

A Novel Process and its Implementation for the Multi-objective Miniaturization of Software*

Nasir Ali^{1,3}, Wei Wu^{1,3}, Giuliano Antoniol¹,
Massimiliano Di Penta², Yann-Gaël Guéhéneuc³, and Jane Huffman Hayes⁴

¹ SOCCER Lab, DGIGL, École Polytechnique de Montréal, Canada
² RCOST, University of Sannio, Italy

³ Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada

⁴ Computer Science Department, LAN, University of Kentucky, USA
E-mail: {nasir.ali, wei.wu, yann-gael.gueheneuc}@polymtl.ca,
antoniol@ieee.org, dipenta@unisannio.it, hayes@cs.uky.edu

ABSTRACT

Smart phones, gaming consoles, wireless routers are ubiquitous; the increasing diffusion of such devices with limited resources, together with society's unsatiated appetite for new applications, pushes companies to miniaturize their programs. Miniaturizing a program for a hand-held device is a time-consuming task often requiring complex decisions. Companies must accommodate conflicting constraints: customers' satisfaction may be in conflict with a device's limited storage and memory. This paper proposes a process, MoMS, for the multi-objective miniaturization of software to help developers miniaturize programs while satisfying multiple conflicting constraints. The process directs: the elicitation of customer pre-requirements, their mapping to program features, and the selection of the features to port. We present two case studies based on Pooka, an email client, and SIP Communicator, an instant messenger, to demonstrate that MoMS supports miniaturization and helps reduce effort by 77%, on average, over a manual approach.

Categories and Subject Descriptors

D2.7 [Distribution, Maintenance, and Enhancement]: Portability; Restructuring, reverse engineering, and reengineering

General Terms

Management; Measurement

*Blue text highlights the difference between this technical report and the conference paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE '11, May 21-28 2011, Waikiki, Honolulu, Hawaii
Copyright 2011 ACM — ...\$10.00.

Keywords

Software miniaturization; Requirement engineering; Feature identification; Multi-objective optimization

1. INTRODUCTION

Society's reliance and dependence on computers is nowhere more obvious than in the ubiquity of hand-held devices. The typical teenager will go no more than a few minutes per day without touching either a cell phone, an MP3 player, a gaming console, or all three (perhaps in the form of just one device). From texting to listening to music, this part of society is literally "attached" to at least one hand-held device most of the time. Society also relies heavily on "smart" devices: wireless routers, GPS navigation systems, etc. with minimal operating systems and limited storage/memory. While smart phones and MP3 players have ample storage (*e.g.*, iPhone 4 or Nokia N900 have over 30 GB of flash disk and 256 MB of memory), routers or GPS navigation systems have storage ranging between two and 64 MB, (*e.g.*, the Linksys WRT54GS v2.0 router has 40 MB while the Garmin eTrex Vista HCx hand-held GPS has 24 MB).

As many people also use desktop computers, either at home or at work, it is not surprising that many programs are ported to hand-held or other limited-resource devices. Consumers demand more programs on desktop computers, more features in newer versions of these programs, and more similar programs for their hand-held devices. However, fitting "heavy" programs into such devices is a difficult task because it requires complex decisions to satisfy many contradictory constraints [7]: the choice of the features to port constrained by the storage space, memory size, computing power, screen size, or network connectivity of the hand-held devices. For example, Microsoft Office Mobile only provides a limited set of features compared to its desktop version, *e.g.*, footnotes, endnotes, headers, footers, page breaks can neither be displayed nor added in Word Mobile. Similarly, Linksys, Netgear, and D-Link routers can run miniaturized versions of Linux (*e.g.*, OpenWRT¹) and "tiny versions of many common UNIX utilities" (*e.g.*, BusyBox²).

¹<http://openwrt.org/>

²<http://www.busybox.net/>

This paper presents MoMS, a novel process for the multi-objective miniaturization of software. MoMS directs (1) the elicitation of a set of pre-requirements (PRs) from multiple customers, including program concepts and environment, customer expectations, etc., (2) the consolidation of these PRs, (3) the identification of the implementation units corresponding to each PR (if any) to obtain features [19], (4) the identification of the device properties required by features and device constraints, (5) the selection of the features to port through a multi-objective optimization and generation of the miniaturized program.

The problem of selecting the (near) optimal set of features with the objective of satisfying customers and some resource constraints is a constrained multi-objective optimization problem. Different customers may require different features: a company might not satisfy one customer by providing her the set of required features (while meeting constraints imposed by the device) without dis-satisfying other customers. Also, satisfying some customers' PRs may cause an increase of device resource usage by the program. A project manager could painstakingly try to identify the "best" set of features satisfying most of her customers but, without an automated approach, she would never know if she has truly chosen the best set of features.

The paper is organized as follows. Section 1.2 presents a motivating example. Section 2 describes the steps of the MoMS process. Section 3 presents a reference implementation of MoMS. Section 4 presents the case studies and threats to their validity while Section 5 discusses the advantages and limitations of MoMS. Section 6 summarizes related work. Section 7 concludes and describes future work.

1.1 Contributions and Organization

The contributions of this paper are:

1. A process (MoMS), described in Section 2, supporting the selection of features to port to hand-held devices for the first time, as discussed in Section 6;
2. A reference implementation of MoMS, with state-of-the-art techniques for PRs elicitation, dependency analysis, and multi-objective optimization; described in Section 3;
3. Two case studies illustrating the MoMS process to miniaturize two open-source programs, Pooka (an email client) and SIP (an instant messenger); presented in Section 4;
4. Discussions of the influences on the miniaturization process of factors such as PR traceability and source code quality; detailed in Section 5.

1.2 Motivating Example

Let us imagine a company, MobileMail, that wants to miniaturize an email client, Pooka, to port it to some hand-held devices with disk storage limitations. We assume that MobileMail has determined that there is a market/demand for such a product and that it has: (1) the source code of Pooka, (2) access to customers who will provide PRs for Pooka, and (3) quantified constraints of the hand-held devices, for example their storage capacity.

Pre-requirement Elicitation: Not too surprisingly, MobileMail does not have a documented set of PRs for either the

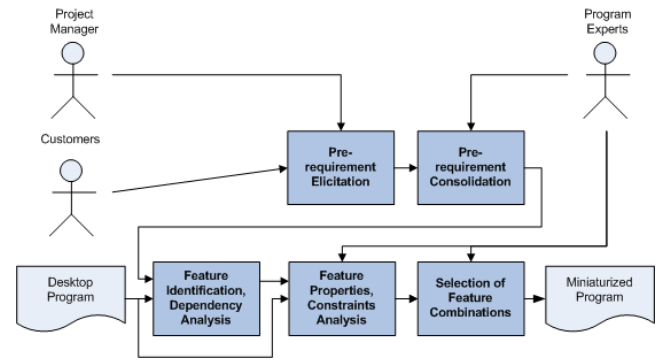


Figure 1: MoMS Process in a Nutshell

desktop or hand-held version of Pooka. Consequently, MobileMail uses available tools to elicit the PRs from some of its customers; *e.g.*, using a survey. It also assigns a value to each customer for later use in balancing their satisfaction.

Pre-requirement Consolidation: MobileMail then merges these elicited PRs into a set of unique PRs, with an indication of the number (and values) of the customers who requested them. Also, MobileMail distinguishes *compulsory* PRs, without which an email client would be of no interest, *e.g.*, sending and receiving emails, from *optional* ones.

Feature Identification: Next, MobileMail determines what classes in the Java source code of desktop Pooka must be part of hand-held Pooka, *i.e.*, what classes implement the PRs. If no classes can be found, then the implementation-less PRs are put aside as requests for enhancement. If some classes can be found, then MobileMail performs dependency analysis to identify all the classes implementing the PR, thus identifying Pooka features.

