using cluster and concept analysis," in *Proceedings of the International Conference on Software Engineering*, pp. 246–255, June 1999.

- [12] V. Tzerpos and R. C. Holt, "Software botryology: Automatic clustering of software systems," in *DEXA Workshop*, pp. 811–818, 1998.
- [13] V. Tzerpos and R. C. Holt, "MoJo: A distance metric for software clusterings," pp. 187–195.
- [14] V. Tzerpos and R. C. Holt, "ACDC: An algorithm for comprehension-driven clustering," in *Working Conference on Reverse Engineering*, pp. 258–267, 2000.
- [15] V. Tzerpos and R. Holt, "the stability of software clustering algorithms," 2000.
- [16] T. A. Wiggerts, "Using clustering algorithms in legacy systems modularization," in *Proceedings of IEEE Working Conference on Reverse Engineering*, 1997.
- [17] N. Anquetil and T. Lethbridge, "Extracting concepts from file names; a new file clustering criterion," in *Proceedings of the International Conference on Software Engineering*, pp. 84–93, April 1998.
- [18] E. Merlo, I. McAdam, and R. De Mori, "Source code informal information analysis using connectionist model," in *Proceedings of the International Joint Conference on Artificial Intelligence*, (Los Altos Calif), pp. 1339–44, 1993.
- [19] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *IEEE Proceedings of the 1998 Int. Workshop on Program Comprehension (IWPC'98)*, 1998.
- [20] S. Shazely, H. Baraka, and A. Abdel-Wahab, "Solving graph partitioning problem using genetic algorithms," in *Midwest Symposium on Circuits and Systems*, pp. 302–305, 1998.
- [21] T. N. Bui and B. R. Moon, "Genetic algorithm and graph partitioning," *IEEE Transactions on Computers*, vol. 45, pp. 841–855, Jul 1996.
- [22] B. Gommen and E. de St. Croix, "Graph partitioning using learning automata," *IEEE Transactions on Computers*, vol. 45, pp. 195–208, Feb 1996.
- [23] H. Maini, K. Mehrotra, C. Mohan, and S. Ranka, "Knowledge-based nonuniform crossover," in *IEEE World Congress on Computational Intelligence*, pp. 22–27, 1994.

- [24] M. Harman, R. Hierons, and M. Proctor, "A new representation and crossover operator for search-based optimization of software modularization," in *AAAI Genetic and Evolutionary Computation Conference (GECCO)*, (New York, USA), pp. 82–87, July 2002.
- [25] G. Antoniol, M. Di Penta, and M. Neteler, "Moving to smaller libraries via clustering and genetic algorithms," (Benvento (Italy)), Mar 2003 (to appear).
- [26] M. R. Anderberg, *Cluster Analysis for Applications*. Academic Press Inc., 1973.
- [27] A. Jain and R. Dubes, *Algorithms for Clustering Data*. Prentice-Hall, 1988.
- [28] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.
- [29] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub Co, Jan 1989.
- [30] L. Kaufman and P. Rousseeew, *Finding groups in data: an introduction to cluster analysis*. Wiley - NY: Wiley-Inter Science, 1990.
- [31] A. Gordon, *Classification - (2nd edition)*. London: Chapman and Hall, 1988.
- [32] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo, "Modeling clones evolution through time series," *Proceedings of IEEE International Conference on Software Maintenance*, pp. 273–280, Nov 6-10 2001.
- [33] N. Anquetil, "A comparison of graphs of concept for reverse engineering," in *Proceedings of the IEEE International Workshop on Program Comprehension*, pp. 231–240, June 2000.
- [34] M. Siff and T. Reps, "Identifying modules via concept analysis," *IEEE Transactions on Software Engineering*, vol. 25, pp. 749–768, Nov-Dec 1999.
- [35] P. Tonella, "Concept analysis for module restructuring," *IEEE Transactions on Software Engineering*, vol. 27, pp. 351–363, April 2001.
- [36] K. Deb, "Multi-objective genetic algorithms: Problem difficulties and construction of test problems," *Evolutionary Computation*, vol. 7, no. 3, pp. 205–230, 1999.

- [37] J. Horn, N. Nafpliotis, and D. E. Goldberg, "A Niche Pareto Genetic Algorithm for Multiobjective Optimization," in *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, vol. 1, (Piscataway, New Jersey), pp. 82–87, IEEE Service Center, 1994.
- [38] E. Zitzler, K. Deb, and L. Thiele, "Comparison of Multiobjective Evolutionary Algorithms on Test Functions of Different Difficulty," in *Proceedings of the 1999 Genetic and Evolutionary Computation Conference. Workshop Program* (A. S. Wu, ed.), (Orlando, Florida), pp. 121–122, 1999.
- [39] AT&T, "Dotty directed graph editor (part of graphviz)."  
<http://www.research.att.com/sw/tools/graphviz/>.
- [40] E. Koutsofios, "Editing graphs with *dotty*," technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, July 1994.
- [41] "The R project for statistical computing."  
<http://www.r-project.org>.
- [42] R. Ihaka and R. Gentleman, "R: A language for data analysis and graphics," *Journal of Computational and Graphical Statistics*, vol. 5, no. 3, pp. 299–314, 1996.
- [43] M. Wall, "GAlib - a C++ library of genetic algorithm components."  
<http://lancet.mit.edu/ga/>.
- [44] M. Neteler, ed., *GRASS 5.0 Programmer's Manual. Geographic Resources Analysis Support System*.  
<http://grass.itc.it/grassdevel/html>: ITC-irst, Italy, 2001.
- [45] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta, "Analyzing cloning evolution in the linux kernel," *Information and Software Technology (SCAM 2002 Special Issue)*, vol. 44, pp. 755–765, Oct 2002.