Towards a Benchmark for Traceability

Eya Ben Charrada[*] David Caspar Cédric Jeanneret Martin Glinz
Department of Informatics, University of Zurich, Switzerland
charrada@ifi.uzh.ch, s0292538@access.uzh.ch, jeanneret@ifi.uzh.ch, glinz@ifi.uzh.ch

ABSTRACT

Rigorously evaluating and comparing traceability link generation techniques is a challenging task. In fact, traceability is still expensive to implement and it is therefore difficult to find a complete case study that includes both a rich set of artifacts and traceability links among them. Consequently, researchers usually have to create their own case studies by taking a number of existing artifacts and creating traceability links for them. There are two major issues related to the creation of one's own example. First, creating a meaningful case study is time consuming. Second, the created case usually covers a limited set of artifacts and has a limited applicability (e.g., a case with traces from high-level requirements to low-level requirements cannot be used to evaluate traceability techniques that are meant to generate links from documentation to source code). We propose a benchmark for traceability that includes all artifacts that are typically produced during the development of a software system and with end-to-end traceability linking. The benchmark is based on an irrigation system that was elaborated in a book about software design. The main task considered by the benchmark is the generation of traceability links among different types of software artifacts. Such a traceability benchmark will help advance research in this field because it facilitates the evaluation and comparison of traceability techniques and makes the replication of experiments an easy task. As a proof of concept we used the benchmark to evaluate the precision and recall of a link generation technique based on the vector space model. Our results are comparable to those obtained by other researchers using the same technique.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement

General Terms

Standardization

[*]Author funded by the Swiss National Foundation under Project PDFMP2-122969

1. INTRODUCTION

Traceability supports to a great extent the tasks of maintaining and evolving software systems. In fact, it allows tracing the implementation back to the requirements and design documents and thus facilitates the comprehension of the code. Traceability links are also useful for analyzing the impact of change and estimating the effort needed for implementing it. Furthermore, traceability links can be used to ensure that all new requirements are implemented and all impacted artifacts are updated.

Although very beneficial, traceability is rarely used because it is expensive to implement and maintain [20]. Therefore, in the last years, several researchers have developed techniques and tools for generating traceability links automatically [1] [12] [13] [15] [6] or semi-automatically [9]. The rapid progress in this field of research has increased the need for performing easy and rigorous validations and comparisons of traceability techniques and tools.

To satisfy this need, the traceability community is currently working on the definition of benchmarks for traceability [5]. A benchmark is *"a test or set of tests used to compare the performance of alternative tools or techniques"* [17]. It is difficult to acquire meaningful and non-trivial data sets that can be used to create traceability benchmarks [5] because there are almost no publicly available projects that include traceability links.

In this paper, we present a candidate benchmark for traceability that includes a rich data set with end-to-end traceability links. We developed the benchmark based on the software for an irrigation system that is published in a book about software design [10]. The benchmark includes all the typical artifacts that are produced during the development of a software system. The main feature of the proposed benchmark is that it provides end-to-end traceability links. As a proof of concept, we used our benchmark to evaluate the results obtained by the RETRO traceability link generation [13], and compared these results to those obtained by other researchers using the same tool or the same technique. The results we obtained were comparable to those obtained by other researchers.

Our paper is structured as follows. In Section 2 we give a brief introduction to traceability and discuss the main challenges related to the evaluation of new traceability techniques. Section 3 is about the benchmark development:

first, we present the rationale for a traceability benchmark, then we explain our benchmark and discuss the properties that such a benchmark should have. We present a proof of concept in Section 4, where we explain how we used our benchmark to evaluate the results obtained from a traceability link generation tool. In section 5 we discuss the threats to validity and limitations of our benchmark. The next steps of our work are presented in Section 6. Finally we discuss the related work in Section 7.

2. BACKGROUND AND MOTIVATION

2.1 Traceability

Software traceability is defined as *"the ability to relate artefacts created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other, the stakeholders that have contributed to the creation of the artefacts, and the rationale that explains the form of the artefacts"* [21].

Traceability is used to support different maintenance activities [21]. It is useful for change impact analysis as it allows identifying the parts affected by a change and thus estimating the effort needed for applying the change. It also supports software verification and validation as it allows checking that all requirements have been implemented in the system and that the system satisfies its specification. Program comprehension can be much easier when traceability links are available because developers can easily trace code elements back to the original requirements, which give the rationale behind the implementation. They can also trace code elements back to the design and architecture documents to get a more abstract view of the system.

Much research has been conducted to support the generation, the maintenance and the use of traceability links. Among these three subjects, the generation of links is currently the most active one. The reason is that defining traceability links is still the most important and the most challenging task. Additionally, maintaining and using traceability links only make sense if the traceability links are defined. Therefore, we focus our work on the generation of links.