Feature Property Analysis: Then, MobileMail experts associate to each feature their required property values, *i.e.*, the storage space for the compiled classes and their related libraries. They also identify the constraints imposed by the hand-held devices, say a maximum storage capacity of 3 MB.

Selection of Feature Combinations: MobileMail now determines a set of features satisfying its customers as much as possible within the constraints imposed by the hand-held devices. It starts with the compulsory features and completes them with (near) optimal sets of optional features by performing a multi-objective optimization. If more than one combination of optional features is found, MobileMail uses other criteria to select one combination: for example, one customer could be more valued than others and her needs are only met by one of the combinations. The result is a set of features (and their related classes and libraries) composing hand-held Pooka.

2. MoMS

Figure 1 shows a high-level diagram of the MoMS miniaturization process. It consists of five steps, which we now describe providing their actors, their inputs/outputs, their activities, and an example.

2.1 Pre-requirement Elicitation

Actors: Customers, project manager.

Output: Customers' PRs, customers' values.

Process: A project manager in charge of providing a minia-

Which functionalities would you like to have in your email client?

- 1
- 2
- 3
- 4
- 5

Figure 2: Excerpt of the results of a survey used during PR elicitation

MAIN CLUSTER # 952	AVERAGE SIMILARITY	Feature: It should have spam filter option
216	filter all unwanted emails market as spam	Feature Traces with dependency: net.suberic.pooka.filter.SpamSearchTerm javax.mail.Message net.suberic.pooka.filter.SpamFilter java.lang.Object javax.mail.search.SearchTerm javax.mail.Part java.io.Serializable
102	spam filter to avoid unwanted emails	
149	The system shall support email filtering	
539	The svstem shall support SPAM filtering	
•	In total, 35 customers' pre-requirements	
•	are in this cluster.	
567	spam filter	

Figure 3: Excerpts of the screen shots of a consolidated PR for Pooka (left) and of a feature and its corresponding Java classes (right)

turized version of a program collects a set of PRs for the miniaturized program from (potential) customers, *e.g.*, using a survey. She also assigns an importance value to each customer, *e.g.*, based on her company’s strategic plan.

Example: An example of collected PR for the Pooka email client is: “it should have spam filter option” and an example of a survey is shown in Figure 2.

2.2 Pre-requirement Consolidation

Actors: Program experts.

Input: Customers’ PRs.

Output: Groups of labeled PRs.

Process: Program experts use the following sub-process to: merge similar PRs, distinguish functional from non-functional PRs, and identify compulsory PRs.

1. Experts merge similar PRs.
2. Experts add a unique label to each grouped PR.
3. Experts manually tag grouped PRs as functional or non-functional.
4. Experts distinguish compulsory PRs, which must be ported, from optional ones, which *may* be ported.

For the sake of clarity and without loss of generality, we only consider functional requirements in this work.

Example: Figure 3 shows a subset of the (optional) PRs collected and consolidated for hand-held Pooka concerning the presence of an anti-spam filter and the number of customers requesting it. Program experts manually tagged Pre-requirements such as PR (1) in Figure 2 (“it shall allow me to send emails”) as compulsory for an email client like Pooka.

2.3 Feature Identification

Actors: Program experts.

Input: Pre-requirements and program source code.

Output: Features traced to the source code.

Process: First, program experts trace PRs to implementation units in the source code, *i.e.*, classes, methods, or

functions. Without loss of generality with respect to other programming languages/paradigms, we consider only Java classes in this work and the following sub-process:

1. Experts trace PRs to classes. Each traced PR corresponds to a software feature.
2. Experts manually validate the features and their traces.

Second, experts associate each feature to the complete set of classes (including other classes and libraries) implementing this feature using dependency analysis.

Example: Figure 3 shows a feature and its related classes.

2.4 Feature Property Analysis

Actors: Program experts.

Input: Features, source code and executable of the program, and any other source of relevant information.

Output: Properties required by the features of the program, in terms of storage space, memory size, etc.

Process: Experts must assess the impact of each feature (and its implementation) on the constraints imposed by the hand-held devices. These constraints can include the device storage property, its memory size, its screen size/resolution (some features may just require too much of those), its processor speed (some features may perform intensive computations), the characteristics of the Web browser used as client for the program (which may not support plug-ins), and so on. Experts can determine the values of some feature properties, *e.g.*, the storage property, by analyzing the size of the corresponding Java class files. For other properties, such as memory occupation, experts may have to resort to a manual process, *e.g.*, analyzing both source code and documentation as well as profiling the program.

Example: Program experts measure feature’s disk occupation as the sizes of the corresponding class files (and libraries, possibly). Pooka compulsory features of sending/receiving emails require 2, 130, 533 bytes.

2.5 Selection of Feature Combinations

Actors: Program experts.

Input: Properties of the features, constraints of the devices, customers’ values, source code of the program.

Output: A miniaturized program.

Process: Experts determine that the program satisfies a set of L customers, $C \equiv \{c_1, c_2, \dots, c_L\}$. The program must implement a set of *compulsory* features $ComF$, identified by the experts from among the customers’ PRs. In addition, each customer requires a set $F_i \equiv \{f_{i,1}, \dots, f_{i,N_i}\}$ of N_i *optional* features that the hand-held version of the program must implement. OF is the set of all F_i :

$$OF \equiv \bigcup F_i.$$

A possible miniaturized program can implement F' optional features, where $F' \subseteq OF$. In theory, there exist $2^{|OF|}$ possible sets F' . However, only some of these sets of features meet the constraints imposed by the hand-held devices on their properties (*e.g.*, the size of the Java class files) with an acceptable level of customer satisfaction.

The project manager ranks customers according to their value, val_i , to her project and her company, where $1 \leq val_i \leq V_{max}$ and V_{max} is a maximum value:

$$Val \equiv \{val_1, val_2, \dots, val_L\}.$$

Then, she defines a Customer Satisfaction Rate (*CSR*), to measure her customers' satisfaction level, as:

$$CSR(F') = \frac{\sum_{i=1}^L \frac{|F_i \cap F'|}{|F_i|} \times \frac{val_i}{V_{max}}}{L}$$

that is the average proportion of customers' requested features that the combination F' contains, weighted by the customers' relative value.

The porting requires dealing with a set of property values $P \subset \mathbb{R}^K$ concerning the device usage, such as disk occupation, and with a set of constraints $HC \equiv \{hc_1, \dots, hc_K\}$, each of them imposing a set/interval hc_j of acceptable values on the corresponding property values. The program is composed of M implementation units $IU \equiv \{iu_1, iu_2, \dots, iu_M\}$. Function *Impl* is a continuous function that takes as input a set of features and returns the corresponding implementation units. Function *Prop* returns the set of property values of a program:

$$Prop : IU' \rightarrow P$$

We define a miniaturized program as:

$$IU' = Impl(F' \cup ComF) = \left(\bigcup Impl(f'_{i,j}) \right) \cup Impl(ComF)$$

i.e., the implementation of the selected optional features F' and of the compulsory features $ComF$.

Consequently, the project manager can obtain all (near) optimal combinations of the features of the program by resolving the problem:

$$\begin{aligned} & \min_{F' \in \mathcal{O}F} (-CSR(F'), Prop(Impl(F' \cup ComF))) \\ & \text{such that } \forall p_i \mid Prop(Impl(F' \cup ComF)) = \\ & \quad [p_1, \dots, p_i, \dots, p_K] : p_i \in hc_i. \end{aligned}$$

of which solutions are miniaturized programs, implementing subsets of the features of the original program, so that each:

- satisfies the constraints $HC \equiv \{hc_1, \dots, hc_k\}$;
- maximizes customers' satisfaction $CSR(F')$, *i.e.*, minimizes their dissatisfaction, $-CSR(F')$; and
- minimizes a given set of property values $Prop(Impl(F' \cup ComF))$, *e.g.*, reduces as much as possible the disk occupation of the miniaturized program.

