Library Miniaturization Using Static and Dynamic Information

Giuliano Antoniol and Massimiliano Di Penta
antoniol@ieee.org, dipenta@unisannio.it

RCOST - Research Centre on Software Technology
University of Sannio, Department of Engineering
Palazzo ex Poste, Via Traiano, I-82100 Benevento, Italy

Abstract

Moving to smaller libraries can be considered as a relevant task when porting software systems to limited-resource devices (e.g., hand-helds). Library miniaturization will be particularly effective if based on both dynamic (keeping into account dependencies exploited during application execution in a given user profile) and static (keeping into account all possible dependencies) information.

This paper presents a distributed software architecture, based on web services, to collect dynamic information at run-time, and an approach for miniaturization of libraries, exploiting both dynamic and static information with the aim of reducing the memory requirements of executables.

New, smaller libraries are identified via hierarchical clustering and genetic algorithms; clustering produces a first initial solution, then optimized by multi-objective genetic algorithms.

The approach has been applied to medium size open source software systems such as Samba and MySQL, allowing to effectively produce smaller, loosely coupled libraries, and to reduce the memory requirements of each application.

Keywords: library miniaturization, dynamic dependencies, clustering, genetic algorithms, trace extraction

1. Introduction

Software systems usually contain large libraries composed by functions used by several applications. However, often applications tend to link a library while using only a small portion of it. This constitutes a serious problem, especially when porting applications on limited-resource devices. Consider, for example, embedded systems; the amount of resources available is often limited, and thus developers are interested to reduce the footprint of executables. Applications running on hand-held devices have similar, even if less stringent, resource requirements. To minimize the overhead derived from linking unused objects, libraries should be split into smaller ones, so that each application links only what strictly necessary.

Previous approaches, such as those proposed in [4, 5, 9], were based on static information, extracted by source code or object module analyzers. This, however, gave a limited view of the complex dependencies and relationships existing between different components of a software system. In fact, statically computed dependencies may not reflect the real user profile, leading to unnecessary large executables. Meanwhile, if, for example, a static calling dependency between two functions exists, then there may exist rare environment and parameter configurations forcing a function to call the other. On the other hand, dynamic information reflects the *in field* application use. Accounting for it when clustering objects into libraries will lead to a more realistic, user profile-oriented, result.

This paper proposes an architecture, based on web services, to collect program traces *in the field* and an approach, based on static and dynamic information, for library miniaturization. The web services architecture allowed us to collect traces even from machines outside a firewall.

We propose to linearly combine static and dynamic information (i.e., dependencies) with a weight, expressing our believe about the frequency of rare events. In other words, static information ensures to keep into account dependencies not dynamically exploited during the execution of the instrumented code. The heuristic to define the new software organization was inspired from what described in [5]: the central idea is to apply clustering techniques to identify software libraries minimizing the average executable size, followed by Genetic Algorithms (GA) to improve the identified solution. A multi-objective fitness function was defined, to keep low, at the same time, both the inter-library dependencies and the average number of objects linked (or dynamically loaded, in case of dynamically-loadable libraries) by each application.

The approach was applied to improve the library orga-

nization of public domain software applications such as *Samba* and *MySQL*.

The paper is organized as follows. First, the essential background notions to help the reader are summarized in Section 2; the architecture of the trace collector is shown in Section 3. Section 4 describes the miniaturization approach and support tools. Information on the case study systems is reported in Section 5. Section 6 presents case study results. Finally, an analysis of related work is reported in Section 7, before conclusions and work-in-progress.

2. Background Notions

To miniaturize software system libraries, clustering and GA were applied to information obtained from both static and dynamic analysis of software system dependencies.

Clustering deals with the grouping of large amounts of things (*entities*) in groups (*clusters*) of closely related entities. In this paper, the agglomerative-nesting (*agnes*) algorithm [23] was applied to build the initial set of *candidate libraries*. As in [5], the optimal number of clusters has been chosen applying the *Silhouette* statistics, proposed by Kaufman and Russeeeuw in [23]. More details on clustering can be found in [2, 22].

GA come from an idea, born over 30 years ago, of applying the biological principle of evolution to artificial systems. 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 operators: 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., by flipping some of its bits if the genome is represented by a bit string). Crossover and mutation are performed on each individual of the population with probability *pcross* and *pmut* respectively, where $pmut \ll pcross$. Further details on GA can be found in [18].

3. Trace Extractor Architecture

To obtain dynamic dependency information from a software system, it is necessary to instrument its code so that, during application execution, the following information is gathered for each function call:

1. The name of the application using the function;
2. The caller function and the name of the object module containing it; and
3. The called function and the name of the object module containing it.

Having both function and object names available, it is possible to perform clustering/miniaturization at different levels of granularity. In this paper we adopted a coarse-grain approach, similarly to what done in [4, 5, 9], aiming at factoring library objects into more smaller new libraries.

