Improving Traceability Link Recovery Methods through Software Artifact Summarization

Jairo Aponte
Dept. de Ingeniería de Sistemas e Industrial
Universidad Nacional de Colombia
Bogotá, Colombia
jhapontem@unal.edu.co

Andrian Marcus
Department of Computer Science
Wayne State University
Detroit, MI, USA
amarcus@wayne.edu

ABSTRACT

Analyzing candidate traceability links is a difficult, time consuming and error prone task, as it usually requires a detailed study of a long list of software artifacts of various kinds. One option to alleviate this problem is to select the most important features of the software artifacts that the developers would investigate. We discuss in this position paper how text summarization techniques could be used to address this problem. The potential gains in using summaries are both in terms of time and correctness of the traceability link recovery process.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *documentation, enhancement, restructuring, reverse engineering, and reengineering.*

General Terms

Algorithms, Measurement, Documentation, Performance, Design, Experimentation, Human Factors, Verification

Keywords

Summarization, Program Comprehension, Information Retrieval, Traceability Management

1. PROBLEM DESCRIPTION

The importance and benefits of recovering and managing traceability links between software artifacts during software evolution has been long establish in the research community and industry. The TEFSE Workshop series is testimony to that and also to the challenges raised by these processes.

Many solutions to the traceability link recovery problem are based on the use of text retrieval techniques [1,7,8,10,11,16], under the assumption that extracting and analyzing textual information contained in the software artifacts is an effective way to determine whether they are related. Most such methods consist of several key steps organized in a pipeline architecture, where the output from each step constitutes the input for the next one. In broad terms, the four major steps proposed are: document parsing, extraction and pre-processing; corpus indexing with an IR method; ranked list generation; and analysis of candidate links. During the last step a list of candidate links is provided to software engineers for examination. They have to review each candidate link in order to determine those that are correct links and discard the false positives. This is a laborious activity that is not undertaken with enthusiasm by developers or other stakeholders.

The main challenges in the analysis of candidate links are:

1. It requires a detailed study of a long list of software artifacts of various kinds. The size of these artifacts ranges from one line of text to dozens of pages or more.
2. Most text retrieval techniques are based on rather complex algorithms and the end results are not transparent to the developers. In other words, it is hard for the developers to determine what attributes of the artifacts determined the algorithm to decide that they are related and should be linked.
3. The list of candidate links often has too many false positives.

In this position paper we propose and discuss solutions to address problem #1 in the above list. Specifically, we propose automatically generating summaries (i.e., concise descriptions) of software artifacts, and offering developers these summaries as a first tool during candidate link analysis. Small artifacts (e.g., a method with less than ten lines of code; a short section in a document; etc.) do not require such summaries as they are easy to read by the developers. Large artifacts (e.g., a class with hundreds of methods; a long chapter in a document; etc.), on the other hand, can benefits from summarization. We expect that developers would be able to make proper decisions on many candidate links, without reading in details the original artifacts, but only their summaries instead. Some of the summaries will not be informative enough to help in these decisions, so developers would still have to read the original artifact. The overhead in such cases is minimal (i.e., reading the summary in addition to the artifact) and it is outweighed significantly by the potential benefits in the other cases.

2. SOLUTION

We propose using techniques from text summarization in order to create summaries for text based software artifacts. For mixed artifacts, such as, the source code, we need hybrid summarization techniques that combine textual and structural information.

The main issues we are discussing in this paper are how to generate the summaries and then how to evaluate them in the context of the traceability link recovery process.

TEFSE'11, May 23, 2011, Waikiki, Honolulu, HI, USA

2.1 Summarization of Software Artifacts

One of the most promising applications of summarization is as a complement or a second level of abstraction for information retrieval tools, since often they return a large number of documents that overwhelm their users. For instance, automated summarizing tools are needed by internet users who would like to utilize summaries as an instrument for knowing the structure or content of the returned documents, in advance, and in that way, be able to effectively filter out irrelevant results.

Within the software engineering field, researchers have investigated whether it is possible and useful to summarize software artifacts automatically and help developers offering them an exact and concise representation of the content of the originals [3,4,5,12,14,15,17]. Supporting software comprehension tasks has been the primary use suggested for such summaries, which are based mostly on text retrieval, machine learning, and natural language processing techniques. Some of these summarization solutions are suitable for our proposed solution, however, we need to answer the following questions before research can move forward.