Finally, the project manager chooses one of the solutions to build a compilable version of the miniaturized program by combining the classes and libraries of the features in the selected combination.

Example: Figure 4 shows a Pareto front built for Pooka that solves the previous problem with a disk storage constraint of 3 MB. The origin of the y-axis of the graph is the size of the compulsory features $ComF$. A Pareto front includes non-dominated solutions, *i.e.*, two solutions cannot be found such that one solution is better than the other for both objectives represented on the x- and y-axis, customers' satisfaction and program size in our case. Solution *B* describes a miniaturized version of Pooka with $CSR = 0.50$, including 19 features, for a program size of 2,975,249 bytes.

3. IMPLEMENTATION

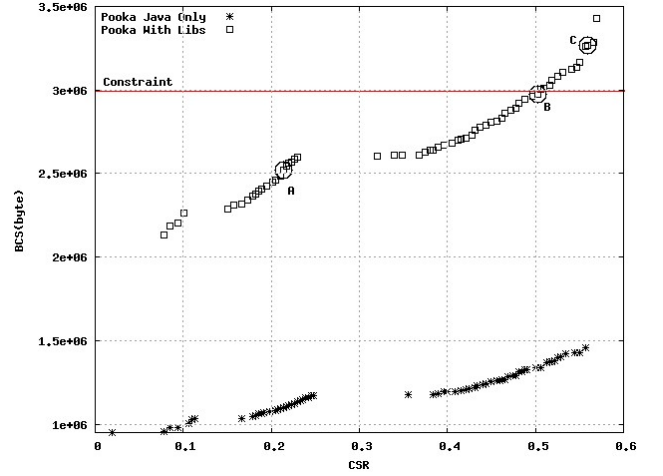


Figure 4: Pareto fronts of Pooka under different assumptions (discussed in Section 5). The two horizontal lines constrain the program size

Figure 5: Excerpt of a survey for PR elicitation

We use state-of-the-art techniques to build a reference implementation of MoMS, without loss of generality with respect to other existing techniques. We support some of the steps with a novel tool, FacTrace³, for PR management.

3.1 Pre-requirement Elicitation

We collect PRs from potential customers using the survey feature of FacTrace. FacTrace lets project managers create surveys for their customers using an embedded implementation of LimeSurvey. Figure 5 shows an excerpt of the FacTrace survey used to collect PRs for an email client.

3.2 Pre-requirement Consolidation

We used Prereqir [19] to perform PR consolidation. Prereqir begins by representing PRs as vectors of weighted words, using a common sub-process that includes tokenizing, stop word removing, stemming, weighting of the obtained words through *tf-idf* indexing [16], and applying an agglomerative nesting clustering algorithm (Agnes [23]). We manually distinguish functional/non-functional and compulsory/optional PRs:

To perform clustering, FacTrace represents PRs in a similarity matrix sm , in which cell $sm_{i,j}$ indicates the cosine

³<http://www.ptidej.net/research/factrace>

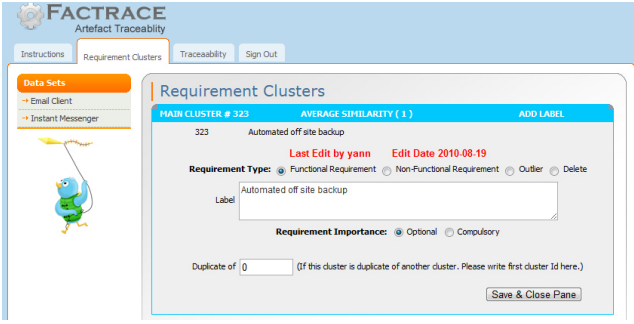


Figure 6: Excerpt of the screen shot of a PR consolidation tool

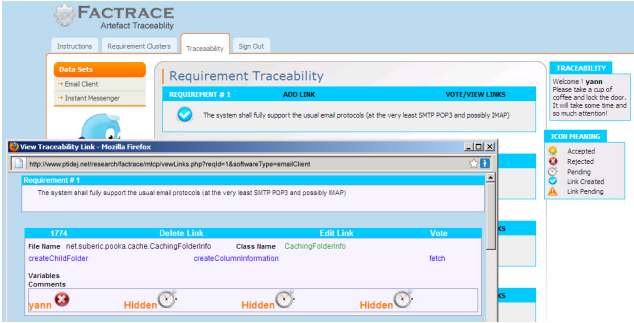


Figure 7: Excerpt of the screen shot of a PR traceability validation tool

similarity between PR i and PR j . Then, it clusters PRs using an agglomerative nesting clustering algorithm, Agnes, which can be called directly from FacTrace to produce clustered PRs. Agnes takes the sm matrix as input and generates clusters of similar PRs. It starts by creating singleton clusters and then keeps merging the closest clusters. The stopping criterion described in Prereqir [19] is used to establish a threshold, which indicates the cut-point of the dendrogram built by Agnes. Finally, once PR are clustered, experts assign labels that represent whole clusters or select any PR that represents whole cluster. Figure 6 shows the user-interface provided by FacTrace to experts to label clustered PRs, fix erroneous clusters, and tag clusters.

3.3 Feature Identification

We reuse an approach based on information retrieval to trace PRs to the source code [1]. The approach has been implemented in the FacTrace Tool. First, using an identifier splitter [31], we collect the identifiers from the source code. These identifiers are split and abbreviations are expanded automatically.

Second, each clustered PR is compared against the term vectors representing each class in the source code using VSM cosine similarity. We again use FacTrace to assess each traced cluster. Figure 7 shows an excerpt of the user-interface provided by FacTrace to validate the traceability links between a PR and some classes.

We perform the dependency analysis using our previous tool, AURA [32], to ensure that each feature is traced to all classes and libraries implementing it. AURA traces dependencies at the method-level (and hence at the class-level).

The traced PRs are features required by some customers to be ported to the hand-held devices.

3.4 Feature Property Analysis

For the sake of illustration, we only consider one property for each feature: its disk space occupation. (We only count the sizes of classes used by multiple features in a miniaturized program once.)

3.5 Selection of Feature Combinations

We support the last step of MoMS by representing solutions of our multi-objective optimization problem [29] as bit-vectors $\vec{x}_i = \{x_{i,1}, \dots, x_{i,|OF|}\} \in \{0,1\}$, where $x_{i,j}$ indicates if feature $f_j \in OF$ is included in the combination of features represented by the solution \vec{x}_i : $x_{i,j} = 1$ if it is included, 0 otherwise.

Let \vec{x} be a solution to our problem, *i.e.*, a feature combination, and Sel a function converting a bit-vector \vec{x} into the corresponding set of features F' , then we define CDR (Customer Dissatisfaction Ratio) and BCS (Byte Code Size) as:

$$CDR(\vec{x}) = -CSR(Sel(\vec{x}))$$

$$BCS(\vec{x}) = \text{Bytecode Size of } Impl(Sel(\vec{x}) \cup ComF)$$

i.e., $BCS(\vec{x})$ is the additional bytecode (with respect to compulsory features) required by the features in \vec{x} .