Once an instrumented version of the software system to analyze is available, gathering traces presents two kinds of problems: the trace compression and the trace collection from a distributed execution environment.

The first problem is due to the enormous number of functions called, even for performing a simple task. To avoid trace database rapidly becoming unmanageable, a pipe-based compressor (its architecture is shown in Figure 1-a) has been implemented. The instrumented program writes traces to a FIFO queue via thread safe-primitives (avoiding synchronization problems with other writing processes) rather than to a file. Then, traces are read from a compressor that performs two levels of compression:

1. Run-Length-Encoding (RLE) compression, replacing sequences of the same trace (generated, for example, by iterative or recursive calls) with a weighted call; and
2. Gzip compression.

Multiple applications run at the same time on several machines of different customers (i.e., the subjects we collected traces from), and network connectivity may not be always present or the communication bandwidth may be devoted to higher priority processes. This leads to a distributed architecture, (shown in 1-b) where traces are firstly stored in a temporary local database, then transferred to a centralized database.

Being customer's machines and trace analysis machines geographically distributed in different sub-networks, firewalls could deny communication through sockets or through middlewares like CORBA. In this case the best solution is represented by web services. Communication is based on SOAP (Simple Object Access Protocol) protocol over HTTP (that traditionally operates on the TCP port 80, usually not filtered by firewalls).

Periodically, a *web service client* sends locally collected traces to the remote *trace collector web service*, that stores them into a centralized database. A locally unique id (including a time stamp) is assigned by the collector to each

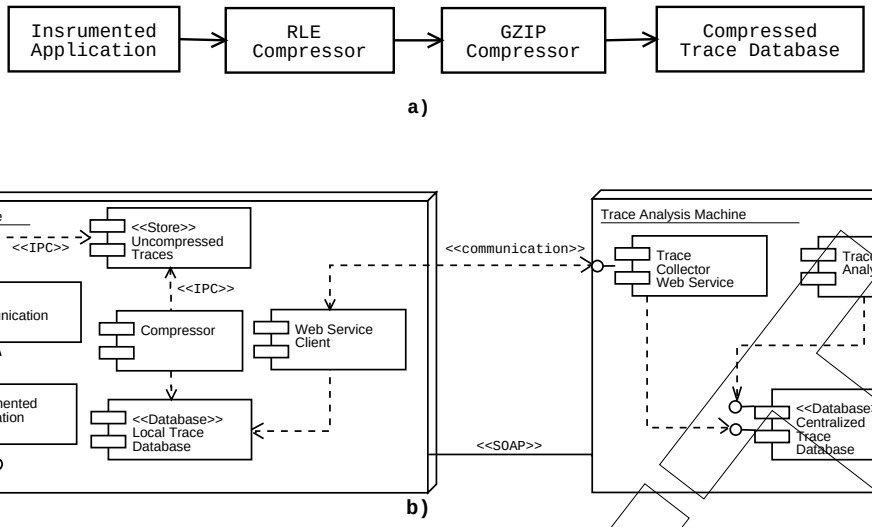

Figure 1. a) Pipe architecture of the trace compressor. b) Deployment diagram of the trace collector system.

client, ensuring that traces of different running instances were not mixed.

To effectively split software system libraries, it is essential that traces were obtained executing system applications in an environment as close as possible to the target profile of use. Moreover, we suggest to exercise applications in all (most of) their functionalities, considering all categories of inputs and also special situations. In other words, a functional testing based on category partitioning [25] should be performed.

For this purpose, results presented in this paper were computed from dynamic information obtained combining both traces obtained performing a category partitioning test, and traces obtained letting users interact with the system for some (three-four) days.

4. The Miniaturization Approach

This section describes the proposed miniaturization approach, derived from what already proposed in [5]. In particular the GA proposed in [5] has been modified to keep into account information obtained from program execution. As shown in Figure 2, the software system undergoes two kinds of analysis:

1. Static analysis, performed using a toolkit based on the `nm` Unix tool (detailed in Section 4.4 and in [4]);

2. Dynamic analysis, using the instrumentation architecture described in Section 3.

Then, static and dynamic information are combined and, finally, the same steps described in [5] are performed.

4.1. Basic Criteria and Representation

As described in [4, 5, 9], given a system composed of m applications and n libraries, the idea is to split the biggest libraries in two or more smaller clusters, such that each cluster contains symbols used by a common subset of applications and the coupling between different clusters is low.

Given $A = \{a_1, a_2, \ldots, a_m\}$ the set of software system applications (as detailed in [4] and in Section 4.4, identified matching the `main` symbol), and given a library l_j to be split, containing a set $O \equiv \{o_1, o_2, \ldots, o_{p_j}\}$ of p_j objects, static analysis performed in our previous works relied on two matrices:

1. A matrix *SMU* of *uses* of library objects by applications, where each item $smu_{i,j}$ is equal to 1 if application i links object j, 0 otherwise; and

2. A matrix *SMD* of *dependencies* between library objects, where each item $smd_{i,j}$ is equal to 1 if object i depends from object j, 0 otherwise.

