Reuse or Rewrite: Combining Textual, Static, and Dynamic Analyses to Assess the Cost of Keeping a System Up-to-date

Giuliano Antoniol, *Dépt. de Génie Informatique, École Polytech. de Montréal*, *antoniol@ieee.org*
Jane Huffman Hayes, *Dept. of Computer Sci., Univ. of Kentucky*, *hayes@cs.uky.edu*
Yann-Gaël Guéhéneuc, *Dept. of Computer Sci., Université de Montréal*, *guehene@iro.umontreal.ca*
Massimiliano di Penta, *Dept. Eng., University of Sannio*, *dipenta@unisannio.it*

Abstract

old: (1) when no comments or documentation exist, it is difficult for developers to understand how a system works; (2) when no requirements exist, it is difficult to know what the system actually does. We present a method, named ReORe (Reuse or Rewrite) that assists developers in recovering requirements for a competitor system and in deciding if they should reuse parts of their existing system or rewrite it from scratch. Our method requires source code and executable for the system and assumes that requirements are preliminarily recovered. We apply ReORe to Lynx, a Web browser written in C. We provide evidence of ReORe accuracy: 56% for validation based on textual and static analysis and 94% for the final validation using dynamic analysis.

Keywords: Maintenance, reuse, requirements, documentation, static analysis, dynamic analysis, feature identification, data mining.

1. Introduction and Problem Statement

Maintenance is the longest and most costly phase of the software lifecycle [13][16]. The cost of maintenance is often measured in the number of expended person hours. It is widely accepted that most of the effort invested in maintenance goes for understanding the source code, generally written by developers other than the maintainers. Often, software is poorly documented: it lacks in-line comments, design, and requirements. As a result, it is not so surprising that 47% of software maintenance effort is devoted to understanding the software [12][16].

Although much research has been carried out to reduce maintenance costs, to the best of our knowledge, there has been little work on the application of these methods and techniques to *a priori* maintenance, *i.e.*, to help managers make the decision to reuse parts of the old system or to completely rewrite a new system. Our work addresses this application by introducing a method, ReORe (Reuse or Rewrite, pronounced Ree-Oh-Ree). Given a set of requirements (possibly a generic requirement document

Undocumented software systems are a common challenge for developers performing maintenance and/or reuse. The challenge is two-f

generated using semi-automated techniques presented in [11]), ReORe uses Information Retrieval (IR) techniques in combination with dynamic feature identification to assess code that can be reused to bring a system up-to-date with competitor systems.

A typical scenario in which to apply our method follows. A company owns a proprietary system, S_{prop}, that is becoming obsolete because of its technology and of market shifts: competing companies have introduced new technologies and functionalities that S_{prop} lacks. The management of the company faces the dilemma of either maintaining S_{prop} to bring it up-to-date or rewriting a new system from scratch (getting out of the market is not an option). Data is required to support an informed decision: the obsolescence of S_{prop} and the amount of reusable code must be quantified. This data must include the requirements of the competing systems, $S_{cptitor}$, (listing their technologies and features) and the entities of the source code of S_{prop} that could be reused to implement these requirements (including any structures, classes, functions, methods, and so on). Underpinning the ReORe method is the idea that the management of the company will assign tasks to developers, experts of the application, and software architects to optimize effort and minimize the usage of the most valuable resources.

In this paper, we present our method and illustrate its application in a real-world situation: We apply ReORe to the Lynx text-oriented open-source Web browser (*i.e.*, S_{prop}) to assess the feasibility of evolving this browser to fulfill the requirements of state-of-the-practice browsers such as Mozilla Firefox and Microsoft Explorer (*i.e.*, $S_{cptitor}$). In the case study, the first three authors played the role of developers while the fourth author acted in the role as the most valuable resource of the application: the architect. ReORe is based on available consolidated technologies, its novelty being the integration of these technologies. It does not guarantee per se that all the relevant information is gathered, its goal is to minimize the effort of most valuable resources while identifying as much as possible reusable code.

The paper is organized as follows: Section 2 discusses the recovery of requirements. Section 3 presents ReORe. Section 4 details the design of the case study. Section 5 describes the application of the approach to Lynx while Section 6 discusses the results and threats to validity. Section 7 presents related work. Finally, Section 8 concludes and outlines future work.