The problem objective is to find a set X of solutions \vec{x} , whose elements are Pareto-optimal, *i.e.*, $\forall \vec{y} \in X, \vec{y} \neq \vec{x}$:

$$\begin{aligned} & (CDR(\vec{x}) < CDR(\vec{y}) \wedge BCS(\vec{x}) \leq BCS(\vec{y})) \\ \vee & (CDR(\vec{x}) \leq CDR(\vec{y}) \wedge BCS(\vec{x}) < BCS(\vec{y})). \end{aligned}$$

All Pareto-optimal \vec{x} constitute the Pareto front. We use the Non-dominated Sorting Generic Algorithm II (NSGA-II) [10] and its JMetal⁴ implementation, in which we use X as a set of chromosomes. NSGA-II evolves the initial population of randomly-generated solutions through a polynomial mutation operator [11] and a simulated binary crossover operator [9]. A binary tournament selection operator selects the solution candidate for reproduction. The two fitness functions (to be minimized) are $CDR(\vec{x})$ and $BCS(\vec{x})$.

4. CASE STUDIES

We now introduce two case studies whose *goal* is to investigate the usefulness of MoMS in miniaturizing programs. The *quality focus* is the balance between customers' satisfaction and the disk occupation of the miniaturized programs as well as the efficiency of MoMS wrt. a manual miniaturization process. The *perspective* is that of a project manager who wants to miniaturize two programs for some hand-held devices and of researchers who want to understand the (dis)advantages of combining state-of-the-art techniques to address the problem of software miniaturization.

The *context* concerns the miniaturization of two open source programs, the Pooka email client and the SIP Communicator instant messenger. The two programs belong to two different domains and gathering PRs for such programs is possible by surveying potential customers possessing little technical background. This context does not reduce the applicability of MoMS to other kinds of programs.

⁴<http://jmetal.sourceforge.net/>

Table 1: Statistics describing Pooka and SIP

		Pooka	SIP
Version		2.0	1.0
Number of Classes		298	1,771
Number of Methods		20,868	31,502
Source Code Sizes		244,870 LOCs 5.39 MB	486,966 LOCs 27.3 MB
Binary Sizes	Java Only	2.91 MB	11.1 MB
	With Libs	4.1 MB	14.3 MB
Imposed Size Constraint		3 MB	6 MB

Pooka⁵ is an email client written in Java using the Java-Mail API. It supports email through the IMAP and POP3 protocols. Outgoing emails are sent using SMTP. It supports folder search, filters, context-sensitive colors, etc. It also supports different user interfaces, in particular Eudora and Outlook-like interfaces. It partially implements an address book.

SIP⁶ is an audio/video Internet phone and instant messenger that supports some of the most popular instant messaging and telephony protocols, such as SIP, Jabber, AIM/ICQ, MSN, Yahoo! Messenger, Bonjour, IRC, RSS. It is based on the OSGi⁷ architecture and its Apache Felix⁸ implementation. SIP is LGPL⁹.

We chose them because we could readily ask our colleagues and students to act as customers of such programs, when compared to, *e.g.*, routers, which are mostly perceived as “black boxes” even by most computer scientists. Table 1 provides some general descriptive statistics for the two programs and the disk occupation constraints that we suppose the programs must meet: 3 MB and 6 MB for Pooka and SIP. We used Pooka to motivate our work in Section 1.2 and to illustrate the MoMS process and its implementation in Sections 2 and 3. For the sake of completeness and of the discussions of the differences between applying MoMS on Pooka and SIP, we provide further results on Pooka.

The *research questions* are:

- **RQ1:** Does the MoMS process and its reference implementation allow one to obtain compilable miniaturized programs balancing customers’ satisfaction and constraints imposed by some hand-held devices?
- **RQ2:** How much time does the MoMS process save a project manager and her program experts during program miniaturization when compared to performing the miniaturization process entirely manually?

We address **RQ1** by measuring and optimizing two different dependent variables: *customer satisfaction* (see Section 2.5) and the *sizes of the ported programs*. We study how MoMS allows one to select a subset of features that, on the one hand, satisfies customers and, on the other hand, meets device constraints with reduced disk occupation.

We address **RQ2** by comparing the *time* needed to perform the porting activities when applying our reference implementation of MoMS with the time needed to perform the

⁵<http://www.suberic.net/pooka/>

⁶<http://sip-communicator.org/>

⁷<http://www.osgi.org>

⁸<http://felix.apache.org>

⁹<http://www.gnu.org/licenses/lgpl.html>

activities manually. We report the average manual and automatic completion times for each step of the process. All but the fourth and sixth authors acted as experts and performed each step together manually, using only simple tools usually available to project managers and developers, such as Microsoft Windows Search and Excel.

All data is available on-line at <http://www.ptidej.net/downloads/experiments/icse11/>.

4.1 Pre-requirement Elicitation

We conducted an online survey to gather PRs for an email client and an instant messenger using FacTrace. We sent 350 invitations to 250 computer science professors/researchers and 100 students. Of the 350 recipients, 151 responded, of which 73 completed the entire survey.

Among the 73 respondents, there were 28 females and 45 males. Statistics analysis of participants and their background revealed some interesting observations: 45.21% were students, 28.7% researchers, and 26.02% industry-related people; 22.6% had no experience whereas 75.34% had one year or more of experience; 70.08% and 80.82% do not actively contribute to the development of an email client and/or an instant messenger, respectively; 71.23% and 72.60% have experience with Java and C programming, respectively. The majority of the respondents, 72.60%, use Microsoft Windows. Surprisingly, respectively 22 (9.59%) and 7 (6.85%) respondents do not use either an email client or an instant messenger. We excluded these respondents from our analysis due to their lack of expertise. The remaining 59 and 66 respondents spent an average of 10 and 9 minutes to write PRs for the email client and instant messenger, respectively. Average completion time of the survey is 20 minutes. The respondents wrote 599 and 639 PRs for the email client and instant messenger, respectively.

In our case studies, to minimize the impact of customer value on subsequent steps, we chose to randomly divide the customers into 7 groups and randomly assigned them a value on a 7-point Likert scale.

Manual Time: Project managers have access to Web surveys to collect PRs. Our survey took ~ 17 and ~ 20 hours to set up and design for an email client and an instant messenger.

Automatic Time: With MoMS, we did not save time during this step with respect to a manual approach because we essentially use a similar survey.

4.2 Pre-requirement Consolidation

Approaches for PR consolidation usually rely on clustering techniques [17, 19]. We clustered the 599 email client and 639 instant messenger PRs using FacTrace, which calls Agnes and displays the obtained clusters for their analysis and the identification of a cut-off threshold. The best thresholds were 41% and 46% in our case studies, below which Agnes mixed different PRs. We thus recovered 221 and 235 clusters for an email client and an instance messenger.

We used FacTrace to manually tag PRs as functional/non-functional and compulsory/optional and to label each cluster. There were 93 functional, 25 non-functional, and 6 spurious PRs for the email client; 82 functional, 20 non-functional, and 9 spurious PRs for the instant messenger.

Manual Time: A manual clustering and tagging/labeling of the PRs took us all ~ 9 hours each for the email client and the instance messenger.

Automatic Time: It took ~ 6 and ~ 8 hours to generate,

analyze, and manually validate clusters for the email client and the instance messenger.

4.3 Feature Identification

To evaluate the performance of the feature identification implemented in MoMS in terms of precision and recall [16] as well as to gain insight about the time required for this step, we manually created an oracle of features for Pooka (our email client) and SIP (our instant messenger): we split the 93 and 82 functional PRs for Pooka and SIP into three batches and three authors built and voted on each other’s traced features. We created 318 feature traces leading to 41 features and 830 feature traces leading to 51 features for Pooka and SIP.

