A Search-Based Approach for Dynamically Re-packaging Downloadable Applications

Thierry Bodhuin, Massimiliano Di Penta, Luigi Troiano
bodhuin@unisannio.it, dipenta@unisannio.it, troiano@unisannio.it

RCOST – Research Centre on Software Technology
University of Sannio — Palazzo ex Poste, Via Traiano 82100 Benevento, Italy

Abstract

Mechanisms such as Java Web Start enable on-the-fly downloading and execution of applications installed on remote servers, without the need for having them installed on the local machine.

The rapid diffusion of mobile devices (e.g., Personal Device Assistants - PDAs) connected to the Internet make these applications appealing to mobile users. However, in many cases the available bandwidth is limited, and its excessive usage can even be a cost when wireless connections are paid on a Kbyte transfer basis.

This paper proposes an approach based on Genetic Algorithms and an environment that, based on previous usage information of the application (i.e. scenarios), re-packages it with the objective of limiting amount of resources transmitted for using a set of application features. The paper reports an empirical study on the application of the proposed approach on three medium-sized Java applications.

Keywords: Mobile Applications, Re-packaging, Search-Based Software Engineering

1 Introduction

The last few years have been characterized by a large diffusion of mobile devices such as Portable Digital Assistants (PDAs) or smart phones. Most of these devices have been initially conceived for providing a well determined set of features, such as dialing a phone call or retrieving a phone number from an address book. During the recent years the complexity of these devices has increased dramatically. Both smart phones and PDAs are provided with a complete operating system, a reasonable amount of memory, an Internet connectivity — using WIFI, General Packet Radio Service (GPRS) or Universal Mobile Telecommunication System (UMTS) connections. Also, the support for developing applications suitable for these devices has been improved with environments like J2ME or other proprietary software development kits based on the Symbian OS or the Windows Pocket PC operating system. As a consequence, applications for various purposes (productivity, games, route planning) have been developed and are available.

The Java technology allows the user for downloading on-the-fly an application, or part of it. In some cases the application can be a client-server application and, thus, the user has just to download its client-side. When an application is distributed and executed on-line, as in the case of applets or Java Web Start [21] applications, the user usually downloads a set of jar archives containing a client-side application, while the server-side may or may not exist. However, some categories of users never access some features of the application, and the download of the associated code may be useless. For example, some users may not be interested in advanced features. In these cases, the perceived download time is crucial for the application. Forcing the user to download parts of the application that will never be used presents serious problems:

1. a waste of memory used on the client side;
2. an increasing of time/cost for the downloading.

Therefore, the user would like to avoid download-

ing useless resources. Actually, additional features may be downloaded on-demand lateron if needed, as in the case of the Java Web Start technology [21]. Two different ways of code downloading are usually used: Eager or Lazy code downloading [15]. Eager downloading means that the code is downloaded before the application begins, while Lazy downloading means that the code will be downloaded only when the application will really need it. The Java Web Start application loads Jar files that are either eager or lazy.

In this paper, we propose an approach that tackles the problem of reducing the downloaded code from networked applications organizing in optimal way the jars' contents. The overall idea is to cluster together (in jars) classes that, for a set of usage scenarios, are likely to be used together. In other words, each time a jar is downloaded because the client needs a class contained into it, the jar's other classes should likely be used soon.

The jar clustering problem has two extreme solutions, each one presenting strengths and weaknesses:

1. classes are singularly accessible: this allows the user not to download classes not needed; each class is downloaded when needed. On the other hand, there will be a *request* overhead each time a class is needed;
2. put together all classes in a single jar: this reduces the *request overhead* to its minimum; on the other hand, the user will have to download unneeded classes.

We need to pursue a compromise between the two extremes above — i.e., creating a set of jars so that, within a given scenario, the overall download is kept low and, at the same time, the *request overhead* as well. Finding a solution to this problem is known to be NP-hard. We propose to cluster together classes according to dynamic information obtained from executing a series of usage scenarios. After having collected execution traces, the proposed approach determines a preliminary repackaging considering common class usages, then improves it by using Genetic Algorithms (GAs).

The research contributions of this paper can be summarized in:

- a search-based approach to cluster together classes into jar;
- a tool support for enacting the proposed approach and for visualizing class usages in the different scenarios and generating optimized jars;
- case studies related to clustering in jars classes for three medium-sized software systems.

The paper is organized as follows. Section 2 gives an overview of the distribution of Java applications using the Java Web Start technology and the related class downloading mechanism. Section 3 describes the proposed approach, giving an overall picture, describing how dynamic information is collected and how it can be used to re–package classes into jars. Section 4 reports the three case studies performed to assess the approach. After a review of the related work and a comparison with them in Section 5, Section 6 concludes.

2 Distributing applications using Java Web Start

Java Web Start [21] is a Java application that runs as a Web Browser Plug-in that uses the eager and lazy downloading mechanisms and executes network–downloaded applications. The Java Web Start application loads Jar files that are either eager or lazy, which means that they can be downloaded later. The meaning of later in this context is unclear, as there is no information about the content of jar files before they are actually downloaded. Till the Java Standard Edition version 1.6, Java Web Start had no real lazy downloading scheme. In fact, the first time the application needs a resource, it will download all the lazy jar files. This is unacceptable for large applications, which may have many features, and so it limits Java Web Start applications to applet-sized applications. From the Java Standard Edition version 1.3, an indexing mechanism [20] for jar files was introduced to optimize the class searching process of class loaders for network applications, especially applets. The jar tool was enhanced to be able to examine a list of jar files and generate directory information as to which classes and resources reside in which jar file. This directory information is stored in a simple text file named INDEX.LIST in the META-INF directory of the root jar file, and the class loader uses such information to find the proper jar file, and then downloads it if necessary.