2.1.1 What Kind of Summaries to Generate?

We argue for the generation of *informative summaries*, which should be capable to represent the original artifacts. An *informative summary* provides succinct description of the original document, giving an idea of what the whole content of document is all about and what is its structure. This type of summaries are particularly suited for analyzing candidate traceability links, as in an ideal situation the generated summary should substitute the original document entirely, and therefore, the software engineer would be able to use the summary instead of the full artifact.

From the point of view of their relationship to the source, the summaries we plan to create can be extracts or abstracts, depending on the type of artifact. *Abstractive summaries* are meant to produce important information about the document in a new way, at a higher level of abstraction than the original document and usually include information which is not present in it. Such summaries are suitable to offer developers a high-level view of the role of a software entity in the system, by using information about how this entity is being used by other entities and the semantic relationships that define this usage. *Extractive summaries*, on the other hand, are obtained from the content of a document by identifying its most important sections and reproducing them verbatim (e.g. paragraphs, sentences, clauses, signatures, terms, etc.).

2.1.2 What Should be Included in the Summaries?

The major challenge is summarizing mixed artifacts, such as, source code, where information is encoded differently than in natural text documents. One issue that we need to address is determining what is relevant in source code and should be included in the summaries. Clearly the answer would be different for various source code elements (i.e., class vs. method) and also may differ between programming languages.

One way to address this issue is through empirical studies. Two types of studies can be used: one in which we ask developers to provide answers to question about what they think should be included in a summary; and another one where we can ask developers to manually write summaries of source code elements and then determine what they used in these summaries. Also, developers will be required to evaluate summaries with different content.

During these studies, we will answer research questions like: Where does the information included by developers in summaries come from? What type of lexical and semantic information do developers favor for generating summaries? What type of structural information do developers include in their summaries? How long are the summaries generated by developers? The outcome of the studies will indicate what information should be (ideally) included in the automatically generated summaries.

2.1.3 What Kind of Artifacts Should be Summarized?

The majority of IR-based traceability recovery approaches have been applied to software artifacts, such as, requirements, source code, external documentation, design documentation, test cases, and bug reports [1,7,8,10,11,16]. Therefore, generating summaries for a wide variety of software artifacts, which have different formats, abstraction levels and granularities, is needed.

Many artifacts are formed essentially of free text documents (e.g., stakeholder requests, documents that capture the business context of the system, user manual pages, mailing lists, bug discussions, interviews, etc.). These are obvious candidates for summarization and for such artifacts we can utilize existing automatic text summarization techniques, created by natural language processing research community over the last 50 years [9]. One note mentioned before has to do with the length of each artifact. The document granularity is usually determined before the text retrieval technique is used to generate the candidate links. Short artifacts will not require summaries, whereas long ones will do.

Source code artifacts (e.g., methods, classes, packages, etc.) are another category of artifacts that should be summarized. Once again, the granularity is determined before the candidate link generation. Different source code artifacts will require different summarization techniques.

Finally, other mixed artifacts, such as diagrams can be considered for summarization. The challenge here is that usually, most diagrams are already visual summaries of more complex structures. How to summarize such artifacts is an issue open to debate and probably harder to answer than for the previously mentioned type of artifacts.

2.1.4 What Summarization Techniques to Use?

As mentioned, for the text-based artifacts we can use extractive approaches to text summarization using statistical methods, since they do not require heavy processing for language generation and have produced satisfactory results in large-scale applications [9]. Most of these artifacts are based on natural language document, which have been the subject of many summarization studies in the past.

For source code artifacts, the summarization tools we envision will generate the summaries in several stages:

- Text retrieval. Extract most relevant terms using text retrieval techniques. These can be the same used to generate the candidate links or different ones.
- Structure based vocabulary. Extract most relevant terms based on their purpose in the code (i.e., method name, class name, etc.).
- Natural language processing. Convert a subset (or all) of the text based summary into human readable sentences.

- Structural information. Add additional structural information about source code artifacts (e.g., super class, callees, callers, etc.).

The textual and structural components of the source code summaries should be separated. More than that, the user should have the option of seeing them together or separate.

Feedback mechanisms should also be incorporated. Specifically, the user should be able to select parts of the summaries that should not be included in the future, or which should be emphasized. Also, the user should be able to indicate additional information that should be included in the summaries.

One issue that also needs to be addressed in this application is how to present the summaries to the developers. The answer to this problem is easier to find when the final content of the summaries is established.

2.1.5 How to Evaluate the Summaries?

Summaries of source code artifacts can be evaluated independent of the traceability context.