In parallel, we applied our automated approach to feature identification and recovered 128 traces (30 features) and 363 traces (36 features) for Pooka and SIP. Using the feature validation interface of FacTrace, we manually discarded false traces and created missed traces.

We then compared the automated and the manually recovered traces: the automated approach recovered 40% and 44% correct feature traces for Pooka and SIP, *i.e.*, 128 and 363 traces, with a precision of 7%. We chose a cut-off threshold of 39% and 42% for Pooka and SIP to balance precision and recall and obtained 28 and 34 features. We favored recall over precision because we preferred to associate more classes to each feature and thus have a compilable implementation, rather than associate less classes with a feature and miss some important classes. Missing classes are not found by the dependency analysis, because if a class is missed, then no dependency may lead to it.

We performed dependency analysis using AURA [32], which yields the minimum, average, and maximum number of classes per feature of 7, 143, and 405 for Pooka and 12, 344, and 863 for SIP. These values show that some features require more classes than others, but never the entire program: the classes of the two programs are not fully coupled and thus various combinations of features would lead to different miniaturized versions of the programs.

Manual Time: We took ~ 135 and ~ 171 hours to recover the 318 and 830 feature traces for Pooka and SIP.

Automatic Time: It took us ~ 21 and ~ 30 hours to generate and to manually validate the feature traces and corresponding features for Pooka and SIP.

4.4 Feature Property Analysis

This step is straightforward in the context of our case studies. We sum the sizes of the class files corresponding to the classes participating in the implementation of each feature to measure the storage required by the features. We use this measure because it is simple yet does not lessen the generality of our process. The minimum, average, and maximum size of class files per feature are 2,449 bytes, 765,039 bytes, and 1,791,848 bytes, for Pooka, and 30,466 bytes, 1,204,657 bytes, and 2,840,149 bytes for SIP. We chose the features “it shall allow me to send emails” and “it shall allow me to receive emails” for Pooka and corresponding PRs for SIP (related to sending and receiving instant messages) as compulsory features, whose sizes are respectively 2, 130, 533 and 3, 265, 635 bytes.

Manual Time: The time required by this step depends on the chosen properties: disk storage only requires summing the sizes of class files, *i.e.*, a negligible amount of time.

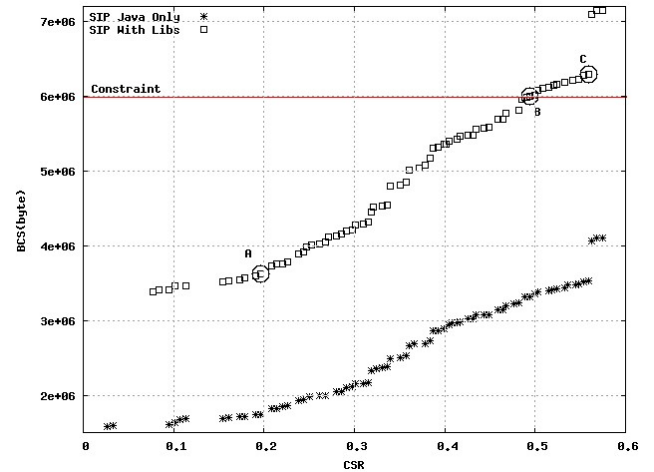


Figure 8: Pareto Fronts of SIP

Automatic Time: Summing class file sizes can be done easily with existing tools, for example TreeSize¹⁰, at no cost.

4.5 Selection of Feature Combinations

We applied NSGA-II to the optional features and their related classes and computed various Pareto fronts with mutation probability of 4%, crossover probability of 90%, population size of 100, and evaluation number of 25,000 (the default values of JMetal NSGA-II). We ensured that larger iteration numbers did not yield better solutions.

Project managers and program experts benefit from these Pareto fronts that assist them to immediately select combinations of features that satisfy customers’ PRs and the constraints of the hand-held devices. The combinations on the Pareto front include, on average, 14 features for Pooka and 17 for SIP. Without considering the constraints on BCS, the sizes of the miniaturized Pooka and SIP with external libraries vary from 2,131,397 to 3,430,686 bytes and from 3,387,158 to 7,149,752 bytes, respectively, *i.e.*, when satisfying the minimum and maximum number of customers.

Figure 4 shows the Pareto front for Pooka, when taking into account libraries. Its y-axis origin is the BCS of the compulsory features in Pooka. It comprises two increasing regions connected by one relatively flat region. Combinations belonging to this flat region can satisfy more customers with a small increase of BCS than those in the steep regions: 50% increase of CSR (from 0.23 to 0.32) only costs 0.4% increase of BCS (from 2,595,844 to 2,607,052 bytes).

As shown in the upper part of Table 2, Solution *B* is the closest to the intersecting point of the Pareto front and the constraint on BCS. It is the most interesting solution for a project manager, because it provides the highest CSR with an acceptable BCS. Solution *A* and the others below *B* are also acceptable because their BCS is smaller than the constraint, although these solutions satisfy less customers. Solution *C* is ruled out, because its BCS violates the given size constraint.

Figure 8 shows the Pareto front of SIP (including libraries). Its y-axis origin is the BCS of the compulsory features in SIP. This front does not have an evident flat part when compared to Pooka. Thus, a project manager can choose Solution *B*,

¹⁰http://www.jam-software.com/treesize_free/

Table 2: Example solutions for Pooka and SIP. These sets of features would complete the set of compulsory features in the miniaturized programs

Programs	Solutions	Characteristics			
Pooka	Solution A (Acceptable)	CSR	Customers	BCS (in bytes)	Features
		0.21	43/59	2,518,099 < 3,000,000	15/30
		It shall have wide range of formatting options for composing emails			
		It shall allow choosing the protocol POP or IMAP and specify their settings			
	Solution B (Best)	The system shall allow email to be sorted by sender date received size or thread			
		CSR	Customers	BCS (in bytes)	Features
		0.50	51/59	2,975,249 < 3,000,000	19/30
		Print emails with printer			
	Solution C (Unacceptable)	Auto save of uncompleted messages			
		It should have spam filter option			
		CSR	Customers	BCS (in bytes)	Features
		0.56	51/59	3,269,939 > 3,000,000	23/30
SIP	Solution A (Acceptable)	Create directories to store emails			
		It should be possible to define email filters			
		Auto completion of email address from global address list (or ldap server) and contact list			
		CSR	Customers	BCS (in bytes)	Features
	Solution B (Best)	0.20	28/66	3,617,152 < 6,000,000	10/36
		It shall support communication over secure protocol (ssh)			
		Inclusion of fonts (change color, text style and size) and smilies for text chatting			
		SMS facility on mobile or phones from chatting			
	Solution C (Unacceptable)	CSR	Customers	BCS (in bytes)	Features
		0.49	43/66	5,998,796 < 6,000,000	23/36
		The system shall allow the display of a contact public information			
		The system shall allow to receive instant message from other connected and approved users			
Solution C (Unacceptable)	Create and manage discussions chat rooms				
	CSR	Customers	BCS (in bytes)	Features	
	0.56	48/66	6,285,599 > 6,000,000	31/36	
	Integration with a OS to open URL				
Solution C (Unacceptable)	Send chat invitation to a contact				
	The system shall allow to receive instant message from other connected and approved users				

which is closest to the constraint and achieves the best CSR. The bottom part of Table 2 describes three possible solutions for SIP. The project manager can use any solution to build a compilable version of the miniaturized program by putting together the corresponding classes.

Manual Time: To obtain such combinations of features manually, it took us ~ 42 and ~ 53 hours for Pooka and SIP.