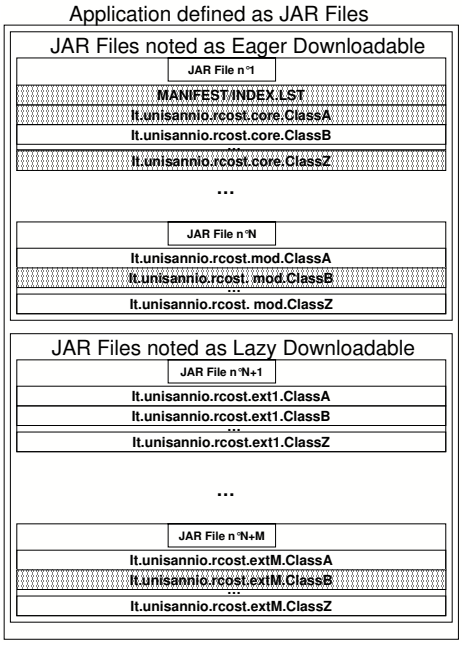


Figure 1: Eager/Lazy Downloading

However, till the Java Standard Edition version 1.6, Java Web Start will not support Jar Indexing. Even with this Java version, the class downloading process will download the full jar containing the requested class. Organizing the content of jars based on the class dependencies or on the downloading order of the requested classes would be an enhancement to networked application startup and memory footprint reduction.

In the case of the Figure "Application defined as JAR Files", where the application is defined by all the jars residing on the server numbered from 1 to N+M, the jar numbered from 1 to N are marked as *Eager*, while the jars numbered from N+1 to N+M are marked as *Lazy*. The classes shown with a special background (clear with tiles) are the classes really needed during the execution of the application. The current version of Java Web Start first downloads all the jars numbered from 1 to N, then when the class `"it.unisannio.rcost.extM.ClassB"` will
be requested it will download all the jars marked as *Lazy*, i.e., the jars numbered from N+1 to N+M. The Java Web Start 1.6, after having downloaded all the jars numbered from 1 to N, would download only the jar numbered N+M, when the class `"it.unisannio.rcost.extM.ClassB"` will be requested, downloading in the same time the classes `"it.unisannio.rcost.extM.ClassA"`, `"it.unisannio.rcost.extM.ClassC"` to `"it.unisannio.rcost.extM.ClassZ"` even if these classes may not be used.

3 The proposed approach

As stated in the introduction, the proposed approach aims at grouping in jars classes that are used together during the execution of a scenario, with the purpose of minimizing the overall jar downloading cost, in terms of time in seconds for downloading the application. The whole approach can be viewed in the context of migrating a Java application towards a distributed, mobile environment, where the client (installed, for example, on a smartphone or on a PDA) only contains the core part of the application, while the other modules are downloaded from the server when needed. In our model the downloading cost includes:

1. an overhead cost due to the request made to the server (i.e., network delay). Each time a new jar is requested, this overhead needs to be considered. The higher is the number of jars produced, the higher will result the total overhead. This means that an excessive fragmentation of classes in many jars, while ensuring that each scenario only gets the classes effectively needed, also causes a large request overhead;
2. a variable part, equal to the sum of the sizes of all the downloaded classes together with their Jar-related information divided by the network bandwidth available.

The first cost can be considered as constant and dependent on the communication protocol while the second is proportional to the downloaded size of the classese and jar-related informations compressed in the jar file. Let us now suppose that a solution to our problem composed of n jar archives Ar_j, each one composed of m_j classes. let us consider an *interaction scenario* S_i and define the function $use_{i,j}$ such that:

$$use_{i,j} = \begin{cases} 1 & Scenario_i\ uses\ archive_j \\ 0 & Otherwise \end{cases} \quad (1)$$

the cost for such a scenario (in sec.) is given by:

$$Cost_i = \sum_{j=1}^{n} use_{i,j} \cdot \left(\frac{\sum_{k=1}^{m_j} size(c_k)}{N_b} + N_d\right) \quad (2)$$

where $size(c_k)$ is the size of the $k-th$ class with jar-related information compressed and contained in the $j-th$ archive, and N_d is the overhead due to the archive request and N_b is the network bandwidth available.

The proposed approach is composed of the following steps:

1. the application to be analyzed is instrumented, and then it is exercised by executing several scenarios instantiated from use cases (see Section 3.1);
2. a preliminary solution of the problem is found, grouping together classes used by the same set of scenarios (Section 3.2);
3. GAs are used to determine the (sub)-optimal set of jars (Section 3.3); and, finally
4. based on the results of the previous steps, jars are created.

3.1 Collecting dynamic information by exercising scenarios

To re-package jars, we first need to collect dynamic information from application's execution. We used run-time instrumentation on the Java bytecode; the instrumentation uses dynamic transformation of code when the software system is executed [6]. While the application is executed, the instrumented code collects information dynamically. For our re-packaging work, we only used a subset of the collected information that correspond to the names of loaded classes with their jar location name, the size of their java bytecode and the class loading time. From this information we can deduce the total set of classes that are used in all scenarios, that is a subset of the complete application, the set of classes used in each scenario and the size of bytecode downloaded for each scenario.

In order to properly collect this information and use it for re-packaging purposes, the application execution should take place according to realistic operations, so that information collected corresponds to a use case scenario of the application itself. Different scenarios should be defined in order to exercise the application's features. Use case descriptions or user manuals can be used for this purpose.

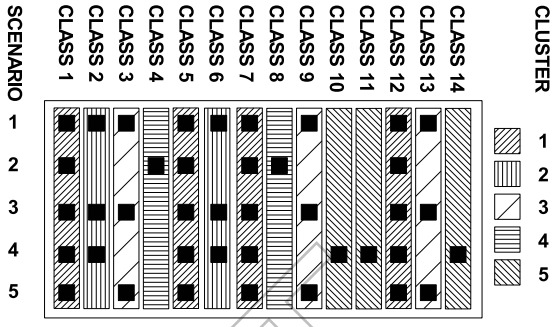


Figure 2: Building the initial solution

3.2 Building a preliminary re-packaging

The first, easiest way to re-package classes is to determine groups of classes used by the same set of scenarios. This task is quite cheap to be performed using a Hashtable where the key is a class name and the value is the list of scenarios using the class. For each scenario, the following tasks are performed:

1. the list of classes used by the scenario is scanned;
2. if the class is not present in the hashtable, it is added using its name as key and the scenario id as value;
3. if the class is present in the hashtable, the scenario id is appended to the value corresponding to the class.