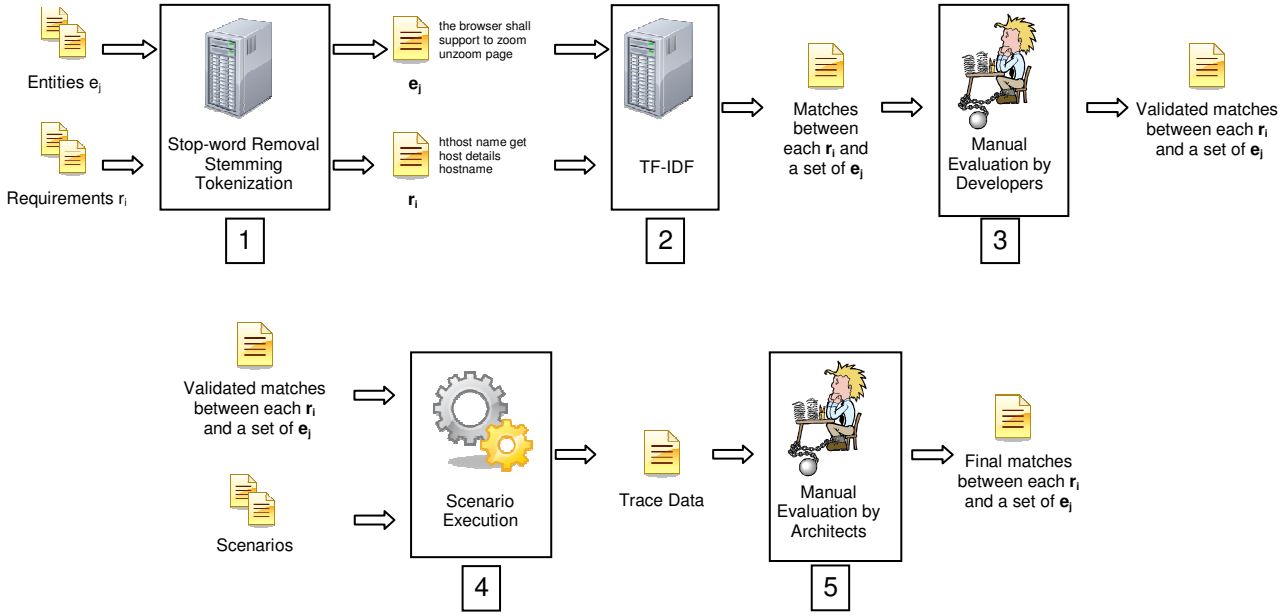

Figure 1. Synopsis of ReORe.

2. Recovering Requirements

The first part of our method is concerned with obtaining the requirements of the competing systems, $S_{cptitor}$. Most systems do not have comprehensive requirements. Therefore, the developers must collect requirements themselves, because it is unlikely that the competitors will provide them freely. In this context, the developers perform absolutely legal actions by purchasing and exercising the functionalities of $S_{cptitor}$ [15].

Obtaining requirements is a preliminary activity to our method but out of the scope of this paper. Therefore, we only succinctly present the essential details of another method, PREREQIR [11], to obtain requirements. The PREREQIR method divides into three steps performed by the developers to:

(1) obtain and vet a list of requirements from diverse stakeholders using anonymous questionnaire,

(2) structure the requirements by mapping them into a representation suitable for grouping via pattern-recognition and similarity-based clustering, and

(3) analyze the clustered requirements to divide them into a set of essential and a set of optional requirements.

The outcome of the PREREQIR method is a set of so-called synthetic sentences that together form the requirements of a system. The synthetic sentences represent the cluster of requirements and therefore subsume the stakeholders' understanding of the system.

3. The ReORe Method

Once a set of requirements for the competing systems is available, ReORe can be applied in five steps, shown in Figure 1. ReORe assumes the availability of a set of requirements R written in textual form and of a set of source code entities E from S_{prop} that implement the system under study. ReORe also uses available entities: C functions (or structures) or C++ (Java) classes or methods. In the following, ReORe is applied to C code and C functions are the reusable entities.

Step 1: Textual processing of requirements and code. The developers process both the requirements R and the set of entities E in parallel. Any requirement $\mathbf{r_i}$ or entity $\mathbf{e_j}$ is mapped into a textual representation and the developers apply standard natural-language processing techniques to extract sets of characterizing tokens, $\{\mathbf{r_i}\}$ and $\{\mathbf{e_j}\}$, from $\{\mathbf{r_i}\}$ and $\{\mathbf{e_j}\}$. Specifically, stop-word removal, stemming, and tokenization are applied to obtain the two sets of tokens $\{\mathbf{r_i}\}$ and $\{\mathbf{e_i}\}$.

Step 2: IR-based tracing of requirements to code. Developers use term frequency–inverse document

frequency [4] to compute the similarity between any $\mathbf{r_i}$ and any $\mathbf{e_j}$. Developers consider each $\mathbf{e_j}$ as a document and $\mathbf{r_i}$ as a query against the set of documents and obtain a ranking of each $\mathbf{e_j}$ for each $\mathbf{r_i}$. Developers only keep as a match those entities $\mathbf{e_j}$ that have a similarity greater than zero and, more generally, higher than a given threshold. We advocate the use of outlier analysis because developers are interested in the outliers of the similarity distribution: pairs ($\mathbf{r_i}$, $\mathbf{e_j}$) with a relatively high similarity score. The standard outlier threshold definition[1] (see also [18]) may or may not be pertinent and thus the threshold should be manually verified and assessed. Consequently, developers consider the entity $\mathbf{e_j}$ as relevant to the implementation of the requirement $\mathbf{r_i}$ if $\mathbf{e_j}$ is returned by the query $\mathbf{r_i}$ with a similarity higher than some threshold identifying outliers in the similarity distribution.

Step 3: Selection of relevant traceability matches. Developers manually evaluate the rankings obtained in Step 2. For each $\mathbf{r_i}$, they retain only the entities $\mathbf{e_j}$ that they believe to concretely participate in the implementation of $\mathbf{r_i}$; they label these $\mathbf{e_j}$ as reusable entities and link them to the requirement $\mathbf{r_i}$. This evaluation is necessarily manual because only the developers can assess, using their knowledge of the system and contextual information, whether an entity $\mathbf{e_j}$ *really* participates in the implementation of requirement $\mathbf{r_i}$.

Step 4: Dynamic analysis. Developers build and execute scenarios that exercise the requirements for which a consensus concerning their implementing entities exist. Two kinds of scenarios must be developed: (1) scenarios exercising core functionalities and requirements and (2) scenarios specifically devoted to some particular requirement. The execution of the scenarios supports the collection of trace data. Core scenarios serve to identify the core implementation entities, likely to constitute the infrastructure of the system. More specific scenarios collect data that can then be studied by the architect in the fifth step, to again rank the entities and confirm/invalidate the matches from the previous steps.