Automatic Time: Execution time is less than 15 minutes on a laptop with an Intel Duo 1.5 GHz processor and 4 GB of memory, running Microsoft Windows XP.

4.6 Answers to the Research Questions

To conclude the case studies, we can answer the two research questions as follows:

- **RQ1:** We answer this question positively. We showed that MoMS results in combinations of features that balance customers' satisfaction and device resource occupation, satisfying constraints imposed by the device itself. Depending on her needs, the manager can choose combinations favoring customer satisfaction or combinations favoring device resource usage. These combinations are by construction compilable because they include, for each feature, the set of all classes implementing the feature.

- **RQ2:** We also answer this question positively.

It took $17 + 9 + 135 + 0 + 41 = 202$ hours and $20 + 9 + 171 + 0 + 53 = 253$ hours to manually perform the miniaturization of Pooka and SIP, respectively.

With MoMS and its reference implementation, it took $17 + 6 + 21 + 0 + 0.25 = 44.25$ hours and $20 + 8 + 30 +$

$0 + 0.25 = 58.25$ hours to perform the miniaturization of Pooka and SIP.

Overall, we saved 177 hours of work, *i.e.*, 77%, using MoMS. Moreover, MoMS describes systematically for the first time the steps for miniaturizing programs, thus further saving time for the project managers who would have, without it, had to progress haphazardly and build their own process.

4.7 Threats to Validity

Threats to *construct validity* can be due to imprecision in the measurements performed in the study. The degree of imprecision of the automatic feature location approach was quantified by means of a manual evaluation of the precision and recall of the approach. For tasks such as PR consolidation, the clusters were manually labeled and assessed by three authors. Also, we made sure that each combination produced by the optimization correctly compiles.

There is a single group threat to *internal validity*. We minimized this threat by examining the amount of effort saved for a project manager and program experts at various steps of the process, *e.g.*, during feature identification. Fatigue effect could affect the measurement of time needed to perform the tasks manually; we limited such a threat by performing the different MoMS tasks on different days but we cannot guarantee that repeating the process multiple times would give the same results.

Threats to *external validity* concern the generalization of our answers. We cannot claim that MoMS would be effective in the same way on all programs and properties. We applied MoMS to two different programs belonging to different domains. Programs having different characteristics, *e.g.*, commercial programs, programs from different domains, and

different languages, could lead to different results. Similarly, different results could be achieved using requirements validated by project managers instead of PRs, although PRs are more realistic when one wants to perform a porting based on customers’ requests. The only device property considered in our case studies was disk occupation. It would be desirable to perform further studies with different sets of properties. Besides other threats concerning RQ2, the effort required for miniaturization is also affected by threats to external validity as developers/experts with different skills, knowledge of the programs, and of MoMS would perform differently.

5. DISCUSSION

We now discuss the impact of various factors on the miniaturization process and outline issues not addressed by MoMS when porting programs.

5.1 Factors Impacting MoMS

Desktop vs. Hand-held Pre-requirements: We also collected PRs concerning the desktop version of the program to port to hand-held devices and not PRs dedicated to the hand-held devices. This was done because customers may not be aware of the constraints imposed by hand-held devices nor of the properties of such devices. Therefore, we suggest that one resort to more general PRs. However, we could also collect PRs specific to some hand-held devices, but these may lead to less features if they are too specialized wrt. the desktop program.

Similarity among Pre-requirements and Source-code Artifacts: Manual validation may be infeasible for large programs with hundreds of PRs and thousands of classes. Consequently, we favor high recall to obtain a hand-held version of a program that may be bigger than required but that is compilable. Favoring precision over recall would lead to hand-held programs with just the minimum number of classes – but some classes could be missing. Dependency analysis could be used to recover these missing classes. But, if a class central to a feature is missed, no dependency analysis may be able to retrieve it.

In our reference implementation of MoMS, we use natural-language processing techniques to consolidate PRs and to trace PRs to source code. We thus assume that there is some textual similarity between the words in the PRs and the identifiers in the source code. This assumption is at the heart of most techniques in PR engineering, so it is not directly a threat to our process. Moreover, in the three first steps, we include sub-steps of manual validation by the project manager and program experts to ascertain that each feature is traced to the corresponding implementation units with perfect recall, even at the expense of precision.

Code Coupling: Class coupling [8] impacts the results of MoMS. A high coupling means that the sizes of the feature combinations will increase quickly as new features are added to satisfy more customers, because each new feature will bring many new classes. After a certain CSR value, the increase in size will slow down dramatically because most of the classes are already present in the combinations. On the contrary, a low coupling means that the sizes of the feature combinations will increase slowly but finally rise quickly as more features are included to satisfy more customers.

Figures 4 and 8 show that coupling impacts the results of the optimization sub-process. While the Pooka Pareto fronts have a flat region, those of SIP do not. This can

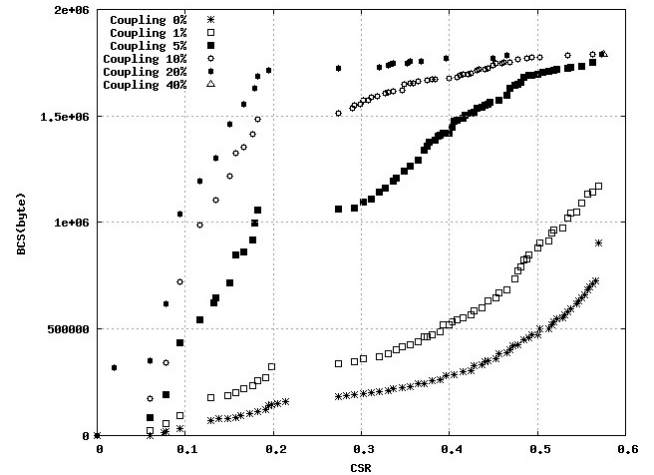


Figure 9: Pareto fronts of synthetic Pooka

be explained by the different coupling that Pooka and SIP exhibit. Pooka has a $CBO = 324$ (1.08 per classes), SIP has a $CBO = 2,186$ (1.23 per classes). Pooka and SIP cohesions are similar with $LCOM5 = 0.612$ and $LCOM5 = 0.626$. The coupling of Pooka is smaller than that of SIP, thus its features are more independent from one another than those in SIP, which explains the flat portion of the Pooka Pareto fronts, composed of different combinations of relatively-independent features.

We evaluate the impact of coupling on the miniaturization process by introducing artificial (synthetic) coupling among classes in Pooka and SIP, and evaluating the influence of different levels of coupling on the sizes of the resulting hand-held versions. This is done by removing the existing dependencies and adding an artificial dependency between each class and a set of randomly selected classes. We vary the coupling percentage to study its influence. We do not distinguish compulsory and optional feature sets so that only one parameter varies: the overall program coupling.

Figure 9 shows the Pareto fronts obtained for Pooka when changing its coupling between classes from 0%, 1%, 5%, 10%, 20%, to 40%. As expected, the acuteness of the slopes of the Pareto fronts change. The front becomes a single point for a 40% coupling, which indicates that no matter what combinations of feature the project manager chooses, they require all the classes of the program. The results for the SIP synthetic data are similar to those obtained for Pooka, although the Pareto front becomes a single point when the coupling between classes reaches 20%, as shown in Figure 10, confirming the impact of coupling on the results of MoMS.

Third-party Libraries: We computed two Pareto fronts per program, one including third-party libraries, and one excluding them. When including external libraries, we assumed that the miniaturized programs are statically linked to all the necessary libraries and ask for nothing but a bare operating system and a Java Virtual Machine (JVM). When excluding the libraries, we assume that the hand-held devices allow the sharing of libraries among Java programs.