After scanning all scenarios, the classes having the same value in the hashtable will be grouped together. Figure 2 shows a simple example where classes used by common sets of scenarios are clustered together.

3.3 Improving the re-packaging using Genetic Algorithms

The initial solution to our optimization problem can be further improved by grouping together some of the class clusters obtained in the previous step. For instance, in case some groups are too small, further grouping could reduce the jar overhead. This paper pursues this goal by using Genetic Algorithms (GAs)[1].

GAs originated with an idea, born over 30 years ago, of applying the biological principle of evolution to artificial systems. Roughly speaking, a GA is 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, named the *chromosome*. 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* and selected through a *selection operator*. High-fitness individuals will have the highest probability to reproduce. Reproduction is made by means of *crossover* (that produces an offspring recombining parents) and *mutation* (that generates a new individual by mutating an old one) operators. Further details on GA can be found, for example, in the Goldberg's book [11].

To solve our optimization problem with GAs, we first need to represent our problem with a proper chromosome. We used an integer array chromosome (see Figure 3), where each item (gene) g_j represents a cluster of classes (generated as described in Section 3.2). The integer value $g_j = k$ assigned to the j-th gene provides a reference to the jar archive k to which the class cluster j is being assigned. At most it is possible to have different values for all genes (i.e., a jar for each cluster). The adopted representation is similar to the one used in the paper [3] to represent staffing distribution. We assume that there is no cloning of clusters/classes across jar archives, while such a cloning can, in some case, help to pursue decoupling [4, 9].

The above defined representation allows *isomorphic* solutions. Two solutions p and q are isomorphic, if it possible to get $p = h(q)$, where $h(\cdot)$ represent the relabeling of jar archives (see for example Figure 4). Isomorphism introduces both advantages and disadvantages in evolutionary heuristics:

[1] Attempts we made using other heuristics such as hill-climbing or simulated annealing gave us no better results.

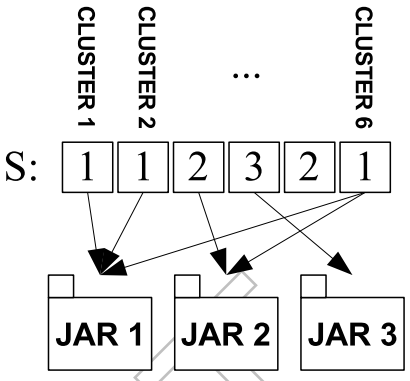


Figure 3: Integer chromosome

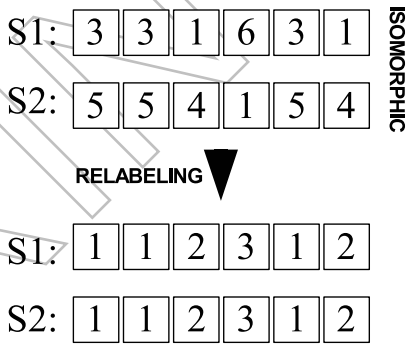


Figure 4: Isomorphic allocations

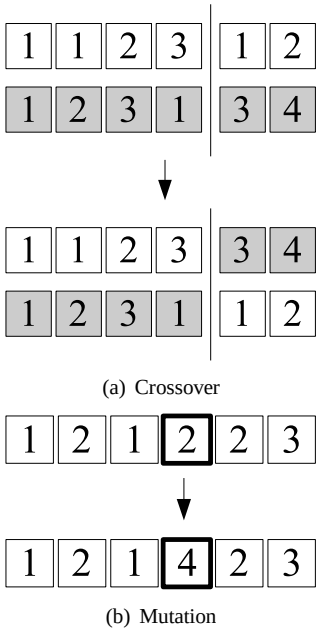
if it enriches the genetic variety of the population, it may make the landscape search rough. In this work we decided to remove isomorphism by relabeling genes as shown in Figure 4.

The GA search is guided by a cost function (to be minimized), that is the average of downloading times, for a particular solution (i.e., class repackaging), over a set of scenario (that we assume having equal likelihood of being executed — although a weighted average can be used for scenarios executed with different frequencies):

$$f(x) = \frac{1}{N} \sum_{i=1}^{N} Cost_i \qquad (3)$$

where N is the number of scenarios and $Cost_i$ is provided by Equation 2.

We used a simple GA with an elitism of 0.1, i.e. we kept the best 10% individuals alive across subsequent generations. Individuals to be reproduced were selected using a roulette-wheel selection, i.e., better individuals have higher probability of being



(a) Crossover

(b) Mutation

Figure 5: Crossover and Mutation Operators

selected. After the selection is performed, new solutions are generated by crossover. The crossover operator is the standard single-point crossover. After a crossover point is randomly chosen, two new solutions are obtained by switching the following genes, as depicted in Figure 5-a.

The mutation operator selects a cluster of classes and randomly changes its allocation to another jar archive, as depicted in Figure 5-b. We adopted a crossover rate of 100%, a mutation rate of 10%, a population of 300 individuals and 200 GA generations. The starting population is composed of randomly generated individuals, i.e., random assignments of clusters to jars.

3.4 Tool support

The approach defined in the previous sections relies on a series of tools, all developed in Java, namely:

1. **dynamic trace extractor.** As described in Section 3.1, the dynamic trace extractor tool makes instrumentation of Java applications by a client-server, event-based approach (events are generated at each class loading and method invocation);
2. **package optimizer.** Exploits dynamic information to determine the (sub) optimal grouping of classes into jars, following the approach described in Sections 3.2 and 3.3; and
3. **visualizer.** Permits the graphical visualization of class usages, and acts as a front-end for the package optimizer. The visualizer belongs to the PODoJA tool that implements the full approach and that has been described in reference [5].

4 Case study

The purpose of the case study hereby reported is to investigate under which extent the re-packaging of classes in jars can reduce the downloading time when porting a Java application in a mobile environment through Java Web Start. The following subsections describe the experimental context, outline the research questions and finally report and discuss the obtained results.