The new model accounts for dynamic information, therefore we need to introduce two new matrices:

1. *DMU*, where each item $dmu_{i,j}$ is the frequency of *uses* of object j by application i. The frequency is computed dividing the number of times application i uses object j by the total number of uses of library objects by all applications; and

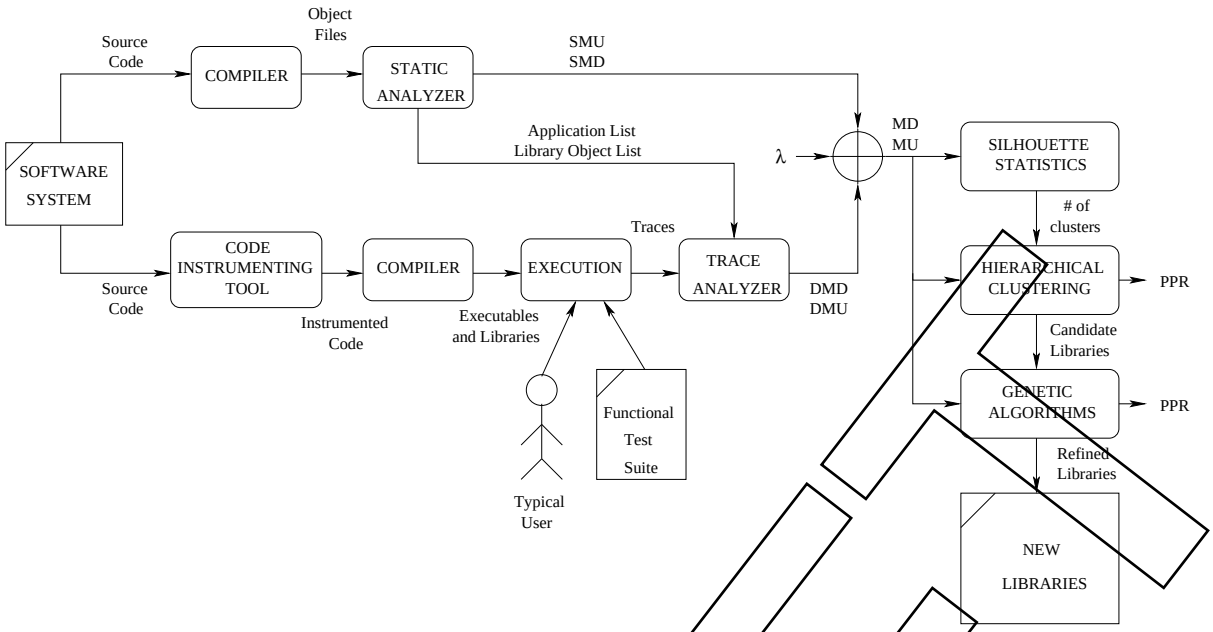

Figure 2. The miniaturization process.

2. *DMD*, where each item $dmd_{i,j}$ is the frequency of *accesses* of library object i to library object j. The frequency is computed like in matrix *DMU*.

To have at least a broad idea of the effectiveness of the trace extraction process (and therefore of our test suite) we define two Coverage Ratios *CR(DMU)* and *CR(DMD)*. In particular, *CR(DMU)* is defined as:

$$
\begin{array}{l}
total = 0 \\
covered = 0 \\
\forall\ application\ a_x \\
\quad \forall\ object\ o_y \\
\quad\quad if\ mu_{x,y} <> 0\ then \\
\quad\quad\quad total = total + 1 \\
\quad\quad\quad if\ dmu_{x,y} <> 0\ then \\
\quad\quad\quad\quad covered = covered + 1 \\
\quad\quad\quad end\ if \\
\quad\quad end\ if \\
CR(DMU) = covered * 100/total
\end{array}
$$

CR(DMD) is computed similarly.

To combine static and dynamic information *SMU* and *SMD* not-null values are replaced by the lowest frequency contained respectively in the *DMU* and *DMD* matrices, assigning to static dependencies the meaning of "rare events".

Given this, we can define matrices *MU* and *MD*, accounting for overall contribution, where:

$$mu_{i,j}(\lambda) = \lambda\ smu_{i,j} + (1 - \lambda)\ dmu_{i,j} \quad (1)$$

$$md_{i,j}(\lambda) = \lambda\ smd_{i,j} + (1 - \lambda)\ dmd_{i,j} \quad (2)$$

Varying λ we can change the influence of static and dynamic information: in particular, for $\lambda = 1$ we obtain exactly the same static model proposed in [5], while for $\lambda = 0$ the model is merely dynamic.

