2011 27th IEEE International Conference on Software Maintenance (ICSM)

MoMS: Multi-objective Miniaturization of Software

Nasir Ali[1,2], Wei Wu[1,2], Giuliano Antoniol[2],
Massimiliano Di Penta[3], Yann-Gaël Guéhéneuc[1], and Jane Huffman Hayes[4]
[1] Ptidej Team, [2]SOCCER Lab, DGIGL, École Polytechnique de Montréal, Canada
[3] RCOST, University of Sannio, Italy, [4] 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, and 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 with features may be in conflict with a device's limited storage, memory, or battery life. This paper proposes a process, MoMS, for the multi-objective miniaturization of software to help developers miniaturize programs while satisfying multiple conflicting constraints. It can be used to support the reverse engineering, next release problem, and porting of both software and product lines. 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 optimized miniaturization and helps reduce effort by 77%, on average, over a manual approach.**

***Index Terms*—Software miniaturization; Requirement engineering; Feature identification; Multi-objective optimization**

I. 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 limited storage, from two to 64 MB (*e.g.*, 40 MB for the Lynksys WRT54GS v2.0, 24 MB for the Garmin eTrex Vista HCx hand-held GPS).

As many people use desktop computers, either at home or at work, it is not surprising that many programs are ported to hand-held and other limited-resource devices to help customers work from anywhere. 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 contradictory constraints [1]: the storage space, memory size, computing power, screen size of the hand-held devices, etc. For example, Microsoft Office Mobile only provides a limited set of features compared to its desktop version, *e.g.*, it does not handle footnotes, endnotes, headers, footers, page breaks, which might make some existing customers jump to the products of other companies. Adobe announced a version of Photoshop for iPad that "functions just like a real version of Photoshop"[1] at the Photoshop World Conference on March, 30^{th}, 2011, which shows that product lines for both desktop computers and hand-held devices is a lucrative trend.

This paper presents MoMS, a novel process and its implementation for the multi-objective miniaturization of programs. In principle, MoMS can be applied without requiring specific automatic support, although we provide a (semi)-automatic implementation of MoMS. MoMS directs: (1) the elicitation of a set of pre-requirements[2] (PRs) from multiple customers; (2) the consolidation of these PRs; (3) the identification of the implementation units corresponding to each PR (if any) to obtain features [2]; (4) the identification of the device properties required by the features and device constraints (in our implementation we consider the program storage occupation and the number of executed bytecode instructions, which could be used as an estimate of the battery consumption [3]); and (5) the selection of the features to port through a multi-objective optimization and the generation of the miniaturized program.

The problem of selecting the (near) optimal set of features with the objective of satisfying customers and resource constraints is a constrained multi-objective optimization problem [4]. Multi-objective optimization is the process of finding solutions to problems with conflicting objectives. In this paper, different customers may require different features: a company might not be able to satisfy one customer without dissatisfying other customers. Also, satisfying all the customers' PRs may cause an increase of device resource usage by the program, which is constrained by the hand-held devices. A project manager could painstakingly try to identify the "best"

[1] http://www.ismashphone.com/2011/04/adobe-shows-off-photoshop-like-application-for-ipad.html

[2] Pre-requirements, also called "user needs", include system concepts, user expectations, system environment, etc. Examples of PRs for a word processor might be "must run under Linux", "not require conversion of existing files" [2]. It is a pre-requirement because it deals with domain as well as software requirements and is available before a formal elicitation process.

978-1-4577-0664-6/11/$26.00 ©2011 IEEE

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.

Thus, the contributions of this paper are:

1. A process, MoMS, supporting the selection of features to port to hand-held devices or devices with limited storage, such as routers or GPS navigation systems. MoMS could be used to support the reverse engineering, next release problem, and porting of both software and product lines.
2. A reference implementation of MoMS, with state-of-the-art techniques for PRs elicitation, dependency analysis, and multi-objective optimization. It considers customers' satisfaction, storage occupation, and CPU consumption.
3. Two case studies illustrating MoMS as used to process to miniaturize two open-source programs, Pooka (an email client) and SIP (an instant messenger).

The paper is organized as follows: Section II introduces a motivating example. Sections III and IV describe the process of MoMS and its reference implementation. Section V illustrates MoMS with two case studies and analyzes their validity. Section VI discusses the advantages and limitations of MoMS. Section VII summarizes related works. Section VIII concludes and describes future work.

II. Motivating Example

Let us imagine a company, MobileMail, that wants to miniaturize an email client, Pooka, to port it to hand-held devices powered by battery with limited disk storage and memory. 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 to gather Pooka PRs; and (3) quantified constraints of the hand-held devices, *e.g.*, their storage capacity. MobileMail would follow the steps below to miniaturize Pooka:

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

2. 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.

3. 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 class can be found, then the implementation-less PRs are put aside as future enhancements; otherwise MobileMail performs a dependency analysis to identify all the classes implementing the PR, thus identifying all Pooka features.