4.1 Context description

The study reported in this paper regards the reorganization of java classes as well as other resource types for three medium-sized software system: ArgoUML[2], an ERP&CRM called Compiere[3] and an internally developed Home Automation system[4]. ArgoUML is an open-source multiplatform, UML environment written in Java. Compiere is an ERP&CRM (Enterprise Resource Planning and Customer Relationship Management) open-source solution and a client-server application type with a database back-end. Our internally developed Home Automation system is a distributed application using the Jini Extensible Remote Invocation [18] for distributed communications. All the three applications under study are medium-size software system and are available with a application client-side part as a Java Web Start application. ArgoUML is composed of 17 jar files for a total of 7.24 MB, and we executed 8 different scenarios for exercising the application. Compiere (the downloadable application client-side part) is composed of 2 jar files for a total of 9.85 MB and we executed 15 different scenarios. Home Automation (downloadable client-side part) is composed of 21 jar files, for a total of 7.03 MB,

[2] http://argouml.tigris.org/
[3] http://www.compiere.org/
[4] http://domotica.rcost.unisannio.it/

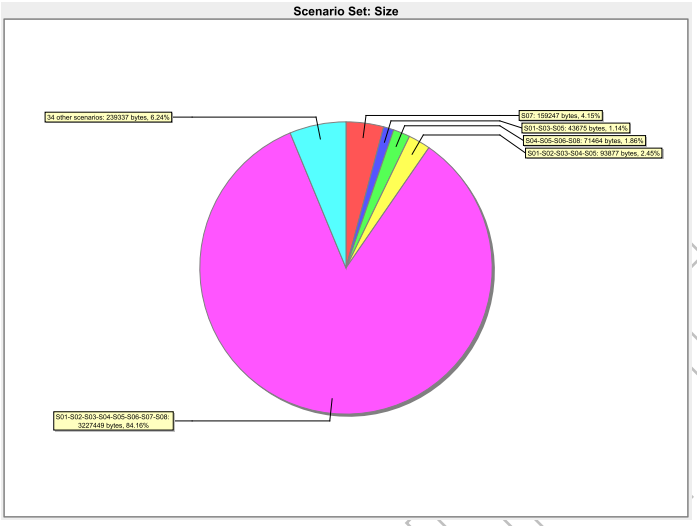


Figure 6: Resources used by different scenarios for ArgoUML

we executed 5 different scenarios. The three systems under study uses different types of resources, for instance: java class byte code, XML/DTD configuration files, images files. Figures 6, 7, and 8 show the distribution, with total size of resources, of the scenarios set that uses commonly a set of resources and shows that the Compiere and ArgoUML systems have both a large central corpus of resources used in all the scenarios (48% for Compiere and 84% for ArgoUML) while "only" 24% in the case of the Home Automation system.

Dynamic information were obtained for each of three software system by executing different scenarios trying to cover at most the application features.

For ArgoUML, all the scenarios start with "Loading the ArgoUML version 0.24 from the network" and ends with "Closing the application", then the scenarios include the following flows of interaction:

1. creation of a class diagram and saving of a project;
2. creation of an activity, exporting the project as a XMI file and saving of a project;
3. creation of an activity, exporting the project as a XMI file and saving of a project;
4. loading a project from a ".zargo" file, Disabling and Enabling critics functionality, visualization of the critics and saving of a project;
5. importing the project from a XMI file, Generation of Java code and saving of a project;
6. importing the project from a XMI file and Export all the graphics (from the File menu);
7. importing from Java code (Creation of UML from Java Files);
8. loading a project from a ".zargo" file and printing.

For Compiere, all the scenarios start with "Loading the Compiere application from the network" and ends with "Closing the application", then the scenarios have included the following features:

1. consulting the product catalog;
2. inserting an order in the system;
3. searching and viewing information about a seller;
4. visualization of the pending requests and payments. Preference modification;
5. verification of production orders and inventory movements. Confirmation of inventory movements;
6. production request management and report printing in pdf;
7. visualization of pending bills. Opening of a bill. Specifying the payment modality. Closing of the bill;

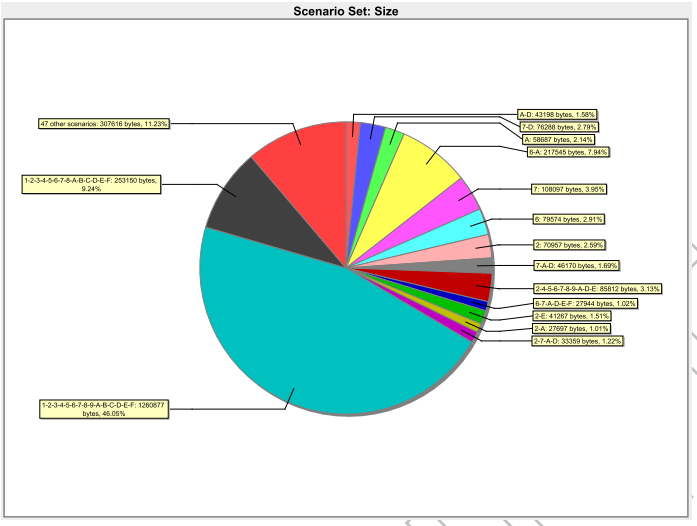


Figure 7: Resources used by different scenarios for Compiere

8. visualization of the details of a bill. Visualization of cashing machine status;
9. visualization of the assets, cashing machine status and resources. Obtaining details for a business partner and a product;
10. advanced management of a selling order and report generation in pdf;
11. analysis of the performance of the day and use of an editing tool;
12. advance status of an activity of workflow and editor tool usage, screen capture and print screen;
13. visualization of a shipment. Bill control. Closing of the shipment process;
14. verification of the security rules of the system. Editor usage;
15. workflow modification and test.