4.2. Determining the Sub-Optima Libraries by Clustering

For each *"old library"*, agglomerative-nesting clustering is performed to determine:

1. The optimal number of clusters k, applying the Silhouette statistics. As explained in [5], the optimum can be found sometimes in correspondence of the curve maximum, sometimes at the *elbow* of that curve, also keeping into account a tradeoff between excessive fragmentation and library size; and

2. An initial solution that constitutes the *starting population* of GA.

A measure of the performances of the miniaturization process, the *Partitioning Ratio* (PR) has been introduced in [9] and also adopted as a quality measure of the GA-based clustering proposed in [5]. Broadly speaking, the PR represents a ratio between the average number of objects linked by applications after and before splitting the libraries. 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. As it will be explained below, the same equation (although having a different meaning) can be used for the new approach.

Let k the number of clusters $l_{x_1}, \ldots, l_{x_k}$ obtained from a library l_x. Then, the *Probabilistic Partitioning Ratio* PPR_x can be defined as:

$$PPR_x(\lambda) = 100 * \sum_{i=1}^{m} \frac{\sum_{j=1}^{k} |l_{x_j}| * mu_{i,x_j}(\lambda)}{|l_x| * mu_{i,x}(\lambda)} \qquad (3)$$

where $|l_x|$ is the number of objects archived into library l_x.

The PPR has a slightly different interpretation with respect to the PR: the new algorithm tries to cluster together objects having *high probability* to be used by the same application(s), to avoid that an application linked objects unlikely to be used (dynamically loading them at run-time only when needed). Clearly, being MU function of λ, we may obtain different PPR indices having different static and dynamic contributions.

To measure the improvement obtained, the PPR is computed (as also shown in Figure 2) both after the preliminary (hierarchical) clustering and after the refined (GA-based) clustering.

4.3. Reducing Dependencies using Genetic Algorithms

Hierarchical clustering gives us a first approximation of the new, smaller libraries, and allows, using the Silhouette statistics, to compute the optimal number of clusters. Hierarchical clustering, however, does not allow us to balance different factors (such as coupling and cohesion) by moving objects between libraries. This could lead to new, coupled libraries, forcing to load (or link) a library each time a symbol from that library is needed, therefore wasting the advantage of having new smaller libraries.

Although adopting dynamic-loadable libraries can help saving memory (each library is loaded at run-time only when needed, and then unloaded when it is no longer used), the initial clusters should be refined to minimize static and dynamic dependencies. This is a NP-hard problem [28], and an approximate solution can be obtained using a GA. We are also experiencing that hybrid approaches based on both GA and hill climbing, according to what also stated in [19], contribute to obtain more accurate results with improved algorithm execution performances. These optimizations, however, are out of scope of this paper.

As explained in [5], our genome is encoded as a bit-matrix, where $g_{i,j} = 1$ if object i belongs to cluster j. The encoding schema widely adopted in literature for remodularization [10] and, in general, for graph partitioning [28] 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 N-size array G, where the integer p in position q means that the object q is contained into partition p. With respect to such a genome, the bit-matrix allows an object belonging to more than one library (indicated by more "1" on the same column). Such "cloning" sometimes represents the only way to reduce inter-library dependencies.

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 multi-objective fitness function tries to:

1. Minimize the probability of inter-library dependencies;
2. Minimize the number of objects linked by applications but not frequently used; and
3. Keep constant the size of each library (with respect to the sub-optima computed by hierarchical clustering), avoiding to let GA grouping a large fraction of the objects in the same library, thus negatively affecting the PR.

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

$$DF(\lambda, g) = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} md_{i,j}(\lambda)\ \delta_{i,j} \qquad (4)$$

where

$$\delta_{i,j} = \begin{cases} 0 & i = j \\ 1 & i \neq j \end{cases}$$

Similarly to the PPR, DF is function of λ: for $\lambda = 1$ it represents, as discussed in [5], a measure of static, inter-library dependencies.

The second factor corresponds to the PPR described in Section 4.2. 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. A similar factor was also applied in [28]. Given S_0 the array of library sizes for the initial population, and S_g the same for the g-th generation:

$$SF(g) = |\sigma_{S_0} - \sigma_{S_g}| \qquad (5)$$

Finally, the fitness function F to be minimized by GA is defined as:

$$F(\lambda, g) = DF(\lambda, g) + w_1\ PPR(\lambda, g) + w_2\ SF(g) \quad (6)$$

where w_1 and w_2 are real, positive weighting factors. The higher is w_1, the better will be the PPR ratio; on the other

hand, increasing too much w_1 decreases the dependency reduction. Similarly, the higher is w_2, the more similar will be the result to the starting set of libraries, while an excessively higher w_2 could not allow a satisfactory dependency reduction.