4. Feature Property Analysis: Then, MobileMail associates to each feature their required property values, *e.g.*, the storage occupation of the compiled program, the average memory and battery consumption of the program, etc. on the targeted hand-held devices. It also identifies the constraints imposed by the hand-held devices, *e.g.*, maximum storage capacity and CPU consumption (described in Section IV).

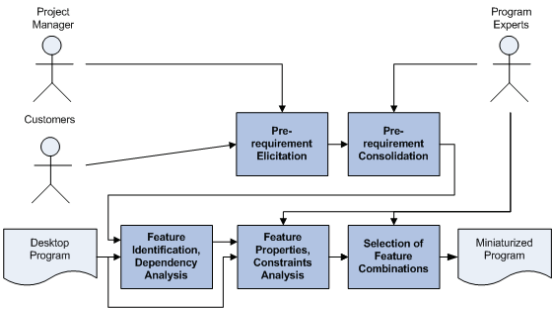

Figure 1: MoMS process in a nutshell

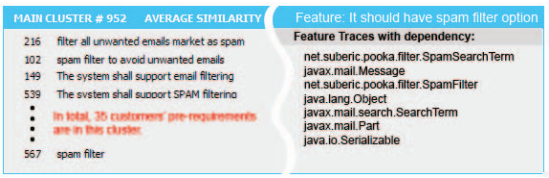

Figure 2: Example of a consolidated PR for Pooka (left) and of a feature and its corresponding Java classes (right)

5. 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 combination. The result is the set of features (and related classes and libraries) of hand-held Pooka.

III. MoMS

This section describes the MoMS process as it would be applied by experts without making any assumption on the available tool support. Figure 1 shows a high-level diagram of the MoMS process. It consists of five steps described below.

1. Pre-requirement Elicitation and 2. Consolidation: A project manager in charge of providing a miniaturized 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.

Then, program experts group similar PRs, label each group, tag grouped PRs as functional or non-functional, and distinguish compulsory PRs from optional ones. Figure 2 shows some (optional) PRs collected and consolidated for hand-held Pooka: a number of customers' requests for an anti-spam filter.

3. Feature Identification: Program experts now trace PRs to implementation units in the source code, *i.e.*, classes, methods,

or functions. Without loss of generality, we consider only Java classes in this paper, thus experts trace PRs to classes. Each traced PR corresponds to a feature. Experts manually validate the features and their traces. Then, experts associate each feature to the complete set of classes (including other classes and libraries) implementing this feature using static analyses. Figure 2 shows a feature and its related classes. The feature list and the corresponding class sets will be used to analyze the properties associated with the features, *e.g.*, to compute the storage size or to estimate the CPU consumption.

4. Feature Property Analysis: Experts assess the impact of each feature (and its implementation) on the constraints imposed by the hand-held devices. Experts also determine the values of the feature properties, *e.g.*, the amount of storage required by each feature, their average memory occupation or battery consumption, under certain usage conditions. Features properties are evaluated based on the experts' experience, perhaps using appropriate (static or dynamic) analysis tools.

5. Selection of Feature Combinations: Experts determine that the program must satisfy a set of L customers, $C \equiv \{c_1, c_2, \ldots, c_L\}$. The program must implement a set of *compulsory* features $ComF$, identified by the experts from among the customers' PRs. In addition, each customer i requires a set $F_i \equiv \{f_{i,1}, \ldots, f_{i,N_i}\}$ of N_i *optional* features that the hand-held version must implement, with $OF \equiv \bigcup F_i$.

A possible miniaturized program can implement $F' \subseteq OF$ optional features. 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, the run-time memory) with an acceptable level of customer satisfaction.

Any company wants to get maximum profit with limited investment; they rank their customers according to the values that they can bring to the company. Thus, the project manager can rank each customer i according to its value, val_i, to her project and her company, where $1 \leq val_i \leq V_{max}$ and $Val \equiv \{val_1, val_2, \ldots, val_L\}$.

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

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

that is the average proportion of customers' requested features that F' contains, weighted by the customers' relative value. We do not set a priority to the optional features because they are already weighted by the numbers and the values of the customers who ask for them in CSR.

The program is composed of M implementation units $IU \equiv \{iu_1, iu_2, \ldots, iu_M\}$. Function $Impl$ is a continuous function that takes as input a set of features and returns the corresponding implementation units.

The porting requires dealing with a set of property values $P \subset \mathbb{R}^K$ concerning the device usage, such as disk occupation and CPU consumption, and with a set of internal constraints $IC \equiv \{ic_1, \ldots ic_K\}$, each of them imposing a set ic_j of acceptable values on the corresponding property values, with $P \in IC \equiv \{p_j \in ic_j \forall j = 1, \ldots, K\}$. Function $Prop$ returns the set of property values of a program: $Prop : IU' \rightarrow P$.

We define a miniaturized program as:

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

i.e., the implementation of the selected optional features F' and of the compulsory features $ComF$. The features must be independent so they can be included or excluded separately, but their implementations are not necessary independent. For example, two independent features can use the same library to process strings. We describe the handling of dependencies between ius in Section IV.