Step 5: Final validation. Combining both textual and static analysis (entity extraction, stop-word removal, stemming, and tokenization), dynamic analysis (trace data), and requirements, developers are able to identify and to validate the entities $\mathbf{e_j}$ implementing a requirement $\mathbf{r_i}$. This last step should be performed by the architect of the system to benefit from her expertise without overloading her with work.

A major advantage of ReORe is it greatly reduces the possibility of introducing, during any step, subjective judgments that could influence the final results by clearly separating and defining these steps. The multi-step process uses the developers in the early steps, whose results are then checked by the architects in the final step. Within the early steps, the developers separately develop lists of requirements and possible matching entities. A different developer is responsible for "breaking" any ties in their results. The architects do not have information about the method applied by the developers to obtain the requirements and the list of entities that may implement these requirements. Also, the architects are not aware that the developers work from ranked lists that are based solely on textual and static information.

4. Case Study Definition and Context

The *goal* of the case study reported in the following sections is to apply ReORe to make an informed decision on maintaining or rewriting an outdated system. The *purpose* of the study is to assess the applicability of ReORe. The *quality focus* of the study is ensuring high traceability between requirements and the existing source code, to help managers make their decision. The *perspective* of the study is that of a company developing a Web browser that must decide to maintain or rewrite a new browser.

For the sake of reproducibility, we assume the existence of a company developing the Lynx Web browser. Lynx[2] is known as "the text Web browser", *i.e.*, it is a free, open-source, text-only Web browser and Gopher client for use on cursor-addressable, character cell terminals. Its development began in 1992 and it is now available on several platforms, including Linux, UNIX, and Windows. It has been used widely both in academia and in industry and can reasonably be expected to be found in some companies.

We mimic the process that Lynx developers must follow to assess the feasibility and the cost of replacing portions of Lynx code to port it/enhance it to modern technologies in an attempt to create a competitor to more recent browsers such as Mozilla Firefox, Opera, or Microsoft Internet Explorer.

As in any real-world scenario, we assume that the complete set of requirements for competing Web browsers is unavailable or incomplete, and that the only available documents (possibly also incomplete) are user manuals. Thus, developers would first apply a method to recover the requirements of competing systems. They could use our own PREREQIR method, summarized in Section 2. Then, they would apply ReORe to match the requirements with entities from the source code of their Web browser and to check the matches using code inspection and dynamic trace analysis. The Lynx Web browser is implemented in C. Thus, without any loss of generality, we choose to focus on functions as the main entities implementing the requirements.

[1] http://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm

[2] http://lynx.isc.org/

The developers do not need to compare the requirements from the competing Web browsers with those of Lynx because such a comparison would only help to quantify the obsolescence of their own browser. This information would be of scarce assistance because their company already knows that their system is obsolete and that it must evolve or fade out. Furthermore, such a mapping would not help to understand how much code is reusable.

As mentioned above, Lynx is largely written in C, mostly following the ANSI standard; macros are used to parameterize function definitions. The case study utilizes Lynx version 2.8.5 compiled and executed under a Linux RedHat server version 5. The Lynx 2.8.5 code-base is organized into three main directories (`lib`, `src`, and `www`); it contains about 2,074 functions contained in 91 source code files for a total of 147 KLOC (counted using the GNU word counting utility `wc`). Configuration definition, function forward declarations, macros, and global data declarations are organized into 156 header files for a total of 27 KLOC.

The default configuration has been used to produce an executable version of Lynx (*i.e.*, no special switch was used). Lynx behavior is largely modifiable via a configuration file and thus configuration details such as external mailer, external editor, or the address of a printer were later passed on to the executable via the configuration file.

For the sake of this case study, the first three authors played the role of developers while the last author played the role of architect. The empirical study aims to address the following research questions:

- **RQ1:** is ReORe able to effectively reduce the information processed in the different steps while ensuring the traceability between requirements and the existing code?
- **RQ2:** what is the effort required for the application of ReORe?

5. Applying ReORe to evolve Lynx

This section describes each step of ReORe as applied to the Lynx case study.

Premise: Recovering Requirements. The first step of our method is concerned with obtaining the requirements of the competing systems, $S_{cptitor}$. Developers chose to apply PREREQIR [11].

For our case study, the first three authors used a convenience sample of more than 200 of their colleagues and acquaintances and sent them an anonymous questionnaire to obtain prioritized lists of requirements for Web browsers. Among the 200 recipients, 25 sent back the questionnaires, out of which 22 were kept for this study. Any questionnaire that was not totally completed was omitted.

Consequently, the developers obtained 128 essential requirements (*e.g.*, "the browser shall support back and forward actions") generated in the form of synthetic sentences; these are essential requirements in that at least two respondents submitted them. Another set of 181 optional requirements have been identified; these are singleton requirements: either variants of one of the 128 essential requirements, requirements provided by a single respondent, or very general requirements (*e.g.*, "it should be easy to use"). Thus, given that Lynx is made of about 2,074 functions and that there are, overall, 128 + 181 = 309 requirements, the set of all possible matches among requirements and functions is 309 × 2,074 = 640,866 matches, *i.e.*, if all functions were to contribute to all requirements. A summary of the number of requirements, functions and matches—after each step of the matching—is reported in Table 1.

Table 1: Lynx functions, requirements, and matches requirements–functions.