The choice of the fitness function has been inspired by the principle of choosing the simplest model (e.g., Occam's razor, minimum description length, small number of free parameters). Hence, equation (6) merely represents a linear combination of the three factors to be minimized. Although different, perhaps more sophisticated functions (e.g., non-linear functions) could lead to better results, a linear combination can reduce the effort needed for calibrating the parameters of such a multi-objective functions [8, 20, 32].

The challenge of dealing with multi-objective functions is to balance conflicting factors. For example, moving an object from a library to another can reduce the PPR while increasing the DF. The most rigorous way to handle multi-objective functions is based on the concept of *dominance* between vectors (i.e., points in a multi-dimension space). We can say that x is *dominated* by y or x is *partially less* than y ($x <_p y$) when the following conditions hold:

$$(x <_p y) \Leftrightarrow (\forall_i)(x_i \leq y_i) \wedge (\exists i)(x_i < y_i) \qquad (7)$$

When working with a multi-objective function, a selection of dominated individuals is chosen as the set of the best individuals for each generation. The influence of the different fitness function factors can be calibrated by adjusting the w_i parameters. Notice that we set a unitary weight to the DF, in that we aimed to maximize dependency reduction. Then we selected w_1 and w_2 using a trial-and-error, iterative procedure, adjusting them each time until the DF, PPR and SF obtained at the final step were not satisfactory. The process was guided by computing each time the average values for DF, PPR and SF, and also by plotting their evolution, in order to determine the 3D space region in which the population should evolve.

In this paper we adopted the same genetic operators proposed in [5]. In particular, the crossover operator is the *one point crossover*: given two matrices, both are cut at the same random column, and the two portions are exchanged. 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; and
2. With probability $pclone < pmut$, it takes a random position in the matrix: if it is zero and the library is dependent from it, then the mutation operator clones the object into the current library.

It is worth noting that the semantics of the *one point crossover* and of the (non-cloning) mutator are exactly the same of those proposed in [10], thus the particular choice of the (bit-matrix) genome, motivated by the possibility of "cloning" does not affect results.

Of course the cloning of an object increases both PPR and SF, therefore it should 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 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 on which l_i and l_j depend from.

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

4.4. Tool Support

The entire miniaturization process required several tools, some of which already described in [5] and in [9]. In particular:

- *The application identifier* that, using the `nm` Unix tool, identifies the list of object modules containing the `main` symbol;
- *The static dependency graph extractor*, also based on the `nm tool`, that produces the SMD and the SMU matrices;
- *The number of clusters identifier:* as said in Section 2, the number of clusters was determined using the Silhouette statistics. In particular, implementations available in the *cluster* package of the *R Statistical Environment* [1, 21] were used;
- *The library miniaturization tool*: it supports the process of splitting libraries in smaller clusters. As said in Section 4.2, this is performed by clustering algorithms. Again, the cluster analysis is performed by the *agnes* function available under the *cluster* package of the *R Statistical Environment*; and
- *The GA library refiner:* implemented in C++ using the *GAlib* [30].

Moreover, to perform trace generation, collection and analysis, the following tools have been implemented:

- *The code instrumenting tool*, that wraps a C/C++ compiler (i.e., *gcc/g++* in our case), by means of a recursive descendant hand-coded parser, and inserts probes into the (pre-processed) source code;
- *The communication library*, linked to the instrumented application to support local communication with the uncompressed trace queue (see Figure 1-b);
- *The trace compressor*, that reads from the FIFO queue traces written by the instrumented application, ad compresses them using RLE and Gzip;
- *The web service client*, that periodically flushes the local trace database, sending traces to the *trace collector web service*;
- *The trace collector web service*, that collects traces from client machines and stores them into the centralized trace database; and
- *The trace analyzer*, that extracts DMD and DMU matrices from the collected traces.

5. Case Studies

As explained in Section 6, for each system we chose to refactor the largest libraries. Characteristics of the analyzed systems are shown in Table 1. This section will briefly describe these software systems, giving also a summary of the heuristics followed to collect dynamic information.

System	Ver	KLOC	Apps	Libs	Libs to re-organize
MySQL	3.23.51	478	38	24	2
Samba	2.2.7	295	16	2	2

Table 1. Case study characteristics.

5.1. MySQL

MySQL (`http://www.mysql.com/`) is an open source, fast, multi-threaded, multi-user SQL database server, intended for mission-critical, heavy loaded production systems. *MySQL* is written using both C and C++ and can be compiled with several different C/C++ compilers.

The power of *MySQL* is in its fastness: in order to pursue this objective, some advanced features (e.g., nested queries) are not available, while others (e.g., transactions) were introduced only in the latest version of the database server.