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

$$\min_{F' \in 2^{OF}} (-CSR(F'), Prop(IU'))$$

which solutions are miniaturized programs, implementing subsets of the features of the original program, so that they: (1) maximize customers' satisfaction $CSR(F')$, *i.e.*, minimize their dissatisfaction, $-CSR(F')$ and (2) satisfy the constraints IC by minimizing a set of property values, $Prop(IU') \in IC$, *e.g.*, reduce as much as possible their disk occupation.

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

IV. Implementation

We use state-of-the-art techniques to implement MoMS, with respect to other existing techniques. We support some of the steps with a novel tool, FacTrace[3], for PR management.

1. Pre-requirement Elicitation and 2. Consolidation: We collect PRs from potential customers using the survey feature of FacTrace. FacTrace lets project managers create surveys for their customers using an online survey tool.

We reuse Prereqir [2] 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 [5], and applying an agglomerative nesting clustering algorithm, Agnes [6]. We manually distinguish functional/non-functional and compulsory/optional PRs: Figure 3 shows the FacTrace user-interface to label clustered PRs, fix erroneous clusters, and tag clusters.

3. Feature Identification: We reuse an approach based on information retrieval to trace PRs to classes [7], implemented in FacTrace. Figure 3 shows parts of FacTrace GUI to validate traces between a PR and some classes.

We developed a static analysis tool to discover the dependencies among the implementations of the features to ensure that each feature is traced to all required classes and libraries. The traced PRs are features required by customers to be ported to the hand-held devices.

4. Feature Property Analysis: In our implementation, we consider two properties for each feature: its storage occupation

[3]http://www.ptidej.net/research/factrace

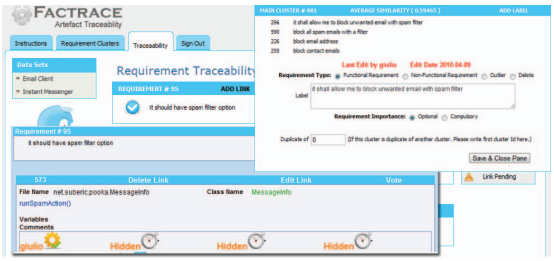

Figure 3: Screen shot of the PR traceability validation tool

and CPU consumption. Storage occupation is important when we port programs to devices with limited storage space, such as routers. We measure it simply as the total size of the classes (and their related libraries) implementing a feature. We count the sizes of classes used by many features only once and define $BCS(f_j) = Bytecode\ Size\ of\ Impl(f_j)$ with $f_j \in F_i$. It is straightforward to generalize BCS to a set of features F_i.

CPU consumption is important because users of hand-held devices, like mobile phones, are sensitive to battery life. Binder and Hulaas "employ[ed] a platform-independent CPU consumption metric, the number of executed JVM bytecode instructions" [3]. We also use the number of executed bytecode instructions (NEBI) for a feature as representative of its CPU consumption and, in turn, of its battery consumption. NEBI is not the most accurate measurement of CPU consumption, because this consumption also depends on the frequency of features usages, peripherals usages (such as GPS), etc. We will replace NEBI with a more accurate estimation of CPU consumption in future work, with no impact on MoMS.

Let us define $M(f_j)$ as the set of all the methods implementing a feature f_j other than $main$ method(s). We denote a method m' calling a method m by $m' \rightarrow m$. Then, we define $TLM(f_j)$ as the set of top-level methods of an execution of f_j: $TLM(f_j) = \{m \mid m \in M(f_j) \wedge \exists m' \notin M(f_j) \wedge m' \rightarrow m\}$. Finally, we define the NEBI measure as $NEBI(f_j) = \sum_{k=1}^{|TLM(f_j)|} NEBI(m_k) \mid m_k \in TLM(f_j)$.

We use JP2 (latest upgrade of J-RAF2) [3] to collect the NEBI for each feature f_j in two ways: If a program has test cases, we obtain $NEBI(f_j)$ by executing the test case(s) of f_j. If the program (or some of its features) does not have test cases, then it is in general impractical to execute each feature manually and we estimate $NEBI(f_j)$ conservatively by executing all the features and by using the method call dependencies to compute $NEBI(f_j) = \sum_{k=1}^{|TLM(f_j)|} NEBI_{max}(m_k) \mid m_k \in TLM(f_j)$, where the maximum NEBI in all executions of m_k is $NEBI_{max}(m_k)$.

5. Selection of Feature Combinations: We support the last step of MoMS by representing solutions of our multi-objective optimization problem [4] as bit-vectors $\vec{x} = \{x_1, \ldots, x_{|OF|}\} \in \{0, 1\}$, where x_j indicates if feature $f_j \in OF$ is included in the combination of features in the solution $\vec{x}$: $x_j = 1$ 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', we define CDR (Customer Dissatisfaction Ratio), BCS (ByteCode Size), and $NEBI$ (Number of Executed Bytecode Instructions) as:

$$\begin{aligned} CDR(\vec{x}) &= -CSR(Sel(\vec{x})) \\ BCS(\vec{x}) &= BCS(Sel(\vec{x}) \cup ComF) \\ NEBI(\vec{x}) &= \sum_{j=1}^{|OF|} (x_j.NEBI(f_j)) + \sum_{j=1}^{|ComF|} NEBI(f_j) \end{aligned}$$

where $BCS(\vec{x})$ and $NEBI(\vec{x})$ are the size of the class files and the number of executed bytecode instructions of an execution of the features in $\vec{x}$ and the compulsory features.