For Home Automation, all the scenarios start with “Loading the Home Automation application from the network” and ends with “Closing the application”, then the scenarios have included the following features:

1. open Locations/RCOST/Device tree, cClick on the HAIDimmer19, open the chart and set to 50% the brightness;
2. open Devices tree, Click on the Axis51, open the properties editor and select the positions: CCBubo (video camera do Pan/Tilt and zoom);
3. open Locations/RCOST/RCOST_2nd_Floor tree, select RCOST_2nd_Floor to view the map, click on the map “HAIVolumetric Sensor”;
4. open Locations/RCOST/RCOST_2nd_Floor tree, select RCOST_2nd_Floor to view the map, click on the map “RFIDPassive_1”, click on the tree devices menu “Axis51” and select the positions: CCBubo (video camera do Pan/Tilt and zoom);
5. open Locations/RCOST/RCOST_2nd_Floor tree, select RCOST_2nd_Floor to view the map, open service tree, select the navigation service, select identity bodhuin, initial location: RCOST_2nd_Floor/DomoticaRoom/BedRoom and Destination: RCOST_2nd_Floor/Studio1, select VirtualNavigator for “Available Moveables” and click on “–”, click on the navigate button and wait until the navigating entity arrive at destination viewing the RCOST_2nd_Floor map.

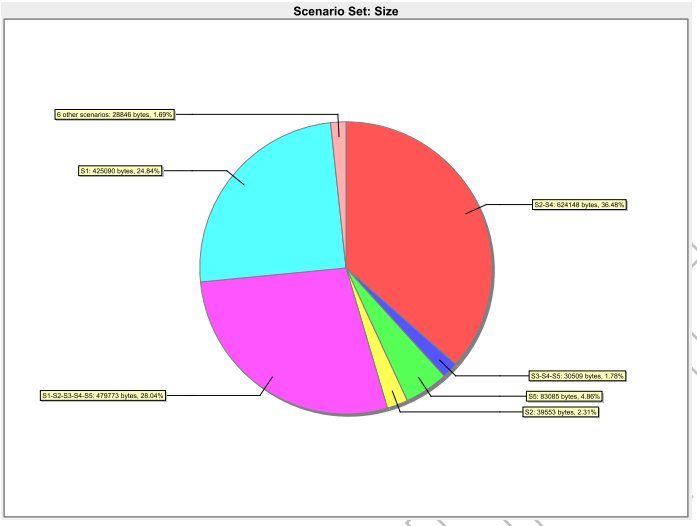


Figure 8: Resources used by different scenarios for HomeAutomation

4.2 Research Questions

The research question this study aims to answer are the following:

- **RQ1:** Given the preliminary class repackaging based on their usage, to what extent GAs are able to improve such a re-packaging?
- **RQ2:** How performance changes when the network delay increases?

4.3 Results

The analysis of execution traces described in Section 3.2 permitted the creation of a preliminary grouping of the resources: 39 jars for ArgoUML, 62 jars for Compiere and 12 jars for Home Automation.

4.3.1 RQ1: Given the preliminary class repackaging based on their usage, to what extent GAs are able to improve such a re-packaging?

As it can be seen in Figures 9, 10, and 11, the preliminary solution can be further improved by using GA as there may be in fact cases of "small" jars that can be grouped together with the aim of saving some overhead. Figures 9, 10, and 11 show the evolution of the cost function through 200 generations of the GA. To avoid randomness, GA runs are repeated 20 times and the figures include the maximum, minimum and average values for the cost function. It is worth noticing that the solution at generation 1 is anyway less expensive than the one obtained by the preliminary optimized solution, since the initial population of the GA has been randomly generated but including also the preliminary optimized solution. Giving the current network delay (N_b=0.2 s), the downloading time for each packaging type and specially the one achieved by using GA, is summarized in Table 1.

Packaging Type Application/	Initial Packaging	Used	Prelim. Optim.	GA
ArgoUML	38.08	15.54	17.30	14.71
Compiere	39.80	11.15	10.53	8.13
Home Automation	39.00	7.04	4.61	3.98

Table 1: Downloading time in seconds for each system and packaging types

"Initial Packaging" corresponds to the packaging provided currently by the original developers, "Used" corresponds to a packaging that is composed of a two jars, one called "Used.jar" that contains all the resources used by all the scenarios and one called "Unused.jar" that contains all the other resources unused in these scenarios but that may

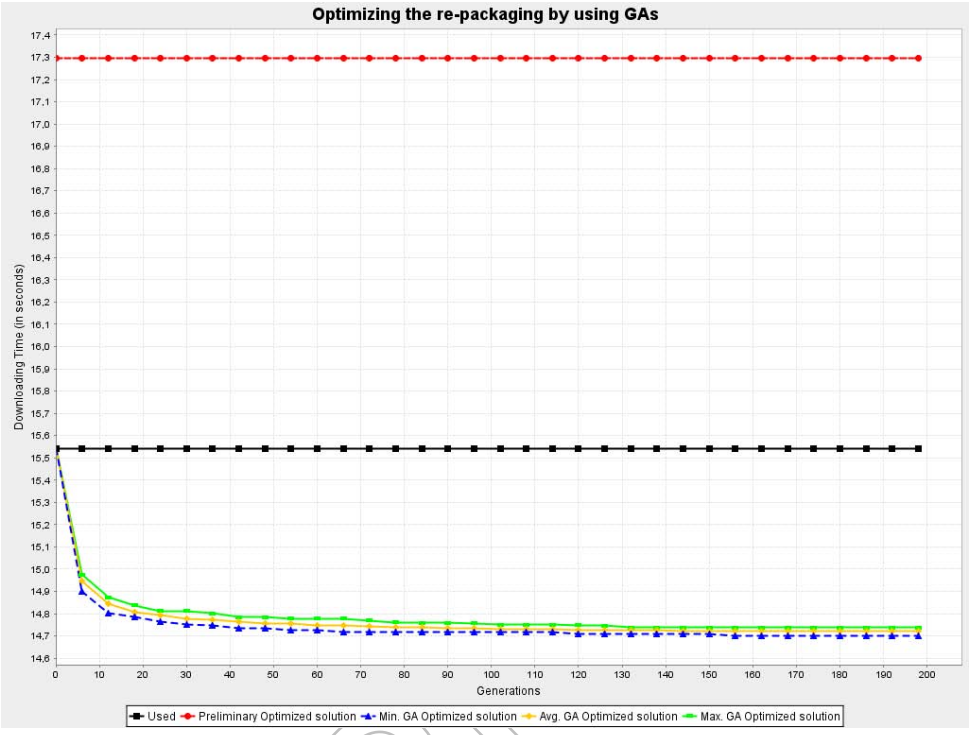


Figure 9: Optimizing the re-packaging by using GAs for ArgoUML

be used if the scenarios are extended by other features not yet used. "Preliminary Optimized Solution" corresponds to the packaging optimized as in Section 3.2. "GA" is the packaging that has been optimized by our GA. By using our approach, we have been able to reduce the downloading time, compared to the "Initial packaging", of:

- 61.37% for ArgoUML;
- 79.58% for Compiere; and
- 89.79% for Home Automation.

while compared to the "Used" packaging, we have been able to achieve a further reduction of:

- 5.35% for ArgoUML;
- 27.14% for Compiere; and
- 43.48% for Home Automation

and compared to the "Preliminary Optimized Solution", we have been able to achieve a further reduction of:

- 14.96% for ArgoUML;
- 22.83% for Compiere; and
- 13.65% for Home Automation

These results are not surprising, it is clear that very often the developers do not optimize the packaging of their application in order to reduce the downloading size by removing all the resources, from the packaging, that are never used. For the three cases between 59% and 82% could be saved by removing the unused resources, for any of the executed scenarios, from the packaging. Even if all the three systems under study have a large central corpus of resources used in all scenarios, the improvement in terms of downloading time has been good compared to the "Used" packaging. In all the three systems, the preliminary optimized solution already produced an effective re-packaging, that GA could even optimize by at least 13%.

The re-packaging optimization process using scenario-based information and GA is clearly useful in reducing the downloading time for these three medium-size applications. The downloading time may be reduced using this approach particularly:

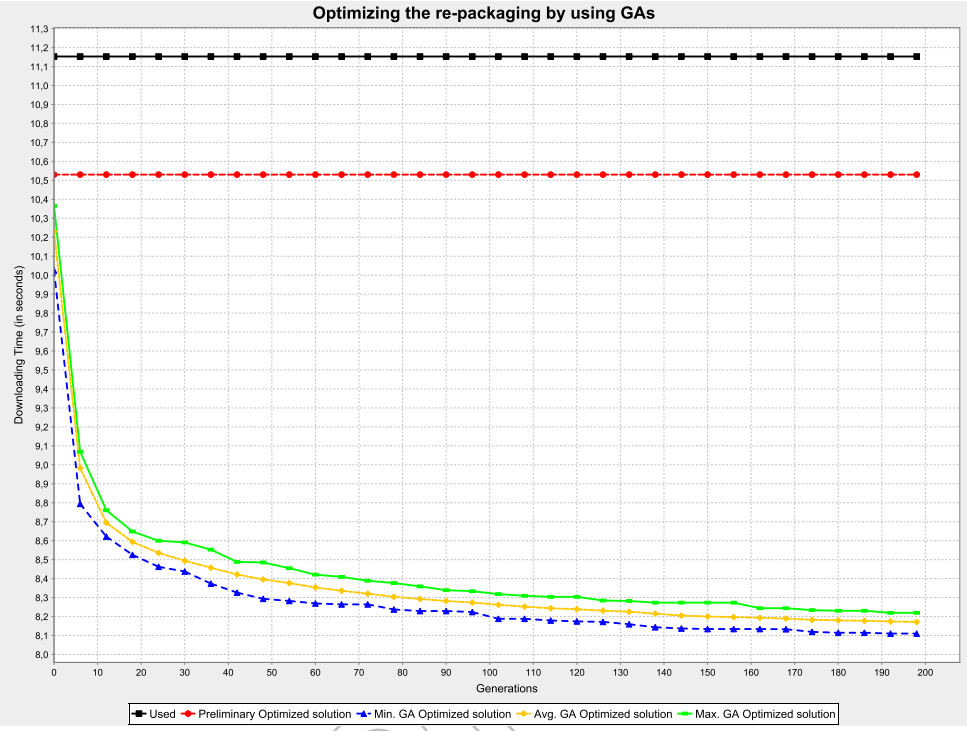


Figure 10: Optimizing the re-packaging by using GAs for Compiere

- when the dynamic information used to perform re-packaging is collected through a large number of scenarios (needed especially when the application is quite complex). In that case, the preliminary re-packaging produces a large number of jars, that can cause an unacceptable downloading time overhead.
- when the set of resources used by all scenarios is limited. This is the case of applications having several (almost) decoupled features that share a limited sets of utility classes, or the case of three-tier application where the classes belonging to the business logic layer classes are decoupled, and the different features share classes of the data layer.

4.3.2 RQ2: How performance changes when the network delay increases?

When the network delay (due to communication protocol) is by far larger than the one considered in the GA optimization (0.2 s). Figure 12 reports a sensitivity analysis we performed with the purpose of comparing the "Preliminary optimized solution", the "Used" packaging and the GA optimization in case that the network delay is different than the value used for the optimization process. As shown, when the network delay increases, the GA optimization starts to be highly more useful compared to the "Preliminary optimized solution" while the "Used" packaging becomes better. However for network delay value lower or slightly higher than the value used for the optimization process, the GA optimization is always the best packaging option.

4.4 Threats to Validity

This section discusses the main validity threats of our case studies, mainly internal, and external validity threats.