Essential Requirements	128
Optional Requirements	181
Total Number of Requirements	309
Number of Functions	2,074
Number of Possible matches	640,866
Matches with similarity > 0	138,896
Matches with similarity > 20%	738
Dynamic Analysis Matches	88
Final Inspection Matches	83

Step 1: Textual processing of requirements and code. Developers applied the first step of our method: they used natural-language processing techniques to tokenize the requirements and the 2,074 functions of the Lynx Web browser. They obtained the two sets $\{\mathbf{r_i}\}$ and $\{\mathbf{e_j}\}$ including, respectively, 309 and 2,074 items. Traditional filtering (*e.g.*, punctuation removal), camel-case splitting, stopping, and stemming are applied to the requirements and functions, $\{\mathbf{r_i}\}$ and $\{\mathbf{e_j}\}$, that are then mapped into a vector space. The dictionary (after processing) contains 5,192 stems, thus requirements and functions are represented with 5,192 dimensions.

Step 2: IR-based tracing of requirements to code. Second, for each tokenized requirement $\mathbf{r_i}$, developers ranked the functions that possibly implement the requirements. We use a vector space model computed with TF-IDF and cosine similarity ranking [4]. Pairs ($\mathbf{r_i}$, $\mathbf{e_j}$) with ranking of zero are discarded in this step to reduce the search space from 640,866 to 138,896 candidate matches. Nevertheless, a manual inspection of the remaining matches is still infeasible. The cosine similarity distribution is highly skewed toward low values and the developers, thus, limited themselves to the matches in the higher fringe of the distribution.

Mean and median of similarity are 0.023 and 0.012, respectively. The 25% and 75% percentile values are 0.006 and 0.026. Maximum similarity value is 62% and inter-quartile range is about 0.021. Standard outlier definition [18] suggests that anything above a threshold of 1.5 times the inter-quartile range from the 75% percentile is to be considered a mild outlier, *i.e.*, about 0.06 (6%) similarity in our case. However, the relative distance of Lynx code to the new requirements is substantial and a threshold computed as above requires a similarity just above 6% for a pair ($\mathbf{r_i}$, $\mathbf{e_j}$) to be kept for inspection. The use of such a threshold would retain about 12,000 matches, still quite a sizeable inspection task. The inspection of a few matches led the developers to the suitable compromise of choosing a threshold of 8/9 times the inter-quartile range, which yields a value of 20% similarity. Consequently, 738 pairs ($\mathbf{r_i}$, $\mathbf{e_j}$) rank above this threshold. Although this threshold may still appear to be a low, it is worth noting that no pair ($\mathbf{r_i}$, $\mathbf{e_j}$) has a score above 62% similarity.

For example, the requirement:

> *"the system shall allow internet cookies to be downloaded in the local file system"*

is matched to the set of functions:

Functions	Ranks
src/LYCookie.c::newCookie	0.3489
src/LYCookie.c::LYProcessSetCookies	0.3088
src/LYCookie.c::LYSetCookie	0.2942
src/LYCookie.c::ARGS1	0.2797
src/LYDownload.c::LYdownload_options	0.2559

where, in the first line, it is reported that a function `newCookie`, contained in file `LYCookie.c`, belonging to the directory `src`, is matched to the requirement with a similarity of 0.3489.

Step 3: Selection of relevant traceability matches. Based on the chosen similarity threshold, requirements {$\mathbf{r_i}$} are divided into two sets (see Table 2). The first subset, A, is the set of the 186 requirements (out of the 309) for which at least one match is above the threshold; this subset is actually the set containing the most likely pairs ($\mathbf{r_i}$, $\mathbf{e_j}$).

Table 2: Requirement breakdown - Sets A and B.

Requirements	Set A	Set B
Essential	88	40
Optional	98	83
Overall	186	123

The second subset, B, is the subset comprising the remaining 123 requirements for which no one single function among the 2,074 achieved at least 20% similarity. Regardless of the set, each requirement $\mathbf{r_i}$ is associated with its ranked list of matching functions. Table 3 reports the mean and standard deviation of the ranked lists for the two sets, A and B.

As reported in the literature on traceability, relevant matches are usually in the top ranked positions [1][14][10]. Yet, sometimes, even a match that is ranked very low can still be relevant. As shown in Table 3, ranked lists in Sets A and B contain more then 400 functions. Inspecting and validating such a high number of pairs ($\mathbf{r_i}$, $\mathbf{e_j}$) would simply result in discarding a very high number of false positives.

Table 3: Summary statistics of ranked lists for Sets A and B.

	Average candidate list length	Candidate list standard deviation
Set A (>= 20%)	464	523
Set B (< 20%)	425	478

Therefore, to balance effort and false positives, developers designed the validation of matches in the following way. For each {$\mathbf{r_i}$} in sets A and B, the top five most relevant candidate pairs ($\mathbf{r_i}$, $\mathbf{e_j}$) are retained. In addition, five other matches are randomly chosen with a uniform distribution between the remaining matches. Consequently, even matches ranked very low still have at least a chance of being manually validated.

Two developers independently evaluated each of the matches by inspecting function stems manually to assess whether the function indeed implements the requirement. Each developer vetted each function as either "Yes," "No," or "Maybe." A majority consensus was determined for each requirement after combining the developers' votes conservatively: only functions vetted as "Yes" by both developers are promoted to the next step, all other combinations are considered to be incorrect matches and are removed from consideration.

Table 4: Agreement and disagreement between developers' votes on the two sets.

Set	Ranking Range	Yes	No	One Yes
A	**1-5**	128	226	324
	6-10	2	790	20
B	**1-5**	25	407	69
	6-10	0	570	10

Vetted lists were merged by the third developer who was not involved in the vetting process. The decision to only retain functions ranked "Yes" by both developers for the next step may appear overly conservative. However, subsequent steps (dynamic analysis and final validation) could re-introduce removed functions, if really relevant. Furthermore, static analyses, such as caller-callee or data

dependency relations in the final set of functions, could be used to recover missed matches; we plan to implement this recovery in future work.