The problem objective is to find a set X of solutions $\vec{x}$, whose elements are Pareto-optimal [8], *i.e.*, solutions superior to all the solutions outside X but not better than the other solutions inside when considering all the objectives. The set of Pareto-optimal solutions is also called Pareto-front. For the problem defined in this paper, $\forall \vec{x} \in X$ and $\forall \vec{y} \notin X$: $\forall\ f(\vec{x}) \leq f(\vec{y}) \wedge \exists f(\vec{x}) < f(\vec{y})$, where $f(x) \in \{CDR(\vec{x}), BCS(\vec{x}), NEBI(\vec{x})\}$ and $f(\vec{x})$ is optimal if $f(\vec{x}) < f(\vec{y})$, because we miniaturize programs.

The set X is the Pareto front, which we compute using the Non-dominated Sorting Generic Algorithm II (NSGA-II) [9], which is a multi-objective optimization algorithm that incorporates elitism to maintain the solutions of the best front found, and its JMetal implementation[4], 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 [10] and a simulated binary cross-over operator [11]. A binary tournament selection operator selects the solution candidate for reproduction. The three fitness functions (to be minimized) are $CDR(\vec{x})$, $BCS(\vec{x})$, and $NEBI(\vec{x})$.

V. 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 and memory occupation of the miniaturized programs as well as the efficiency of MoMS with respect to 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. For example, we could 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. This context does not reduce the applicability of MoMS to other kinds of programs.

[4]http://jmetal.sourceforge.net/

Table I: 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	486,966 LOCs
		5.39 MB	27.3 MB
Binary Sizes	**Java Only**	2.91 MB	11.1 MB
	With Libs	4.1 MB	14.3 MB
Constraint on CSR		> 0.4	> 0.4
Constraint on BCS		< 3.5 MB	< 8.5 MB
Constraint on NEBI		$< 2.0^{10}$	$< 1.2^{10}$

Pooka[5] is an email client written in Java using the JavaMail 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. SIP[6] (now Jitsi[7]) 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.

Table I provides some general descriptive statistics for the two programs and the constraints that we suppose the programs must meet: 0.4 for CSR, 3.5 MB and 8.5 MB for BCS, 2^{10} and 1.2^{10} for NEBI, respectively. We used Pooka to motivate our work in Section II and provide further results on SIP for the sake of completeness and discussions.

Our *research questions* are:

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

We address **RQ1** by measuring and optimizing three different dependent variables: *customer satisfaction*; the *program byte-code size* on disk, represented by BCS; and its *CPU consumption*, represented by $NEBI$. We study how MoMS allows a project manager to select a subset of features that, on the one hand, satisfies customers and, on the other hand, meets device constraints with reduced storage and memory.

We address **RQ2** by comparing the *time* needed to perform the porting activities when applying our implementation of MoMS with the time needed to perform the 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 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/icsm11a/.

[5]http://www.suberic.net/pooka/
[6]http://sip-communicator.org/
[7]http://www.jitsi.org/

1.a. 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. Of the 73 respondents, 70.08% and 80.82% did not actively contribute to the development of an email client and–or instant messenger. Also, we excluded surveys from 14 (9.59%) and 7 (6.85%) respondents who admitted not being confident with email clients and instant messengers, thus obtaining 59 and 66 respondents respectively. Overall, the respondents wrote 599 and 639 PRs for the email client and instant messenger.

To minimize the impact of customer value on subsequent steps, we chose to randomly divide the customers into 7 groups and 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.

1. Pre-requirement Elicitation and 2. Consolidation: Approaches for PR consolidation usually rely on clustering techniques [12], [2]. We clustered the 599 email-client and 639 instant-messenger PRs using FacTrace, which uses Agnes and displays the obtained clusters for their analysis and the identification of a cut-off threshold. The best thresholds were 41% and 46%, below which Agnes mixed different PRs. We thus recovered 221 and 235 clusters.

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 each ~ 9 hours for Pooka and SIP. We assume that the project manager does not use any clustering technique. Having some knowledge of clustering would reduce the time and, therefore, the 9 hours are an upper-bound.

Automatic Time: It took ~ 6 and ~ 8 hours to generate, analyze, and manually validate clusters for Pooka and SIP.

3. Feature Identification: We evaluated the performance of the feature identification implemented in MoMS in terms of precision and recall [5] and time required for this step. We manually created an oracle of features for Pooka and SIP: 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 (41 features) and 830 feature traces (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 an overly large (but compilable) implementation, rather than associate less classes with a feature and possibly miss some important classes. Missing classes may not be found by the dependency analysis, if no explicit dependency lead to them (*e.g.*, reflection is used).