We can use *intrinsic* evaluation techniques, common in the field of text summarization, to measure how much an automatic summary resembles summaries generated by humans. Two approaches can be used: *offline*, also called *automatic evaluation*, as it does not require human intervention and usually involves the comparison between the system's output and a gold standard and *online*, which requires humans to assess the output of the summarization system according to some predefined guidelines.

For the intrinsic offline evaluation (that is, *target-based-evaluation*) we need to use the developer created summaries as a *gold standard*, that is, an "ideal summary" for each software artifact. Using this gold standard, we can assess the quality of the automatic summaries using several measures from text summarization, such as: Precision, Recall, Cosine Similarity, the Pyramid method, Rouge, etc [9]. This type of evaluation offers an objective view of the quality of automatic summaries.

The intrinsic online evaluation (known as *direct evaluation*) involves developers analyzing and rating automatic summaries using a Likert scale, according to the ability of the summaries to depict the intent of the software entity they summarize. Once again, developers need to be involved. This evaluation will help determine which summarization approach developers believe approximates the best the intent of software entities.

Based on the results of this intrinsic evaluation, we can choose the summarization approaches to use as a support for analyzing candidate traceability links, as performance in an intrinsic evaluation often predicts real-world usefulness. Moreover, these techniques provide repeatable, inexpensive, and automatically-scorable evaluations, whose results are useful for tool development in terms of offering feedback as to how we might improve summarization tools.

2.2 Evaluation in Context

The ultimate goal is to determine the effects of summarization on the task of analyzing candidate traceability links (i.e. *extrinsic evaluation*). Specifically, to assess the impact of various software artifact summarizers on the decision process that a software engineer employs during candidate link selection. Such an empirical evaluation needs the followings:

- Datasets containing a variety of software artifact types including requirements, design documents, and source code. Ideally, each dataset used in the evaluation will be associated with a trace matrix that defines the set of correct traceability links between the artifacts in the system.
- Subjects who will analyze candidate links, provided by IR-based traceability recovery methods, and determine those that are actual links. They should be divided to work under different conditions, including a full-artifact condition (where the subjects have access to the original artifacts), a tool condition (where the subjects have access only to the automatically generated summaries from the originals), a human condition (where the subjects have access only to the sentence-based human generated summaries).
- Traceability link recovery tools that implement different IR methods to generate ranked lists of candidate links based on their similarities.
- Metrics that allow us to assess the effect of using summaries on the performance - in terms of retrieval accuracy - of an IR-based traceability recovery method. Measures commonly used are: Precision, Recall, and F-Measure. Time and effort based metrics should also be used.
- Post-task questionnaires to get information from each participant regarding the usefulness of summaries during the experiments, what further information they would have liked within the summaries, and additional information that allow us to analyze the work done by them.

3. RELATED WORK

Summarization technology has been recently applied to several types of software artifacts with promising results. For instance, brief and accurate descriptions of various source code artifacts have been proposed as a suitable tool to support program comprehension [3,4]; abstracts of bug report discussions, generated using conversation-based classifiers, were proposed as a suitable instrument during bug report triage activities [15]; the summarization of the content of large execution traces was suggested as an tool that can help programmers to understand the main behavioral aspects of a software system [5]; an abbreviated and accurate description of the effect of a software change on the run time behavior of a program was proposed to help developers validating software changes and understanding modifications [2]; high level descriptions of software concerns were designed for raising the level of abstraction and improving the productivity of developers, while working on evolution tasks [14]. Moreover, natural language-based documentation such as requirement records, user stories, manual pages and e-mail conversations can be summarized using general purpose text summarization techniques, in order to reduce the effort and time spent by developers reading, understanding and evolving the artifacts of a software project [9,19].

Regarding evaluation of summaries, it is worth noting that, within the broad field of summarization research, this is one of the most difficult, controversial and challenging tasks, since in most of the cases there is no clear idea of what constitutes a good summary, it is possible to obtain more than one correct summary for the original source, and subjectivity of humans judges has negative effects on the results. Despite of these problems, several approaches have tried to assess the quality of the summaries either intrinsically, by measuring their inner quality usually

against an ideal summary, or extrinsically by measuring their effectiveness for a given task [6].