As reported in Table 4, among the five highest ranking functions (listed as "1-5"), "Yes" are predominant, with an average of 14% for the 186 set A requirements. Table 4 also shows that, for the top five matches, there is a 35% disagreement (only one of the developers said "Yes"). Among the five randomly-selected functions, few are evaluated as "Yes." For example, in Set A, only two "Yes" were given for functions in the random five (listed as "6-10"). As expected, for subset B, the number of positive agreements is only 25, *i.e.*, 2%: most matches are discarded (977 total "No" responses for 1,230 matches).

Step 4: Dynamic analysis. Developers used dynamic analysis to gather information and further filter candidate matches. Given each requirement for which at least one function, *i.e.*, a pair ($\mathbf{r_i}$, $\mathbf{e_j}$), was vetted "Yes" by both developers, another developer built a scenario that attempted to exercise the relevant functions and performed dynamic analysis. There are 55 scenarios for set A and 13 for set B. Once a scenario was built and executed with success, trace data were collected using Valgrind/Callgrind as done in [3][2][3].

The dynamic analysis was performed in two steps. The first step aimed at executing three basic scenarios, to gather functions likely to constitute the core functions contributing to the implementation of Lynx (*e.g.*, functions handling HTTP, SSL, and FTP protocols). These functions may be relevant to a possible rewriting of the Web browser.

Table 5 reports the numbers of distinct functions and their relative scenarios. The numbers are cumulative: for example, the functions in the line relative to the SSL scenario are added on top of the basic HTTP scenario. Overall, in this first step of dynamic analysis, 547 functions were flagged as potentially reusable.

Table 5: Recovered backbone functions via three basic scenarios.

Scenario	Number of Functions
HTTP	382
SSL	137
FTP	28

Following this first step, for each of the 55 requirements in set A and 13 in set B, a scenario was created and dynamic data collected. Scenario creation is a costly activity. First, a requirement is carefully read; the Lynx configuration file is then inspected to check if its parameters have been set up properly (*e.g.*, external mailer, printer, startup page, mouse activation, etc); then, Lynx user documentation is inspected to verify if any other set up is required. Finally, a check is performed to verify that the operating system configuration is sound and that other support programs are running. For example, to exercise a scenario related to the requirements on FTP or HTTPS, the developer must ensure that an FTP server or a secure Web server is available.

Once a proper configuration and environment are ascertained, a scenario is built in a narrative way. Continuing with the example of HTTPS, a scenario could be "*the user starts Lynx, passing as a parameter the file linx.cfg; she types G and enters the address of the secure Web server https://127.0.0.1; when prompted, she enters her username and password; finally, she types Q and exits the Web browser.*" Four hours of work were required to setup the environment, define, and execute the first scenario. Overall, about 40 hours of manual activity were needed to define and run 68 scenarios (i.e., 55 for set A) and to analyze data and filter matches.

Once trace data was collected for a scenario, the developer verified whether or not the candidate functions appeared in the trace. If not, it was necessary to check if the binary code of the missing functions was linked into the Lynx executable or if it was dead code and, thus, would never be called. Sometimes, the source code was inspected to check for the presence of processor macros preventing compilation, such as hardware architecture or operating system dependencies.

Table 6: Retained and discarded matches in the dynamic trace collection phase.

Set	Ranking Range	Retained	Discarded
A	**1-5**	77	51
	6-10	1	1
B	**1-5**	10	15
	6-10	-	-

As reported in Table 6, in this step, for set A, 60% of the pairs ($\mathbf{r_i}$, $\mathbf{e_j}$) in the top five positions were promoted to the final verification step while only one of the two functions ranked in position 6-10 were retained. It is worth noting that set B contains 1,230 pairs ($\mathbf{r_i}$, $\mathbf{e_j}$) and only ten out of the 25 (See Table 4) were considered relevant once dynamic analysis was performed.

Step 5: Final validation. Finally, the architect manually validated the matches of each requirement with each function using several sources of available information, ranging from execution traces, source code, and interactions with the system. Figures reported in Table 7 show a substantial agreement with those in Table 6, confirming the matches from previous steps. It is important to underscore that the author playing the role of

[3] Note that if test cases already exist for some of these scenarios, this step can be performed with much less effort. However, the proposed method does not assume the existence of such test cases.

the architect was not involved in prior phases to avoid any possible bias.

Table 7: Retained and discarded matches in the last phase.

Set	Ranking Range	Retained	Discarded	Added
A	**1-5**	73	4	12
	6-10	1	-	-
B	**1-5**	9	1	1
	6-10	-	-	-
Overall		**83**		**13**

This step is important to further reduce possible mismatches. As shown by the data of Table 6, this step actually helped in adding 13 pairs of requirements-functions. Interestingly, three of these functions were not tagged as relevant to three distinct requirements in the previous steps. These three functions are not compiled under Linux due to preprocessor flags; they were discovered by inspecting the code of other relevant Linux functions. The other 10 were functions that the dynamic analysis discarded as non-executed.

Table 8: Final requirement break down into Sets A and B.

Requirement	Set A	Set B
Essential	23	5
Optional	23	4
Overall	46	9

.