We performed dependency analysis 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. Feature Property Analysis: We use the BCS and $NEBI$ measures defined in Section IV.

The minimum, average, and maximum size of class files per feature are 2,449 bytes, 994,017 bytes, and 2,939,233 bytes, for Pooka, and 43,155 bytes, 1,530,935 bytes, and 4,609,122 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 $1,636,781 \oplus$ [8] $1,358,459 = 2,148,535$ and $3,036,886 \oplus 2,106,357 = 3,601,369$ bytes.

The minimum, average, and maximum NEBI per feature are 0 bytes, 712,372,175 bytes, and 8,828,175,748 bytes, for Pooka, and 0 byte, 363,920,173 bytes, and 9,194,130,192 bytes for SIP. There are features with NEBI of 0 byte: their traced implementation methods were not executed during NEBI counting because no traceability links can be 100% accurate. We again 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 NEBIs are respectively $7,614,678,105 + 87,591,544 = 7,702,269,649$ and $6,129,181,285 + 103,973,943 = 6233,155,228$ 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.

Automatic Time: Summing class file sizes can be done easily with existing tools, for example TreeSize[9], at no cost.

[8]We define $\oplus$ as a sum that does not include the same classes and libraries twice for BCS.

[9]http://www.jam-software.com/treesize_free/

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, 12.95 features for Pooka and 17.31 for SIP. Without considering the constraints on BCS and $NEBI$, the sizes of the miniaturized Pooka and SIP with external libraries vary from $2,148,535$ to $3,696,575$ bytes and from $3,601,369$ to $9,305,678$ bytes, respectively, *i.e.*, when satisfying the minimum and maximum number of customers. The minimum and maximum NEBIs of Pooka and SIP are 7.7×10^9 to 3.07×10^{10} and 6.23×10^9 to 1.92×10^{10}.

Figure 4 shows the Pareto front for Pooka (including libraries). The top (3D) graph shows the Pareto front with all dimensions (NEBI, BCS, and CSR); the other two represent the projections on the NEBI–CSR and BCS–CSR planes

We observe that the distribution of NEBI is sliced (the bottom graph of Figure 4) because we use its conservative estimation. Near the NEBI of 2.0×10^{10}, there are many solutions with different CSRs. The BCS is proportional to the CSR for Pooka (the middle graph of Figure 4).

We highlight two solutions, one with a circle and another with a triangle. On the projection BCS–CSR, both solutions satisfy the constraints and the triangle solution has higher CSR; but on the projection NEBI–CSR, only the circle solution satisfies all the constraints and it has the highest CSR.

Figure 5 shows the Pareto front and its projection for SIP (including libraries). Similar to the results of Pooka, the NEBI of SIP is also sliced (the bottom graph of Figure 5) and the BCS increases with the CSR (the middle graph of Figure 5). From the projection of NEBI and CSR, we observe that some solutions have high NEBI and low CSR, thus we would not use them to miniaturize SIP. The two highlighted solutions satisfy the constraints on NEBI and CSR, but the triangle solution is far outside the limits of BCS.

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

Automatic Time: Execution time was less than 5 minutes for Pooka on a laptop with an Intel Duo 1.5 GHz processor and 4 GB of memory, running 32 bit Microsoft Windows XP. Analyzing SIP required more than 2 GB JVM heap size (impossible on a 32-bit Windows XP) so we used a server CentOS 5.3 with AMD Opteron Dual-Core 2.4GHz and 16GB RAM. The execution time was 6 minutes.

VI. Discussion

We now discuss the results of our case studies, the impact of various factors on the miniaturization process, and outline issues not addressed by MoMS.