Internal validity threats, are related to the causal relationship between what measured, i.e., the performance improvement and the bandwidth usage

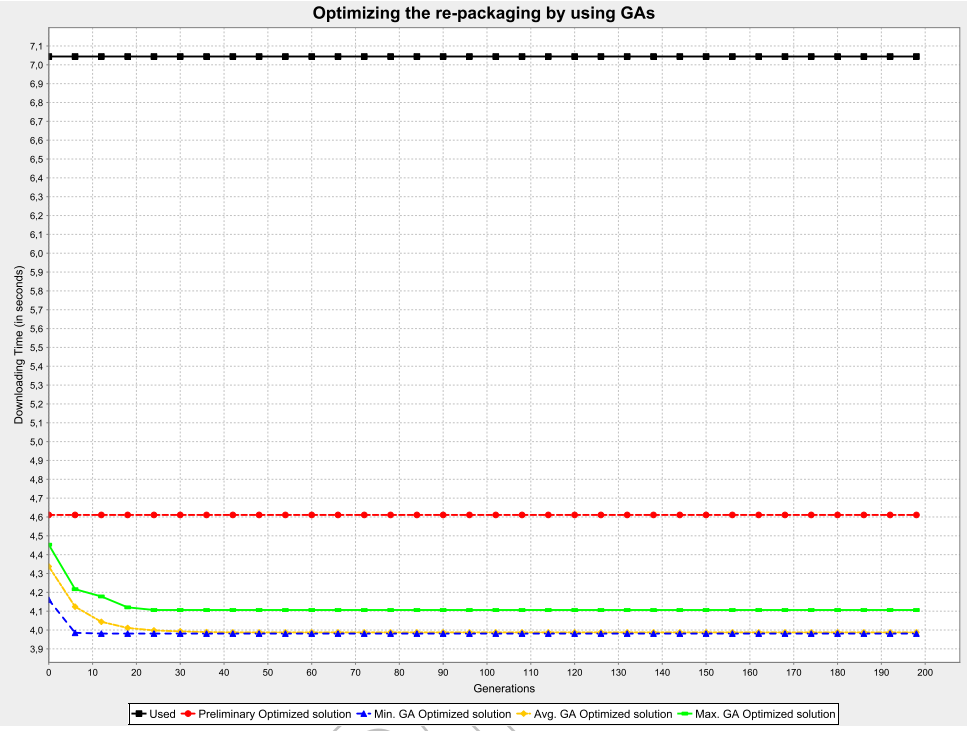


Figure 11: Optimizing the re-packaging by using GAs for Home Automation

reduction, and the treatment, i.e., the repackaging. Threats can be due to the inherent randomness of the search-based algorithms. This was limited by repeating each simulation 20 times and showing descriptive statistics.

External Validity threats are related to extend the obtained results beyond our case studies. These threats can be due to the particular applications we chose as case studies and to the particular scenarios. Certainly, other applications and, above all, different execution scenarios could have lead to different performance improvement. However, it is the authors' opinion that the three systems chosen constitute a pretty variegated set of systems having different size and belonging to different domains.

5 Related work

The proposed approach is related to works in the field of software clustering, software miniaturization and porting of applications on mobile devices.

Different software clustering approaches have been developed in the past for various purposes. Surveys have been presented by Wiggerts [27] and by Tzerpos and Holt [26]. Anquetil and Lethbridge in [1] devised a method, relying on file names, for decomposing complex software systems into independent subsystems. An approach relying on inter-module and intra-module dependency graphs to refactor software systems was presented by Mancoridis et al. [14]. GAs were used by Doval et al. [10] to identify clusters on software systems. Finally, Harman et al. [12] reported experiments of modularization and remodularization by comparing GAs with hill climbing techniques and by introducing a representation and a crossover operator tied to the remodularization problem. Their case studies revealed that hill climbing outperformed GAs. Mahdavi et al. [13] proposed an approach aimed to combine multiple hill climbs for subsequent searches, thus reducing the search spaces. While the above mentioned works aimed at remodularizing software systems (and thus cohesion of

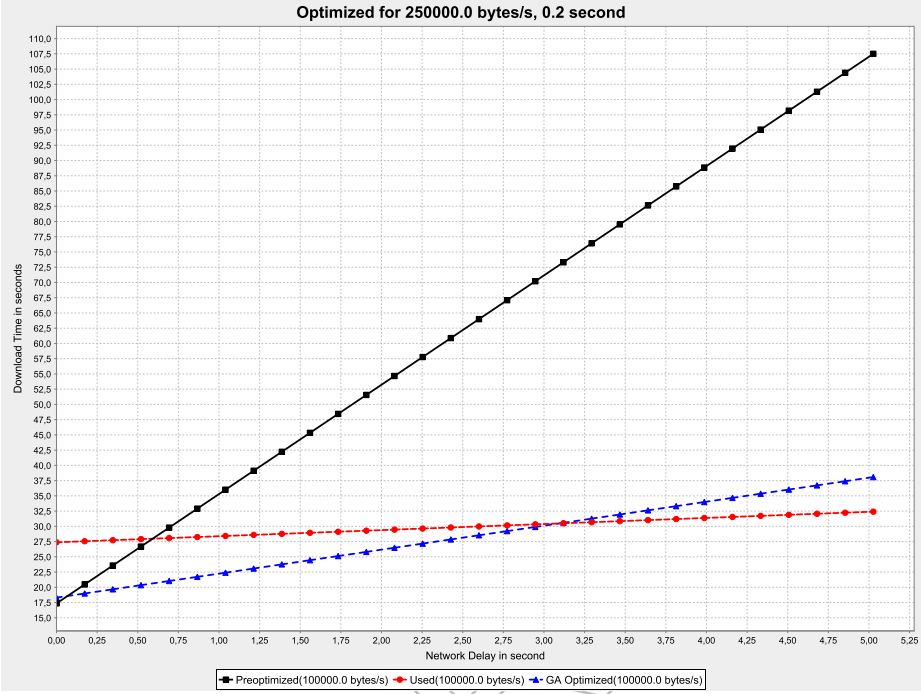


Figure 12: What happens varying the network delay with a lower transmission bandwidth (Compiere: Optimized for 250KB/s, 0.2 s of network delay and real bandwidth of 100KB/s)

the obtained modules is important) our aim is not to achieve logical cohesion inside each jar, that can be even automatically generated when deploying the application through Java Web Start, while the original application organization is kept for maintenance/evolution purposes.