Figure 4 shows that the Pareto front of Pooka excluding external libraries (labeled “Java Only”) has a similar shape as that with external libraries. The major difference between them is that the BCS of the former increases slower than that

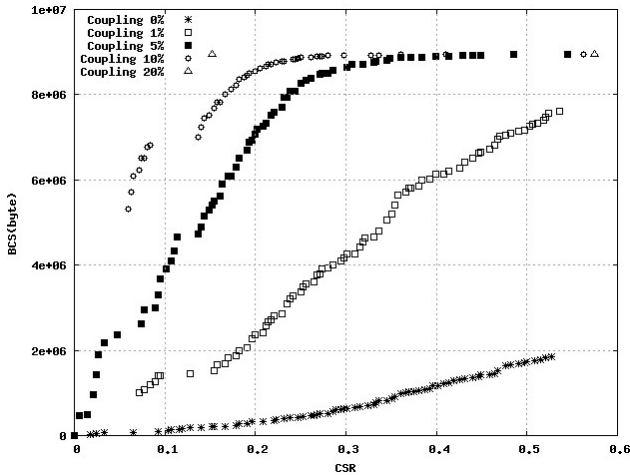


Figure 10: Pareto fronts of synthetic SIP

of the latter, because the former involves less bytecode per feature. Similarly, Figure 8 shows that the Pareto front of SIP excluding external libraries has no obvious differences from the one with external libraries, but again with smaller BCS. The jump around $CSR = 0.55$ on the right side indicates the presence of features with relatively large BCS, but not requiring external libraries. We conclude that external libraries do not essentially impact the results of MoMS.

5.2 Issues not Addressed by MoMS

MoMS aims at selecting features of a program to be ported to a hand-held device. Yet, other issues must be considered even though they are out of the scope of this work.

Dead Code and Code Clones: As highlighted in previous work [12], miniaturizing a program might require removal of dead code, clones, and unnecessary libraries. The approach proposed in this paper intrinsically removes dead code, as it only considers for the porting the code required to execute the selected features. Although, we performed the miniaturization at the class level, we could further reduce the space required by the program by removing unnecessary methods (if any). Clone refactoring is also possible [4].

Unavailable/Different APIs: We assumed that the hand-held devices targeted by the miniaturization process include a full JVM and Java libraries. However, hand-held devices could only support a limited JVM. A proper adaptation of the ported program is needed, following a process similar to migrating to a different language [34] or alternative APIs offered by the available class libraries [25].

GUI Issues: Porting a desktop GUI to a hand-held device screen, possibly dealing with touch-screens, limited browsers, etc. is a process different than, but complementary to, the one proposed by MoMS. Indeed, the ported/redesigned GUI must only account for the ported features. Approaches for migrating GUIs to hand-held devices exist [5].

Non-functional Requirements: Finally, program miniaturization requires dealing with non-functional PRs, *e.g.*, performance issues, because some features may perform well on a desktop computer with a powerful processor but poorly on a hand-held device with a slower processor. Non-functional PRs require program profiling and performance analysis, out of scope of MoMS, although performance properties can also

be considered in MoMS as any other properties.

Compilation and Execution: The resulting miniaturized program obtained through our reference implementation of MoMS are by construction compilable, because we conservatively favor a perfect recall and, thus, the possible inclusion of more classes than necessary.

However, the compiled version of the hand-held program may not run for several reasons, including the lack of a complete Java class libraries on the devices; missing resources, such as images, XML files, and so on not handled by the miniaturization process; missing the user-interface or “main” calls to activate the included features.

As with any other transformation techniques, testing of the resulting program is essential to ensure that the program is executable.

6. RELATED WORK

The problem of software miniaturization was introduced by Di Penta *et al.* [12], who proposed a process to reduce the footprint of a program during its porting to some hand-held devices. The process deals with different issues (such as removing dead objects, refactoring clones, removing circular dependencies, and modularizing) using an optimization based on clustering and genetic algorithms, guided by a fitness function of the program footprint. Previously, Di Penta and Antoniol [3] also proposed an algorithm that combines the usage of static and dynamic data to renovate libraries, so that objects that are used together frequently are clustered into the same library. They applied their algorithm to several medium and large-size open-source programs, GRASS, MySQL, QT, and Samba, effectively producing smaller, loosely-coupled libraries and reducing memory requirement for the programs. We also used optimization techniques to support the porting of a program to some hand-held devices. However, we extend the previous work by proposing a process that covers a wide range of previous work dealing with PR elicitation, PR consolidation, and software miniaturization and that can maximize customers’ satisfaction under constraints imposed by the hand-held devices.

6.1 Pre-requirement Elicitation

Requirement elicitation and consolidation has been addressed by previous research. In particular, Goldin and Berry [17] used signal processing techniques to abstract common requirements, while Hayes *et al.* [19] proposed the Prereqir approach, which uses clustering techniques to consolidate and trace customers’ requirements to source code. Our work is inspired by Prereqir for requirement consolidation and feature location.

6.2 Feature Identification

The identification of the implementation units in a source code that implement some features is a well-known problem, addressed as early as 1995 by Wilde *et al.* [33], who use two sets of test cases to build and compare two execution traces, one where a feature is exercised and another where the feature is not, to identify the source code associated with the feature in the program.

Since 1995, many works have been proposed to trace feature to code. Chen and Rajlich [6] developed an approach to identify features using Abstract System Dependencies Graphs. Eisenbarth *et al.* [13, 14] combined previous ap-

proaches by using both static and dynamic data to identify features. Greevy *et al.* [26] studied the evolution of object-oriented program entities from the point of view of their features. Antoniol and Guéhéneuc [2] proposed an epidemiological metaphor that combines both static and dynamic data to identify features. Poshyvanyk *et la.* [30] combined this previous approach with a LSI-based approach to reduce the efforts in identifying the features while further improving precision and recall.

We use an IR-based traceability recovery approach, largely inspired from previous work by Lucia *et al.* [24]. We also use our previous work, AURA [32], to gather automatically the dependencies of any implementation units.

6.3 Feature Property Analysis

We are not aware on any work to assign systematically properties to implementation units. However, related work includes well-known performances analyses, such as those performed by profilers, *e.g.*, JProfiler, which uses test cases or user inputs to compute an average computation times for all the methods called during the execution of a program.

6.4 Feature Combinations

Combining features or requirements has been largely investigated in the past in the form of requirement prioritization. Previous techniques include the pair-wise comparison of requirements through the Analytic Hierarchy Process (AHP) [27] as well as simple requirement ranking. Karlsson and Ryan [21] developed a cost-value approach for requirement prioritization, based on AHP. Crucial issues in the application of AHP to requirement prioritization are the explosion of possible pair-wise comparisons, $n \cdot (n - 1)$, where n is the number of requirements, and the need for an appropriate process and tool support for stakeholders applying AHP. Karlsson *et al.* [20] also defined a set of heuristics to reduce the number of possible pair-wise comparisons and a process and a tool to apply AHP to requirement prioritization.

Karlsson *et al.* [22] also performed a more extensive evaluation of AHP and an enhanced version of it (hierarchical AHP), comparing it with other prioritization methods, basically ordering/search methods (bubble sort, minimal spanning tree, binary search tree) or partitioning of requirements into priority groups. Despite scalability problems, AHP was found to be the most reliable prioritization method.

Omolade, Saliu, and Ruhe [28] treated the problem of selecting candidate features for release planning as a multi-objective optimization problem, where they tried to pursue a tradeoff between customers' satisfaction and the effort spent in implementing change requests. They used impact analysis to determine the part of the implementation being affected by a change/feature request.