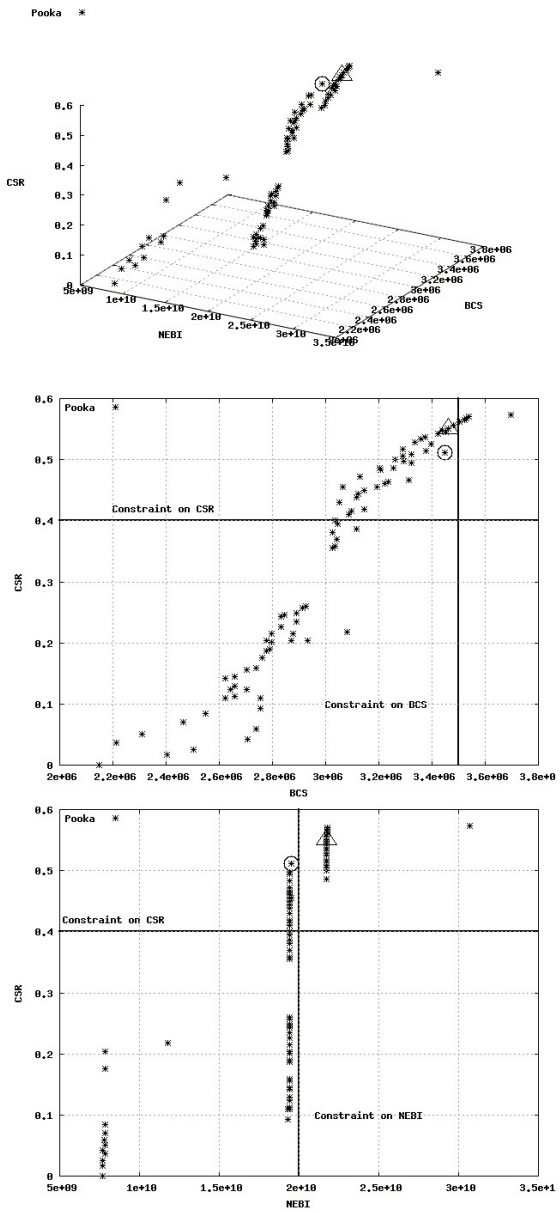

Figure 4: Pareto fronts of Pooka (The circle and the triangle are two solutions discussed in the text.)

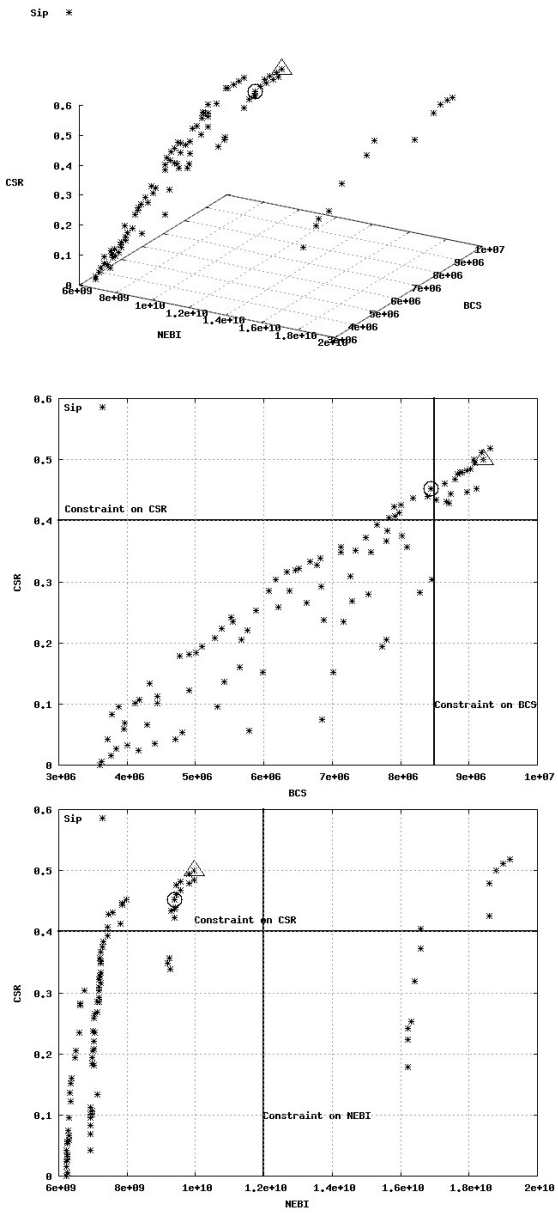

Figure 5: Pareto Fronts of SIP (The circle and the triangle are two solutions discussed in the text.)

A. 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. Depending on her needs, the project 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 $17 + 9 + 135 + 0 + 42 = 203$ 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.
  On average, we saved 176.75 hours of work per system, *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.

B. Factors Impacting MoMS

Code Coupling: Class coupling [13] has an impact on the results of MoMS. A high coupling means that the sizes of the feature combination 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 combination. On the contrary, a low coupling means that the sizes of the feature combination will increase slowly and rise faster than highly-coupled systems as more features satisfy more customers.

JVM and Java Library Version: We assumed that the hand-held devices targeted by the MoMS process include a full JVM and Java libraries. However, hand-held devices may only support a limited JVM; this issue is part of our future work.

C. Issues not Addressed by MoMS

MoMS assists the 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 [14], miniaturizing a program might require removal of dead code, clones, and unnecessary libraries. Our approach removes classes if they are not a part of selected features. We could further reduce BCS values by removing unnecessary methods (if any). Clone refactoring is also possible [15].

Unavailable/Different APIs: Because hand-held devices could only support a limited JVM, a process similar to migrating to a different language [16] or alternative APIs offered by the available class libraries [17] could be a solution. Similarly, applications developed for Linux Debian must use different libraries when being ported to devices with Maemo[10].

GUI Issues: Porting a desktop GUI to a small hand-held device screen 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 [18].

Testing: As with any other transformation techniques, testing of the resulting program is essential to ensure that the program is executable. Well-written test cases are also helpful to accurately measure the dynamic properties of features, such as memory consumption or battery-life consumption.

D. 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 labelled and assessed by three authors. Also, we made sure that each combination produced by the optimization correctly compiled. Finally, as explained in Section IV, we are aware that NEBI only represents a proxy for battery consumption and memory occupation of the program. NEBI could be replaced without impacting MoMS.