Overall, the architect spent about 10 hours on this step. As shown in Table 8, the applied method provides evidence that 28 requirements out of the 181 essential requirements and 27 out of the 128 optional requirements have at least one pair ($\mathbf{r_i}$, $\mathbf{e_j}$) and, thus, some form of initial implementation. However, three out of the five essential requirements of set B (in Table 8) are variants of other requirements that were already captured in the 23 essential requirements. Two of the three deal with printing and one with bookmarking. The remaining two requirements of set B are related to the possibility of changing font size, font type, and browser look-and-feel. These are not variants of any of the requirements in set A.

Overall, developers obtained evidence that only 25 essential requirements have some initial implementation.

6. Discussion

Clearly, not all steps have the same cost. Step 3 is labour intensive though supported by IR ranking. The sheer size of the system may force developers to reduce the number of inspected links. Step 4 requires the explicit identification and execution of scenarios to refine the results from Step 3. Writing scenarios is quite a complex task and there are not guarantees that, in general, it is possible to develop scenarios for each requirement. We believe that developers' and managers' judgment plays a key role in balancing knowledge of the system with the effort of recovering reusable code. Indeed, company knowledge will be often enough to identify large portion of reusable code, key scenarios or meaningful thresholds and, thus, to reduce the effort required to apply ReORe.

In scenarios such as the one presented above, a ground truth will hardly ever be available. An *a-priori* gold set of pairs ($\mathbf{r_i}$, $\mathbf{e_j}$) with which to compare the results of ReORe is unavailable due to the lack of knowledge for legacy systems, imprecision in collecting requirements, and the focus of the implementation or evolution strategy on a subset of the most important customer requirements. Therefore, we cannot quantify ReORe in terms of information retrieval measures such as precision and recall because neither "the" set of correct matches ($\mathbf{r_i}$, $\mathbf{e_j}$) nor the subset of implemented $\mathbf{r_i}$ are known.

One way to assess the accuracy and circumvent the intrinsic impossibility of computing recall or precision is to consider ReORe as a waterfall process in which each step plays the role of the Oracle for previous steps. Accuracy is thus quantified as the percentage of matches for which an agreement between any two subsequent steps was found; this accuracy is representative of the effort saved.

It is important to note that a function may be related to different requirements and that some of the 96 functions reported in Table 7 may also be in the set of core functions executed by the scenarios of Table 5. Indeed, out of the 96 functions, there are only 52 unique functions. Interestingly, only one function of the 10 in set B (see Table 7) was not in the set of functions collected in set A. This fact seems to suggest that it does not really pay-off to investigate low ranked functions and pairs ($\mathbf{r_i}$, $\mathbf{e_j}$). However, Table 8 may suggest the usefulness of inspecting pairs ($\mathbf{r_i}$, $\mathbf{e_j}$) in set B to check for the presence or absence of some initial implementation of requirements in the legacy system. Indeed, out of the five essential requirements discovered in Set B (see Table 8), three are somehow captured by other requirements in Set A (i.e., by some of the 23), the other two are newly discovered essential requirements not comprised in the 23.

Table 9: Accuracy achieved in the two main phases: manual ranking versus dynamic analysis and dynamic analysis versus final inspection.

		Accuracy	
		Dynamic Analysis	**Final Inspection**
Set A	**1-5**	60%	95%
	6-10	50%	-
Set B	**1-5**	40%	90%
	6-10	-	-
Overall		56%	94%

Table 9 reports the accuracy computed as defined above. For example, dynamic analysis validated 77 matches and discarded 51 of the set A top five positions (see Table 6). This means that the dynamic analysis only exercised 77 functions out of the 128 classified relevant in Table 4. In other words, 60% of functions went on to the next steps; about 40% of the matches produced in the first steps are false positives. These false positives are not inspected by the architect and, thus, also represent effort saved: the most valuable resource does not need to inspect these matches. As reported in Table 9, accuracy in the final inspection achieved a higher value; indeed, the 95% accuracy for set A and top five matches means that only a few false positives slipped through the dynamic analysis step.

It should also be noted that functions activated by the three basic scenarios and reported in Table 5 may or may not be matched to any high level requirements. These functions are likely to participate in some kind of infrastructure, a set of utilities on top of which requirements are implemented. It is interesting to note that, out of the 52 functions recovered, 35 are not contained in the set of 547 functions. These are functions specific to some requirements and are not part of the infrastructure or of the three basic browser pieces of functionality: access via HTTP, FTP, or SSL to a remote resource.

- **Answer to RQ1:** we believe that the answer to RQ1 is yes because there is an order of magnitude of difference between the initial search space (in Table 1) and the number of pairs inspected by the architect (in Table 7). Moreover, matches have been discovered and validated with a subset of essential requirements and the method also enables the architect to recover missing matches. Further work beyond the goal of this paper is needed to quantify false negatives. Evidence collected in the case study also suggests that ReORe may well pay off in a smoother evolution process, at least for a Web browser. Indeed, only about 20% of the essential requirements of a modern and graphical browser such as Mozilla Firefox or Microsoft Internet Explorer are reflected in Lynx source code: out of 2,074 Lynx functions, only 52 relate to one or more "modern" requirements. This may well be the situation of many legacy systems that risk being confined to a market niche.

- **Answer to RQ2:** the scenario we have presented is that of a company facing market pressure to evolve one of its products due to technology evolution, competitor market share, or repositioning to a more profitable niche. We believe that Lynx is representative of a product that is confined to particular needs, such as a text-based environment. Therefore, any company facing a similar evolution challenge would be willing to invest a few hundred hours to make an informed decision. Indeed, in the case of Lynx, it took the developers about 85–90 hours of manual labor to perform requirement mapping into the existing code base. Clearly, more effort must be acknowledged for the preliminary step of requirement gathering, market analysis, and customer survey. Yet, the authors are not Lynx experts or Lynx developers, whereas any company facing the evolution dilemma is likely to possess a deeper know-how of and expertise with the source code, thus reducing the effort of the requirement mapping. With Lynx, the effort required to assess matches was just a fraction of the effort required to develop a system of the size of Lynx. However, we have only one data point and cannot generalize to other systems or domains without further studies.