Harman *et al.* [18] presented a study in which they analyzed the use of single and multi-objective optimization in the selection of features to deal with the next release problem, also providing a heat matrix visualization to support the selection. They found that requirements with higher cost on inaccuracy have almost always the highest impact on the prioritization.

Finkelstein *et al.* [15] used multi-objective optimization to find tradeoffs between conflicting requirements coming from different customers.

We share with this previous work the need for feature selection while dealing with conflicting PRs. However, we

also handle constraints imposed by the hand-held devices to which the programs are ported.

7. CONCLUSION AND FUTURE WORK

Society's reliance and dependence on computers is nowhere more obvious than in the ubiquity of hand-held devices and other limited-resource devices. Many people want to use the same programs on their hand-held devices as on their desktop computers.

This paper presented MoMS, a multi-objective miniaturization process, that can be applied to a program to elicit its pre-requirements, identify its features, select the "best combinations" of features to port to hand-held devices, and generate a compilable miniaturized version of the program. We defined "best combinations" as the combinations of compulsory and optional features that satisfy customers as much as possible while minimizing device resource usage and satisfying specific constraints imposed by the device.

We described the MoMS process using a fictitious company, MobileMail, that want to miniaturize an email client Pooka. We also described a reference implementation of MoMS combining techniques from various fields, including requirements engineering, natural language processing, feature identification, and multi-objective optimization.

Two case studies using Pooka, an email client, and SIP, an instant messenger, showed that MoMS can support project managers and program experts in selecting the program features to be ported to some hand-held devices while balancing different objectives by clearly showing the combinations providing better customer satisfaction (CSR) with lower increase of disk size (BCS). The studies allowed us to estimate the time saved using MoMS compared to a manual process: $202 - 44.25 = 157.75$ hours for Pooka and $253 - 58.25 = 194.75$ hours for SIP, *i.e.*, 78% and 77% reduction of effort.

We discussed factors impacting the MoMS process, such as code coupling and third-party libraries. We argued that these factors do not reduce the applicability of the process and quality of its implementation. We also discussed other issues not addressed by MoMS but potentially impacting its results, *e.g.*, dead code.

Future work includes applying MoMS on other programs and with other constraints, *i.e.*, memory footprint and screen resolution. We will also take into account non-functional requirements. We will also further study the impact of various factors on the results of MoMS, *e.g.*, by characterizing programs and features in terms of their relative coupling; we will examine requirement elicitation and consolidation to assess the impact of different techniques; we will evaluate the effort required by MoMS compared to other approaches and tools and, thus, we will be able improve our tool support for MoMS, FacTrace.

Acknowledgments

Hayes is funded in part by the National Science Foundation under NSF grant CCF-0811140. Ali, Antoniol, Guéhéneuc, and Wu are partly supported by the Canada Research Chairs in Software Change and Evolution and in Software Patterns and Patterns of Software.

8. REFERENCES

- [1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and

- documentation. *IEEE Trans. Software Eng.*, 28(10):970–983, 2002.
- [2] G. Antoniol and Y.-G. Guéhéneuc. Feature identification: An epidemiological metaphor. *Transactions on Software Engineering (TSE)*, 32(9):627–641, September 2006. 15 pages.
 - [3] G. Antoniol and M. D. Penta. Library miniaturization using static and dynamic information. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 235, Washington, DC, USA, 2003. IEEE Computer Society.
 - [4] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the Working Conference on Reverse Engineering (WCRE 2000)*, pages 98–107, 2000.
 - [5] G. Canfora, Gerardoand Di Santo and E. Zimeo. Developing Java-AWT thin-client applications for limited devices. *IEEE Internet Computing*, 9(5):55–63, 2005.
 - [6] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In A. von Mayrhauser and H. Gall, editors, *Proceedings of the 8th International Workshop on Program Comprehension*, pages 241–252. IEEE Computer Society Press, June 2000.
 - [7] T. L. Cheung, K. Okamoto, F. Maker, III, X. Liu, and V. Akella. Markov decision process (MDP) framework for optimizing software on mobile phones. In *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, pages 11–20, New York, NY, USA, 2009. ACM.
 - [8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.
 - [9] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. Wiley-Interscience Series in Systems and Optimization. John Wiley & Sons, Chichester, 2001.
 - [10] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II. In *PPSN VI: Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, pages 849–858, London, UK, 2000. Springer-Verlag.
 - [11] K. Deb and M. Goyal. A combined genetic adaptive search (genas) for engineering design. *Computer Science and Informatics*, 26:30–45, 1996.
 - [12] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo. A language-independent software renovation framework. *J. Syst. Softw.*, 77(3):225–240, 2005.
 - [13] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In G. Canfora and A. A. Andrews, editors, *Proceedings of the 8th International Conference on Software Maintenance*, pages 602–611. IEEE Computer Society Press, November 2001.
 - [14] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *Transactions on Software Engineering*, 29(3):210–224, March 2003.
 - [15] A. Finkelstein, M. Harman, S. A. Mansouri, J. Ren, and Y. Zhang. A search based approach to fairness analysis in requirement assignments to aid negotiation, mediation and decision making. *Requir. Eng.*, 14(4):231–245, 2009.
 - [16] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, facsimile edition edition, June 1992.
 - [17] L. Goldin and D. M. Berry. Abstfinder, a prototype natural language text abstraction finder for use in requirements elicitation. *Automated Software Engg.*, 4(4):375–412, 1997.
 - [18] M. Harman, J. Krinke, J. Ren, and S. Yoo. Search based data sensitivity analysis applied to requirement engineering. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, pages 1681–1688, 2009.
 - [19] J. H. Hayes, G. Antoniol, and Y.-G. Guéhéneuc. PREREQIR: Recovering pre-requirements via cluster analysis. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE)*, pages 165–174. IEEE Computer Society Press, October 2008. 10 pages.
 - [20] J. Karlsson, S. Olsson, and K. Ryan. Improving practical support for large-scale requirement prioritising. *Requir. Eng.*, 2(1):51–60, 1997.
 - [21] J. Karlsson and K. Ryan. A cost-value approach for prioritizing requirements. *IEEE Software*, 14(5):67–74, 1997.
 - [22] J. Karlsson, C. Wohlin, and B. Regnell. An evaluation of methods for prioritizing software requirements. *Information & Software Technology*, 39(14-15):939–947, 1998.
 - [23] L. Kaufman and P. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, 1990.
 - [24] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4):13, 2007.
 - [25] M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 205–214. ACM Press, May 2010.
 - [26] S. D. Orla Greevy and T. Girba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(6):425–456, November 2006.
 - [27] T. Saaty. *The Analytic Hierarchy Process*. McGraw-Hill, Inc., 1980.
 - [28] M. O. Saliu and G. Ruhe. Bi-objective release planning for evolving software systems. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 105–114, 2007.
 - [29] Y. Sawaragi, N. Hirotaka, and T. Tetsuzo. *Theory of multiobjective optimization*. Academic Press, Orlando, 1985.
 - [30] Denys Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *Transactions on Software Engineering (TSE)*, 33(6):420–432, June 2007. 14 pages.
 - [31] Nioosha Madani, Latifa Guerrouj, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. Recognizing words from source code identifiers using speech recognition techniques. In R. Ferenc and J. C. Dueñas, editors, *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society Press, March 2010. Best paper. 10 pages.
 - [32] Wei Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. AURA: A hybrid approach to identify framework evolution. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*. ACM Press, May 2010.
 - [33] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. In K. H. Bennett and N. Chapin, editors, *Journal of Software Maintenance: Research and Practice*, pages 49–62. John Wiley & Sons, January-February 1995.
 - [34] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 195–204. ACM Press, May 2010.