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 her 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 this threat by performing the different MoMS tasks on different days, but repeating the process multiple times may give different 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. We also considered two different properties: storage and memory occupations. Yet, programs having different characteristics (*e.g.*, commercial programs, programs from different domains, and different languages) or different properties 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 miniaturization based on customers' requests. 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.

VII. Related work

Our work relates to three areas of software and computer engineering: software miniaturization, requirement engineering, and the use of multi-optimization in hardware and software systems. In general, we extend previous work by proposing a process that unifies a wide range of approaches dealing with PR elicitation, PR consolidation, and software miniaturization, and that can maximize customers' satisfaction under constraints imposed by the hand-held devices. Yet, to the best of our knowledge, only a handful of studies have been previously published on the use of multi-objective optimization to identify the "best" set of features in a product, be it software or hardware and this may be the first work aimed at instantiating a product line from an existing program by decomposing its features, mapping them to PR elicited from customers, and optimizing their combinations with respect to different variables, including customers' satisfaction.

Software Miniaturization: The problem of software miniaturization was introduced by Di Penta *et al.* [14], 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) using clustering and genetic algorithms, guided by the program footprint.

Requirement Gathering: Previous work addressing requirement elicitation and consolidation exists. For example, Goldin and Berry [12] used signal processing techniques to abstract common requirements while Hayes *et al.* [2] proposed the Prereqir approach to consolidate and trace customers'

[10]http://www.maemo.org

requirements to source code using clustering techniques, from which we draw inspiration.

Feature Identification: The identification of the implementation units in source code that implement some features is a well-known problem. It was addressed as early as 1995 by Wilde *et al.* [19], who used 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, much work has been proposed to trace features to code. Chen and Rajlich [20] developed an approach to identify features using Abstract System Dependencies Graphs. Eisenbarth *et al.* [21] combined previous approaches by using both static and dynamic data to identify features. Greevy *et al.* [22] studied the evolution of object-oriented program entities from the point of view of their features. Antoniol and Guéhéneuc [23] proposed an epidemiological metaphor that combines both static and dynamic data to identify features. Poshyvanyk *et al.* [24] combined this previous approach with an LSI-based approach to reduce the effort in identifying the features while further improving precision and recall.

We use an information retrieval-based traceability recovery technique, largely inspired by previous work by Antoniol *et al.* [7].

Requirement Prioritization and Release Planning: Previous work on requirement prioritization includes the pair-wise comparison of requirements through the Analytic Hierarchy Process (AHP) [25] as well as simple requirement ranking. Karlsson and Ryan [26] developed a cost-value approach for requirement prioritization based on AHP. Crucial issues in the application of AHP 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. They also defined [27] a set of heuristics to reduce the number of possible pair-wise comparisons, a process, and a tool to apply AHP.

Saliu, and Ruhe [28] treated the problem of selecting candidate features for release planning as a multi-objective optimization problem, to obtain 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. Finkelstein *et al.* [29] used multi-objective optimization to balance different customers' conflicting requirements.

Hardware Product Lines Optimization: Successful hardware product lines include the Swiss Army Knife and Swatch Watches. Yet only a few studies have been published describing the optimization of such product lines. Nanda [30] describes the application of multi-objective optimization to the design of a universal electric motor. He applied NSGA-II and discussed the Pareto fronts of the inter-dependencies of various design variables and confirmed the applicability of multi-objective optimization to hardware products.

Also, Hassan *et al.* [31] describe the application of multi-objective optimization to characterize the tradeoffs between commonalities and performances among the satellites of a product line. They applied NSGA-II and discussed the number of common technologies with respect to the masses of three satellites under various constraints.

Finally, Kwong *et al.* [32] proposed the application of multi-objective optimization (NSGA-II) to optimize a product line based on a dataset containing customer preference and information on competitors' products.

Software Product Lines Optimization: The previous work most related to ours in the domain of product line optimization is that of Benavides *et al.* [33], in which they proposed an extension to Czarnecki's feature meta-model [34] with so called extra-functional features: features related to quality or non-functional features, expressed as relations among one or more attributes of functional features. They solved a constraint satisfaction optimization problem (CSOP) to reason on feature models and find the optimal number of products or the commonalities among products in a product line but did not consider competing features.

Frenzel *et al.* [35] proposed a method to consolidate a family of software systems, which are variants from one another, into a software product line. They claimed that software variants emerge from the ad-hoc copying of large pieces of code, whole systems even, and their adaptations to different contexts and assumed that the implementations and architectures of the software variants are sufficiently similar. They mapped variants two-by-two manually with each other by reusing the mapping between one software variant and its recovered architecture and the next variant.

To the best of our knowledge, none of the proposed approach provides the complete solution to miniaturize and optimize software systems.

VIII. Conclusion and Future work

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 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 showing the combinations providing better customers' satisfaction (CSR) with small storage size (BCS) and low memory consumption (NEBI). We estimated the time saved using MoMS compared to a manual process: $203 - 44.25 = 158.75$ 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. 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 to 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.