To obtain traces from *MySQL*, all its utilities were exercised as explained in Section 3. In particular: administrating databases, e.g. creating, flushing, removing databases; checking *MySQL* configuration; checking and compressing indexed sequential (ISAM) files; accessing to *MySQL* log files; importing/exporting tables from/to ASCII files; dumping a database; and, above all, performing several SQL operations from the interactive console.

5.2. Samba

Samba (`http://www.samba.org`) is a freely available file server that runs on Unix and other operating systems (usually to share resources between Unix-based systems and Microsoft-based systems). The code has been written to be as portable as possible. It has been "ported" to many unixes (Linux, SunOS, Solaris, SVR4, Ultrix, etc.).

Samba consists of two key programs, `smbd` and `nmbd`, plus a bunch of other utilities. They implement four basic services: file sharing & print services, authentication and authorization, name resolution and service announcement (browsing). Moreover, *Samba* comes with a variety of utilities. The most commonly used utilities are: `smbclient` a simple SMB (Server Message Block, a protocol for sharing general communications abstractions such as files, printers, etc.) client; `nmblookup` a NetBIOS name service client, and `swat` which is the *Samba Web Administration Tool*, it allows the configuration of Samba remotely, using a web browser.

Traces were extracted during functional testing and four days of normal use. All *Samba* functions were exercised in their options, for example: changing passwords and managing users with `smbpasswd`; accessing and modifying Samba configuration with `testparm` and `swat`; and performing several file-transfer operations (getting files, putting files, removing files, creating/renaming/removing directories, etc.) both with `smbclient` and GUI clients like *Microsoft Windows Explorer*^{TM} or *Gnomba*.

6. Case Study Results

This section reports results obtained applying the proposed miniaturization process to *Samba* and *MySQL*. Similarly to [5], for Samba we basically re-organized the two system libraries, trying to minimize dynamic dependencies between them, clustering together, at the same time, objects having a high frequency of use by a common set of applications. With the same criteria, we decomposed in small chunks the two largest *MySQL* libraries: *libmysqlclient* and *libmysys*.

Not surprisingly, dynamic execution exploited a limited set of uses and dependencies: for *Samba* we obtained a $CR(DMD)$ of 40% and a $CR(DMU)$ of 30%. Indices were even smaller for both *MySQL* libraries: $CR(DMD)$ of 25% and $CR(DMU)$ of 20%. This supported the approach, described in Section 4, of combining static and dynamic information. On the other hand, it should be consid-

System	Library	# of objs	# of Clusters (k)	Silhouette statistics	Before GA				After GA			
					DF (0.5)	DF (1)	PPR (0.5)	PPR (1)	DF (0.5)	DF (1)	PPR (0.5)	PPR (1)
Samba	`lib + libsmb`	77	2	0.71	95.26	180	6.64%	96%	18	33	6.00%	94%
MySQL	`libmsqlclient`	80	3	0.64	125	166	2.85%	93%	37	49	2.70%	93%
	`libmysys`	92	2	0.47	69	99	3.12%	99%	6.80	10	3.00%	97%

Table 2. Results for $\lambda = 0.5$.

ered that different user execution profiles will lead to different coverage indices.

Results were computed for three different values of λ: 0, 1 and 0.5, i.e., considering only dynamic information, only static information, and a combination of both.

The first step was to determine the optimal number of clusters: for $\lambda = 1$ and $\lambda = 0.5$, Silhouette statistics gave us the same optimal number of clusters, i.e. two for *Samba*, three for *libmysqlclient* and two for *libmysys*. Showing details on Silhouette statistics analysis is out of scope of this paper (details are reported in [5]).

Results obtained combining both static and dynamic information are shown in Table 2. DF and PPR before and after applying GA are reported. This time DF is, as explained, a measure of the dynamic coupling between libraries. Static DF (i.e., the number of static dependencies after applying GA, indicated with $DF(1)$), are also reported, as well as static PPR (indicated with $PPR(1)$). It is worth noting that PPR values are very smaller with respect to PR ones: in other words, the algorithm tries to cluster objects having a high frequency of common use.

Table 2 highlights different results. First of all, the GA significantly (for Wilcoxon tests performed with significance level $\alpha = 95\%$) reduced the dynamic dependencies, leaving almost unaltered the PPR indices. The latter is not a negative aspect: PPR obtained, as it will be discussed below, are already good after hierarchical clustering. On the other hand, GA tried to minimize the DF as much as possible, without significantly affecting the PPR. This allowed us to create *dynamically independent* libraries, i.e., with a low probability of coupling.

Although, in this case, the objective of the fitness function was to minimize dynamic dependencies and to cluster together objects having high frequency of common use, it is interesting to analyze what happened to static parameters ($PPR(1)$ and $DF(1)$). Clearly, as shown by values over 90% and, in some cases, close to 100%, the $PPR(1)$ was not minimized by hierarchical clustering nor by GA. Again, this is not surprising: common, but rare use of objects by the same set of applications were not weighted, in terms of importance, as the most frequent use relationships.