Our method is underpinned by the idea that the most expensive steps, such as scenario building and trace collection, and the most valuable resources, such as the architect in the final validation step, should be carefully planned and managed to minimize resource usage (*i.e.*, **RQ1**). We emphasize that subsequent filtering and information reduction steps did not prevent the recovery of missing functions in the last step, and, in fact, discovered new functions relevant to three requirements. The application of ReORe took only about one hundred hours, saving valuable time and effort and assisting the management in making an informed decision (*i.e.*, **RQ2**).

6.1 Threats to Validity

External validity threats concern the generalization of our findings. We believe that our findings support the evidence that ReORe can be applied and that modern IR techniques and tools help reduce the cost of data collection. The scenario presented is realistic and likely to be representative of many real-world situations. However, the size of the systems, the technology gap, or the difficulty in obtaining requirements may prevent others from applying our method *as-is*.

Construct validity threats concern the relationship between the theory and the observation. Such threats can be due to errors introduced by measurement instruments. Requirement and code indexing and similarity computation were performed using widely adopted toolsets. For example, we used the Perl stopper and stemmer available from the Virginia Polytechnic Institute and State University. Also, a TDF-IDF implementation is made available by the open-source Lucene project. Nevertheless, we cannot exclude the possibility that another chain of tools may produce slightly different results or that different developers would rate functions in different ways. One critical element is the choice of the thresholds. We experimented with rank thresholds (*e.g.*,

the five most similar elements plus an additional five randomly selected pairs ($\mathbf{r_i}$, $\mathbf{e_j}$)); other developers may apply different strategies, such as score thresholds (*e.g.*, the elements with scores in the top 1%) or the gap threshold algorithm introduced by Zhao [19]. These strategies would produce different sets of candidate matches that must be verified and thus, as shown in [7], may impact accuracy. We plan to better assess the influence of thresholds in future work. Dynamic analysis depends on the test cases used to exercise the system. It might be that some functions were not invoked despite their relevance. Also, it is possible that some functions were (conditionally) compiled only under particular configurations. Nevertheless, the final validation step limited this threat, by recovering functions discarded by the dynamic analysis.

Reliability validity concerns the possibility of replicating the study and obtaining the same results. The source code and documentation of Lynx are publicly available; the set of collected requirements are available from the authors upon request.

Internal validity refers to the influence of independent variables on dependent variables and the existence of confounding factors. The main threat to the internal validity of this study is the level of subjectivity introduced by developers. However, as explained in Section 2, a multiple-step approach limits this threat because each step validates/integrates the results of the previous one. Also, bias was limited by separating each step so that developers contributing to one step were not involved in other steps. Also, different techniques (static analysis, IR, dynamic analysis, and code reading) were applied. Moreover, the final ranking was made by an expert who was not aware of the process and tools used in the previous steps.

7. Related work

Our method relates to prior work on traceability using static and dynamic data and on recovering requirements.

7.1 Traceability using Static Data

Many advances have been made in the area of traceability of static data (such as source code, comments, design documents, user manuals, and so on). Researchers apply simple keyword searches, rule-based approaches, and information retrieval techniques to the traceability of many software artifacts. Gotel and Finkelstein [9] define the traceability problem. Antoniol *et al.* [1] describe the use of information retrieval techniques to recover traceability between code and user's manuals. Marcus and Maletic [14] repeated Antoniol's study and also apply latent semantic indexing. Cleland-Huang *et al.* introduce event-based tracing [5] and goal-centric tracing. Spanoudakis *et al.* [17] discuss rule-based traceability of UML models and code. Egyed *et al.* [8] discuss value-based traceability. Hayes *et al.* [10] discuss the use of feedback as well as introduce measures for evaluating traceability methods.

7.2 Traceability using Dynamic Data

A hybrid approach to feature identification and comparison was presented by Antoniol and Guéhéneuc in [3]. The approach applies parsing, reverse engineering, processor emulation, and dynamic analysis to obtain data on the functions executed when exercising a feature. The data is analyzed using an epidemiological metaphor to identify functions most likely to be part of the implementation of the feature. This approach can be used to understand large, multi-threaded object-oriented systems. It produces a ranked list of functions and classes participating in a feature. It supports the extraction and examination of any micro-architecture as well as supports the comparison of micro-architecture evolutions.

7.3 Recovering Generic Requirements for an Application

The authors previously discuss the mining of domain-specific mental models to obtain pre-requirements information (PRI) in [11]. Such PRI (or generic requirements) can be used to support traceability, can be applied as checklists when building applications within the domain, can be used to support reuse, etc. PREREQIR uses partition around medoids and agglomerative clustering to obtain, structure, analyze, and label textual PRI from a group of diverse stakeholders. In a study of the Web browser domain, with information obtained from 22 stakeholders, we found that about 55% of the PRI were common to two or more stakeholders and 42% were outliers. We automatically labeled the common (essential) and outlier (optional) PRI (82% correctly labeled) to build a generic requirement document for Web browsers. The current work assumes the existence of such a requirement document.

8. Conclusions and future work

This paper proposed ReORe, a method to trace requirements into source code using static, textual, dynamic, and requirements analyses, to assess whether to rewrite or reuse code from a system being updated to challenge competing systems.