Software miniaturization for Java application has been addressed by *Jax* which is an application extractor for Java software systems [25] whose goal is the size reduction of Java programs with particular interest to applets to be transmitted over the network. *Jax* is based on transformations including removal of redundant methods and fields, devirtualization and inlining of method calls, renaming methods, fields, class and packages, and transforming class hierarchies. Recently, the PACK200 compression scheme [22] has been defined for compressing specifically jar files containing mainly java byte codes. Another approach, devoted to reduce the size of Java libraries for embedded systems, was proposed by Rayside and Kontogiannis in [17]. Di Penta et al [8, 9] presented a framework, using various techniques such as clustering, clone detection and GAs, for reducing the application's footprint when porting software system on devices with limited memory. Antoniol and Di Penta [2] extended the previous work by combining static and dynamic application for remodularization's purposes. We share with these works the idea of porting applications on devices with limited space and the use of dynamic information. However, our aim here is to distribute the application between a (mobile) client and a server, and to optimize the usage of bandwidth when downloading classes.

Application porting to mobile devices has been investigated in order to partition the code execution, considering limitations of mobile devices such as power consumption and size restrictions which lead to other constraints such as processor speed and memory size. The main idea is to partition the application in order to deliver code parts to mobile devices, keeping the remaining parts running on the network infrastructure. Different solu-

tions have been proposed. For instance, Philippsen and Zenger [16] proposed a solution for partitioning Java applications at source code level. A similar solution has been proposed by Spiegel [19]. More recently Tatsubori et al. [23], and Tilevich and Smaragdakis [24] proposed application partitioning at bytecode level. Another approach, proposed by Chandra et al. [7], entails the use of a proxy just-in-time (JIT) compiler, in order to provide optimized native code to mobile devices.

6 Conclusions and work-in-progress

This paper described an approach exploiting dynamic information to re-package Java classes into jars. In the context of porting the application towards a distributed, mobile environment through Java Web Start, the proposed approach aims to reduce downloading cost: this is especially crucial in cases where the connection cost is proportional to the bandwidth actually used (e.g., for GPRS connections).

After collecting the dynamic information by executing the instrumented application, we first generate a preliminary re-packaging by putting together resources used by common sets of scenarios, and then we use GAs to improve such a re-packaging.

We performed a case study composed of three medium-sized software systems to evaluate the approach. We found that, even when there is a large corpus of classes used in all scenarios, a cost reduction is still possible, even if in such a case the preliminary optimized solution is already a good solution. Clearly, the benefits of the proposed approach strongly depends on several factors, such as a) the amount of dynamic information collected and the number of scenarios subjects of the analysis, b) the size of the "common corpus" and c) the network delay.

In conclusion, we found that the presented approach has been successful for reducing the packaging size in all the three software systems that have been subject of the study. The approach could be used inside application servers for automatically optimizing the packaging of Java Web Start-based software systems by using scenarios usage information that could be dynamically obtained.

References

[1] Nicolas Anquetil and Timothy Lethbridge. Extracting concepts from file names; a new file clustering criterion. In *Proceedings of the International Conference on Software Engineering*, pages 84–93. IEEE Computer Society Press, Los Alamitos, CA, USA, April 1998.

[2] G. Antoniol and M. Di Penta. Library miniaturization using static and dynamic information. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 235–244, Amsterdam, The Netherlands, Oct 2003.

[3] G. Antoniol, M. Di Penta, and M. Harman. A robust search-based approach to project management in the presence of abandonment, rework, error and uncertainty. In *Software Metrics*, September 2004.

[4] 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.

[5] T. Bodhuin. PODoJA: Packaging optimizer for downloadable java applications. *Proceedings of IEEE Working Conference on Reverse Engineering*, Oct 23-27 2006.

[6] T. Bodhuin and M. Tortorella. A tool for static and dynamic model extraction and impact analysis. In *European Conference on Software Maintenance and Reengineering*, March 2005.

[7] D. Chandra, C. Fensch, W.K. Hong, L. Wang, E. Yardimci, and M. Franz. Code generation at the proxy: An infrastructure-based approach to ubiquitous mobile code. In Springer-Verlag, editor, *Proceedings of the Fifth ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS 2002)*, Malaga, Spain, 2002.

[8] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo. Knowledge-based library refactoring for an open source project. In *Proceedings of IEEE Working Conference on*

Reverse Engineering, pages 128–137, Richmond - VA, October 2002.

[9] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo. A language–independent software renovation framework. *Journal of Software and Systems, special issue from WCRE 2002, Elsevier Science*, 77:225–240, 2005.

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

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

[12] Mark Harman, Rob Hierons, and Mark 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. Springer-Verlag.

[13] K. Mahdavi, M. Harman, and R. M. Hierons. A multiple hill climbing approach to software module clustering. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 315–324, Amsterdam, The Netherlands, Sep 2003.

[14] 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 *Proceedings of the IEEE International Workshop on Program Comprehension*, 1998.

[15] M. Marinilli. *Java Deployment with JNLP and WebStart*. Sams Publishing, September 2001.

[16] Michael Philippsen and Matthias Zenger. JavaParty: transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[17] D. Rayside and K. Kontogiannis. Extracting java library subsets for deployment on embedded systems. *Science of Computer Programming*, 45(2-3):245–270, Nov/Dec 2002.

[18] F. Sommers. Call on extensible rmi–an introduction to jeri. *http://www.javaworld.com/javaworld/jw-12-2003/jw-1219-jiniology.html* .

[19] Andre Spiegel. PANGAEA: An automatic distribution front-end for JAVA. In *IPPS/SPDP Workshops*, pages 93–99, 1999.

[20] Sun Microsystems, Inc. The java archive tool - JAR indexing. *http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/jar.html*.

[21] Sun Microsystems, Inc. JavaWebStart. *http://java.sun.com/j2se/1.5.0/docs/guide/javaws/index.html*.

[22] Sun Microsystems, Inc. Pack200 - JAR packing tool. *http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/pack200.html*.

[23] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of a legacy java software. In Springer-Verlag, editor, *In European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes In Computer Science*, pages 236–255, 2001.

[24] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. CoC Tech. Report GIT-CC-02-17, Georgia Tech, 2002.

[25] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. *ACM SIGPLAN Notices*, 34(10):292–305, 1999.

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

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