About $DF(1)$, Table 2 shows a significant reduction after applying GA. Resulting static dependencies (from 10 of *libmysys* to 37 of *libmysqlclient*) were not so small to allow (like in the static analysis performed in [5] and shown in Table 3) obtaining completely independent libraries, but were minimized to reduce the coupling even for rare events.

After discussing obtained figures, results should be interpreted analyzing the meaning of the libraries obtained. Analyzing the two new *Samba* libraries we observed, for example, that the first library contained objects frequently used by daemons (*smbd* and *nmbd*), while the second library contained objects often used by administration utilities, such as *smbpasswd*. Similarly, *MySQL libmysys* library was split in a library containing general-purpose functions, often used by the SQL engine, and another library containing utility functions used by different *MySQL* applications. Finally, *libmysqlclient* was split in a library containing general-purpose functions (memory allocation, stream handling, etc.), a library containing utility functions for client applications, and a library for interfacing client applications with the database server.

System	Library	Before GA		After GA	
		DF (1)	PPR (1)	DF (1)	PPR (1)
Samba	`lib + libsmb`	106	72%	2	64%
MySQL	`libmsqlclient`	187	76%	7	70%
	`libmysys`	158	89%	1	66%

Table 3. Results for $\lambda = 1$.

Results obtained from static information were basically the same presented in [5] (although we analyzed a different version of *Samba*) and are reported, for sake of completeness, in Table 3. It is worth noting that DF and PPR correspond to the *static* parameters reported in [5].

Finally, we considered the case of $\lambda = 0$, i.e., taking into account only dynamic links. Although $DF(\lambda = 0)$ factors were successfully minimized and PPRs were comparable (and even smaller) to those obtained for $\lambda = 0.5$, we decided to discard these results. This because the number of static dependencies after applying GA remained high, increasing the risk that a dependency not exploited by instru-

mentation will occur, forcing therefore an extra library to be loaded at run-time. In other words, results obtained for $\lambda = 0$ confirmed one of the most important lessons learned from this work: dynamic information allows a successful miniaturization, but it should be complemented with static information.

7. Related Work

The present work deals with basically three kinds of analysis tools: clustering techniques, GA and trace extraction and analysis.

A survey of clustering techniques applied to software engineering was presented by Tzerpos and Holt in [29]. In [3] a method for decomposing complex software systems into independent subsystems was proposed by Anquetil and Lethbridge. Our work shares with [24] the idea of analyzing intra-module and inter-module dependency graphs, finding a tradeoff between having highly cohesive libraries and a low inter-connectivity.

GA has been recently applied in different fields of computer science and software engineering. An approach for partitioning a graph using GA was discussed in [28]. GA were used by Doval et al. [10] for identifying clusters on software systems. We share with this paper the idea of a software clustering approach using GA, trying to minimize inter-cluster dependencies. Finally, Harman et al. [19] reported experiments of modularization/remodularization, comparing GA with hill climbing techniques, also introducing a representation and a crossover operator for that problem. Their case studies revealed that hill climbing outperformed GA.

Recently, several works have considered a new perspective taking into consideration both static and dynamic information extracted from software artifacts [6, 11, 31] to locate concepts, to define coupling, or to support maintenance and program comprehension activities. In [27] T. Systä presented a tool for extracting scenarios from traces obtained from debugging Java bytecode. Similarly, in [26] authors used dynamic information to recovery collaboration diagrams and roles. An automatic technique for extracting operation sequences was shown in [15], while [7] described an approach, based on trace analysis, to discover thread interactions. Eisenbarth et al. [13, 12, 14] applied CA on information obtained from both static and dynamic analysis to understand features present into source code. More generally, sinergies and dualities between static and dynamic analysis were illustrated in [16].

The idea of using dynamic information for clustering was proposed in [17], where authors proposed a tool, Gadget, for extracting the dynamic structure of Java programs. With respect to [17], we propose to combine static and dynamic information, in that often not all dependencies and uses are exploited when executing instrumented programs. Moreover, we propose an architecture to collect traces from a distributed execution environment.

Our library reorganization studies come from [4] where we proposed the idea of recovering libraries and creating a source file directory structure using CA. This paper shares with [4] the idea of finding libraries searching for sets of objects used by common groups of applications. In [9] the miniaturization of *GRASS* libraries was described and then refined in [5], proposing the static model then extended in the present paper to consider dynamic information.

8. Conclusions

The miniaturization approach proposed in this paper produced smallest, loosely coupled libraries from the original biggest ones. By incorporating dynamic information into clustering and GA, we optimized libraries with respect to a given user profile. However, different profiles would have lead to different libraries, making the proposed mechanism appealing for generating customized software configurations. Combining static and dynamic information allowed us to keep into account dependencies not exploited by software instrumentation. Finally, the web services architecture revealed itself to be essential for collecting traces in a geographically distributed environment.