We evaluated our method using a case study related to the assessment of the Lynx Web browser, with respect to its competitors such as Mozilla Firefox or Microsoft Internet Explorer. Due to the very nature of the problem addressed, traditional measures of precision and recall could not be obtained. Instead, we evaluated the accuracy of ReORe as the percentage of candidate matches judged

relevant in one step and retained in the following step. We showed that the accuracy in the two labor intensive steps was 56% and 94%. These results support the evidence that the most valuable resources (the architect, in our case study) are less involved in non-productive tasks, *i.e.,* the task of discarding spurious matches. As reported in the discussion, only 6% of false positives were discarded by the architect.

There was some evidence suggesting that it does not pay-off to investigate low ranked functions and pairs ($\mathbf{r_i}$, $\mathbf{e_j}$). However, there was also evidence that the evaluation of low ranked pairs ($\mathbf{r_i}$, $\mathbf{e_j}$) resulted in the discovery of some functions that might not have been found otherwise. In the case of Lynx, only two essential requirements (out of the original 128) and one function were from requirements with top similarity score below 20%.

Overall, we concluded that the evolution of Lynx could require quite a substantial effort because only a fraction of the code (about 25 %) can be reused and only 25 essential requirements are reflected in the present code base. Lynx is a niche product and likely will remain so. The lesson learned in the case study of Lynx is that the more a company waits, the more likely it will have to rewrite its system from scratch.

As part of our future work, we plan to examine the use of static analyses for recovering missed matches. We plan to assess the influence of different threshold strategies. We also plan to apply ReORe to other systems with other developers.

Acknowledgements

This work was partially funded by Canada Research Chair Tier I in Software Evolution, a NSERC Discovery Grant. It is also partially funded by NASA under grant NAG5-11732. We thank those colleagues who answered our questionnaire, from which we drew the essential and optional Web browser requirements.

9. References

[1] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. Recovering Traceability Links between Code and Documentation. IEEE Transactions on Software Engineering, Volume 28, No. 10, October 2002, 970-983.

[2] Giuliano Antoniol, Yann-Gaël Guéhéneuc: Feature Identification: A Novel Approach and a Case Study. ICSM 2005: 357-366

[3] Antoniol, G., Guéhéneuc, Y.-G.: Feature Identification: An Epidemiological Metaphor. IEEE Trans. Software Eng. 32(9): 627-641 (2006).

[4] Baeza-Yates, R., B. Ribeiro-Neto. Modern Information Retrieval, Addison-Wesley, 1999.

[5] Cleland-Huang, J., Chang, C. K., Christensen, M. J.: Event-Based Traceability for Managing Evolutionary Change. IEEE Trans. Software Eng. 29(9): 796-810 (2003).

[6] Cleland-Huang, J., Settimi, R., Ben Khadra, O., Berezhanskaya, E., Christina, S.: Goal-centric traceability for managing non-functional requirements. ICSE 2005: 362-371.

[7] Eaddy, M., Aho, A. V., Antoniol, G., and Guéhéneuc, Y.-G.: Cerberus: Tracing Requirements to Source Code Using Static, Dynamic, and Semantic Analysis. Proceedings of the International Conference on Program Comprehension, June, 2008.

[8] Egyed, A., Biffl, S., Heindl, M., Grünbacher, P.: Determining the cost-quality trade-off for automated software traceability. ASE 2005: 360-363.

[9] O.C.Z. Gotel and A.C.W. Finkelstein. An analysis of the requirements traceability problem. In 1st International Conference on Requirements Engineering, pages 94--101, 1994.

[10] Hayes, J. H.; Dekhtyar, A.; Sundaram, S. K., "Advancing candidate link generation for requirements tracing: the study of methods," Transactions on Software Engineering, Volume 32, Issue 1, Jan. 2006 Page(s): 4 – 19.

[11] Hayes, J. H., Antoniol, G., Guéhéneuc, Y.-G.: PREREQIR: A Window into Mental Models, University of Kentucky Technical Report TR 493-08, March 6, 2008.

[12] Lehman, M.M. "Programs, Life Cycles, and Laws of Software Evolution". Proceedings of the IEEE, vol 68, no 9, 1980.

[13] Lientz, B.P., and Swanson, E.B. Software Maintenance Management, Addison-Wesley Publishing Company, 1980.

[14] Marcus, A.; Maletic, J. "Recovering Documentation-to-Source Code Traceability Links using Latent Semantic Indexing," Proceedings of the Twenty-Fifth International Conference on Software Engineering 2003, Portland, Oregon, 3 – 10 May 2003, pp. 125 – 135.

[15] Samuelson, P.: Reverse Engineering Under Siege. Communications of the ACM, Volume 45, Issue 10, Oct. 2002, Pages 15-20.

[16] Schach, S., Jin, B., Wright, D., Heller, G., and Offutt, J.: Determining the distribution of maintenance categories: Survey versus empirical study. Kluwer's Empirical Software Engineering 8(4):351-365.

[17] Spanoudakis, G., Zisman, A., Pérez-Miñana, E., Krause, P.: Rule-based generation of requirements traceability relations. Journal of Systems and Software 72(2): 105-127 (2004).

[18] Venables, W., Ripley, B.: Modern Applied Statistic with SPLUS. Springer, Fourth Edition, 2002.

[19] Zhao, W., Zhang, L. , Liu, Y., Luo, J., Sun, J.: Understanding How the Requirements are Implemented in Source Code. Proceedings of the Asia-Pacific Software Engineering Conference, December, 2003.