Within software engineering research, most of the approaches for assessing summaries are informal. For example, in [17] the authors present an approach to generate comments for methods by identifying and lexicalizing the most relevant units. The generated comments were evaluated by asking developers how much accurate, adequate and concise those descriptions were. An exception to this informal situation is [15], where bug reports summaries were evaluated by using intrinsic measures such as Precision, Recall, F-score and Pyramid method, to assess the informativeness, redundancy, irrelevant content and coherence. Then, these results were compared against scores assigned by human judges to the same features. In that sense, the term-based summaries created in [3] from source code using information-retrieval techniques were evaluated using the Pyramid method. Also, the descriptions of source code produced in [4] underwent intrinsic-online evaluation for assessing the agreement between developers.

4. BEYOND TRACEABILITY

In addition to the potential benefits of the software summaries in the candidate link selection process, we believe they could be consumed not just by developers, but also by tools. For example, we envision using the source code summaries to support tools for automatic reverse engineering of legacy code, re-documentation, etc. We expect the summaries to be used by existing software searching and navigation tools.

5. ACKNOWLEDGEMENTS

Andrian Marcus was supported in part through grants from the US National Science Foundation (CCF-1017263; CCF-0845706; CCF-0820133).

6. REFERENCES

[1] Antoniol, G., Canfora, G., Casazza, G., Lucia, A. D. and Merlo, E. 2002. Recovering Traceability Links between Code and Documentation. *IEEE Trans. Softw. Eng.*, 28, 10 (October 2002), 970-983.

[2] Buse, R. and Weimer, W. 2010. Automatically documenting program changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 33-42.

[3] Haiduc, S., Aponte, J. and Marcus, A. 2010. Supporting Program Comprehension with Source Code Summarization. In *Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering – Volume 2*. ACM, New York, NY, USA, 223-226.

[4] Haiduc, S., Aponte, J., Moreno, L. and Marcus, A. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *Proceedings of the 17th Working Conference on Reverse Engineering*, 35-44.

[5] Hamou-Lhadj, A. and Lethbridge, T. 2006. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*. IEEE Computer Society, Washington, DC, USA, 181-190.

[6] Hariharan, S. and Srinivasan, R. 2010. Studies on intrinsic summary evaluation. *Int. J. Artif. Intell. Soft Comput.*, 2, 1/2 (April 2010), 58-76.

[7] Hayes, J. H., Dekhtyar, A. and Osborne, J. 2003. Improving Requirements Tracing via Information Retrieval. In *Proceedings of the 11th IEEE International Conference on Requirements Engineering*. IEEE Computer Society, Washington, DC, USA, 138- 147.

[8] Hayes, J. H., Dekhtyar, A. and Sundaram, S. K. 2006. Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. *IEEE Trans. Softw. Eng.*, 32, 1 (January 2006), 4-19.

[9] Jones, K. S. 2007. Automatic summarising: The state of the art. *Inf. Process. Manage.*, 43, 6 (November 2007), 1449-1481.

[10] Lormans, M. and Deursen, A. V. 2006. Can LSI help Reconstructing Requirements Traceability in Design and Test? In *Proceedings of the Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Washington, DC, USA, 47-56.

[11] Marcus, A. and Maletic, J. I. 2003. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering*, IEEE Computer Society, Washington, DC, USA, 125-135.

[12] Murphy, G. C. 1996. Lightweight Structural Summarization as an Aid to Software Evolution. PhD Thesis, University of Washington, 1996.

[13] Putrycz, E. and Kark, A. 2008. Connecting Legacy Code, Business Rules and Documentation. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008.

[14] Rastkar, S. 2010. Summarizing software concerns. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ACM, New York, NY, USA, 527-528.

[15] Rastkar, S., Murphy, G. C. and Murray, G. 2010. Summarizing Software Artifacts: A Case Study of Bug Reports. In *Proceedings of the International Conference on Software Engineering*. ACM, New York, NY, USA, 505-514.

[16] Settimi, R., Cleland-Huang, J., Khadra, O. B., Mody, J., Lukasik, W. and DePalma, C. 2004. Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts. *In Proceedings of the Principles of Software Evolution, 7th International Workshop*. IEEE Computer Society, Washington, DC, USA, 49-54.

[17] Sridhara, G., Hill, E., Muppaneni, D., Pollock, L. and Vijay-Shanker, K. 2010. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 43-52.

[18] Witte, R., Li, Q., Zhang, Y. and Rilling, J. 2008. Text mining and software engineering: an integrated source code and document analysis approach. *IET Software*, 2, 1 (February 2008), 3-16.

[19] Zhou, X. 2008. Discovering and summarizing email conversations. PhD Thesis, University Of British Columbia, 2008.