Work in progress is devoted to perform a more detailed analysis on the new (miniaturized) libraries, also incorporating maintainer feedback into GA in order to improve the results. Application of the proposed method to other, biggest, systems, for which library miniaturization is particularly relevant, is also being performed. Finally, we will also investigate on the influence of the chosen fitness function on the results.

9. Acknowledgments

Authors would thank the anonymous reviewers for their precious suggestions and feedback.

References

[1] The R project for statistical computing. `http://www.r-project.org`.

[2] M. R. Anderberg. *Cluster Analysis for Applications*. Academic Press Inc., 1973.

[3] N. Anquetil and T. Lethbridge. Extracting concepts from fi le names; a new fi le clustering criterion. In *Proceedings of the International Conference on Software Engineering*, pages 84–93, April 1998.

[4] 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, pages 281–290, Toronto, ON, Canada, May 2001. IEEE Press.

[5] G. Antoniol, M. Di Penta, and M. Neteler. Moving to smaller libraries via clustering and genetic algorithms. In *European Conference on Software Maintenance and Reengineering*, pages 307–316, Benevento (Italy), Mar 2003.

[6] E. Arisholm. Dynamic coupling measures for object-oriented software. In *Proceedings of the Eighth IEEE Symposium on Software Metrics*, pages 33–42, Ottawa, Canada, June 4-7 2002.

[7] J. Cook and Z. Du. Discovering thread interactions in a concurrent system. In *Proceedings of IEEE Working Conference on Reverse Engineering*, pages 255–264, Richmond, VA, USA, October 2002.

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

[9] 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*, pages 319–328, Richmond - VA, Oct 2002.

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

[11] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 602–611, Florence, Italy, November 6-10 2001.

[12] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 602–611, Florence, Italy, Nov 2001. IEEE Computer Society Press.

[13] T. Eisenbarth, R. Koschke, and D. Simon. Feature-driven program understanding using concept analysis of execution traces. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pages 300–309, Toronto, ON, Canada, May 2001. IEEE Computer Society Press.

[14] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, Mar 2003.

[15] T. Eisenbarth, R. Koschke, and G. Vogel. Static trace extraction. In *Proceedings of IEEE Working Conference on Reverse Engineering*, pages 128–137, Richmond, VA, USA, October 2002.

[16] M. D. Ernst. Static and dynamic analysis: synergy and duality. In *ICSE Workshop on Dynamic Analysis (WODA)*, Portland, Oregon, USA, May 2003.

[17] J. Gargiulo and S. Mancoridis. Gadget: A tool for extracting the dynamic structure of java programs. In *Conference on Software Engineering and Knowledge Engineering*, Buenos Aires, Argentina, June 2001.

[18] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub Co, Jan 1989.

[19] 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)*, pages 82–87, New York, USA, July 2002.

[20] J. Horn, N. Nafpliotis, and D. E. Goldberg. A Niched Pareto Genetic Algorithm for Multiobjective Optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, volume 1, pages 82–87, Piscataway, New Jersey, 1994. IEEE Service Center.

[21] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[22] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[23] L. Kaufman and P. Rousseeew. *Finding groups in data: an introduction to cluster analysis*. Wiley-Inter Science, Wiley - NY, 1990.

[24] 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*, 1998.

[25] T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the Association for Computing Machinery*, 31(6), June 1988.

[26] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of IEEE International Conference on Software Maintenance*, Montréal, QC, Canada, October 2002.

[27] T. Systä. On the relationships between static and dynamic models in reverse engineering java software. In *Proceedings of IEEE Working Conference on Reverse Engineering*, Atlanta, GA, USA, October 1999.

[28] E. Talbi and P. Bessire. A parallel genetic algorithm for the graph partitioning problem. In *ACM International Conference on Supercomputing*, Cologne, Germany, 1991.

[29] V. Tzerpos and R. C. Holt. Software botryology: Automatic clustering of software systems. In *DEXA Workshop*, pages 811–818, 1998.

[30] M. Wall. GAlib - a C++ library of genetic algorithm components. `http://lancet.mit.edu/ga/`.

[31] S. M. Yacoub, H. Ammar, and T. Robinson. Dynamic metrics for object oriented designs. In *Proceedings of the Sixth International Software Metrics Symposium*, pages 50–61, Boca Raton, FL, USA, November 4-6 1999.

[32] E. Zitzler, K. Deb, and L. Thiele. Comparison of Multiobjective Evolutionary Algorithms on Test Functions of Different Difficulty. In A. S. Wu, editor, *Proceedings of the 1999 Genetic and Evolutionary Computation Conference. Workshop Program*, pages 121–122, Orlando, Florida, 1999.