REFERENCES

[1] 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*. New York, NY, USA: ACM Press, 2009, pp. 11–20.

[2] J. H. Hayes, G. Antoniol, and Y.-G. Guéhéneuc, "PREREQIR: Recovering pre-requirements via cluster analysis," in *Proceedings of the* 15^{th} *Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, October 2008, pp. 165–174, 10 pages.

[3] W. Binder and J. Hulaas, "Using bytecode instruction counting as portable cpu consumption metric," *Electronic Notes in Theoretical Computer Science*, vol. 153, pp. 57–77, May 2006.

[4] Y. Sawaragi, N. Hirotaka, and T. Tetsuzo, *Theory of multiobjective optimization*. Academic Press, Orlando, 1985.

[5] W. B. Frakes and R. Baeza-Yates, *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, June 1992.

[6] L. Kaufman and P. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, 1990.

[7] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[8] V. Chankong and Y. Y. Haimes, *Multiobjective Decision Making: Theory and Methodology*. North-Holland, 1983.

[9] 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*. London, UK: Springer-Verlag, 2000, pp. 849–858.

[10] K. Deb and M. Goyal, "A combined genetic adaptive search (geneas) for engineering design," *Computer Science and Informatics*, vol. 26, pp. 30–45, 1996.

[11] K. Deb, *Multi-Objective Optimization using Evolutionary Algorithms*, ser. Wiley-Interscience Series in Systems and Optimization. John Wiley & Sons, Chichester, 2001.

[12] L. Goldin and D. M. Berry, "Abstfinder, a prototype natural language text abstraction finder for use in requirements elicitation," *Automated Software Engg.*, vol. 4, no. 4, pp. 375–412, 1997.

[13] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[14] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo, "A language-independent software renovation framework," *J. Syst. Softw.*, vol. 77, no. 3, pp. 225–240, 2005.

[15] 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)*, 2000, pp. 98–107.

[16] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API mapping for language migration," in *Proceedings of the* 32^{nd} *International Conference on Software Engineering*. ACM Press, May 2010, pp. 195–204.

[17] M. Nita and D. Notkin, "Using twinning to adapt programs to alternative APIs," in *Proceedings of the* 32^{nd} *International Conference on Software Engineering*. ACM Press, May 2010, pp. 205–214.

[18] G. Canfora, G. Di Santo, and E. Zimeo, "Developing Java-AWT thin-client applications for limited devices," *IEEE Internet Computing*, vol. 9, no. 5, pp. 55–63, 2005.

[19] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," in *Journal of Software Maintenance: Research and Practice*, K. H. Bennett and N. Chapin, Eds. John Wiley & Sons, January-February 1995, pp. 49–62.

[20] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Proceedings of the* 8^{th} *International Workshop on Program Comprehension*, A. von Mayrhauser and H. Gall, Eds. IEEE Computer Society Press, June 2000, pp. 241–252.

[21] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *Transactions on Software Engineering*, vol. 29, no. 3, pp. 210–224, March 2003.

[22] S. D. Orla Greevy and T. Girba, "Analyzing software evolution through feature views," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 6, pp. 425–456, November 2006.

[23] G. Antoniol and Y.-G. Guéhéneuc, "Feature identification: An epidemiological metaphor," *Transactions on Software Engineering (TSE)*, vol. 32, no. 9, pp. 627–641, September 2006, 15 pages.

[24] 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)*, vol. 33, no. 6, pp. 420–432, June 2007, 14 pages.

[25] T. Saaty, *The Analytic Hierarchy Process*. McGraw-Hill, Inc., 1980.

[26] J. Karlsson and K. Ryan, "A cost-value approach for prioritizing requirements," *IEEE Software*, vol. 14, no. 5, pp. 67–74, 1997.

[27] J. Karlsson, S. Olsson, and K. Ryan, "Improving practical support for large-scale requirement prioritising," *Requirement Engineering*, vol. 2, no. 1, pp. 51–60, 1997.

[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*, 2007, pp. 105–114.

[29] 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," *Requirement Engineering*, vol. 14, no. 4, pp. 231–245, 2009.

[30] J. Nanda, "Multiobjective genetic algorithm for product design."

[31] R. A. Hassan and W. A. Crossley, "Multi-objective optimization of communication satellites with two-branch tournament genetic algorithm," *Journal of Spacecraft and Rockets*, vol. 40, no. 2, pp. 266–272, 2003.

[32] C. Kwong, X. Luo, and J. Tang, "A multiobjective optimization approach for product line design," *IEEE Transactions on Engineering Management*, vol. 58, no. 1, pp. 97–108, 2011.

[33] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated reasoning on feature models," in *Proceedings of* $17th$ *International Conference on the Advanced Information Systems Engineering*, 2005, pp. 491–503.

[34] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, June 2000.

[35] P. Frenzel, R. Koschke, A. P. J. Breu, and K. Angstmann, "Extending the reflexion method for consolidating software variants into product lines," in *Proceedings of* $14th$ *Working Conference on Reverse Engineering*, 2007.