Various approaches have been developed to generate traceability links among different types of software artifacts automatically. Most of these approaches are based on Information Retrieval (IR) models such as the probabilistic IR model or the vector space model. The probabilistic IR model computes the probability that two documents are related and uses the calculated probability to rank generated links [1]. In the Vector Space IR model, the similarity between two documents is calculated as the cosine of the angle between two vectors D_1 and D_2, where the elements of D_1 are the weights of the vocabulary terms in the first document and D_2 the weights of the terms in the second document [12].

Generated links are usually evaluated in terms of precision (*"the number of correct retrieved links (C) divided by C plus the number of retrieved false positives"* [11]) and recall (*"The number of correct retrieved links (C) divided by C plus the number of correct missed links"* [11]).

In order to improve the quality of the generated links, many researchers combined IR techniques with other methods such as analyst feedback [12], machine learning [6] or execution tracing [8]. These methods improved the quality of generated links. However, the precision of generated links is still low when the recall is high.

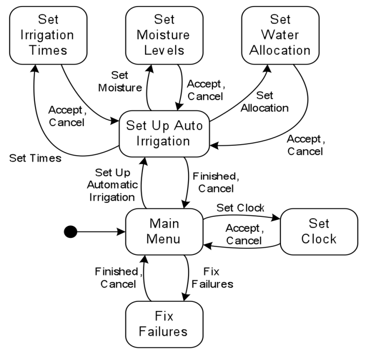
2.2 Evaluating traceability link generation techniques: The Challenge

As more and more effort is spent to enhance and improve traceability link generation techniques, the need for a rigorous evaluation of these techniques is increasing. However, it is difficult to find a project that includes a rich set of artifacts and the traceability links among them so that it can be used to evaluate link generation techniques. Indeed, as traceability is expensive to implement, almost all publicly available projects have either no or only partial traceability. Projects that do have traceability links are in most cases confidential and cannot be published. Hence, researchers usually develop their own examples to evaluate their approaches. There are two main problems related to developing one's own example. First, the development of a meaningful and large enough example takes time. This distracts researchers from their original goal, namely elaborating effective traceability techniques. Second, the cases are constructed specifically for one particular method or technique; therefore they are not necessarily usable by other researchers in the field. For example, if a researcher develops an example that includes the traceability links between requirements and design documents, the example can only be used for generating links between these two types of artifacts. Whoever would like to evaluate a technique for generating links between design documents and code will have to develop a new example.

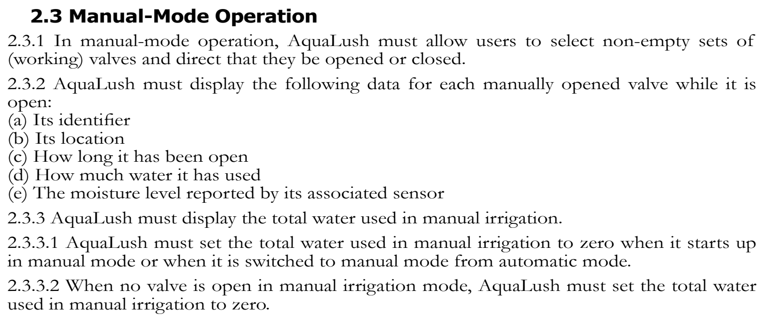
Comparing traceability techniques is also an important challenge. The quality of generated links depends considerably on the used example. In [16], the authors obtained 90% recall for a precision of 17% when using the vector space model on the EasyClinic project, while they got 47% recall for the same precision value when applying the same technique on the eTour project. Therefore, the effectiveness of link generation techniques can only be compared when they are applied on the same case.

Consequently, there is an urgent need for developing a case that is both complete and publicly available to the community. The case should include all artifacts that are produced during the development of a software system, as well as the traceability links among these artifacts. Such a case can then serve as a benchmark for traceability. This can be done by defining the tasks that should be performed on the case and the measures that are used to evaluate the effectiveness of the method used to perform the tasks. Constructing a benchmark for traceability will not only solve the problems mentioned above, it will also support advancing the research in the field of traceability because it facilitates the replication of experiments and the comparison of results to each other [17] [19].

In the next section, we present the basic components of a traceability benchmark that includes many artifacts produced during the development of a software system and provides end-to-end traceability links among these artifacts.



(a) Statechart



2.3 Manual-Mode Operation

2.3.1 In manual-mode operation, AquaLush must allow users to select non-empty sets of (working) valves and direct that they be opened or closed.

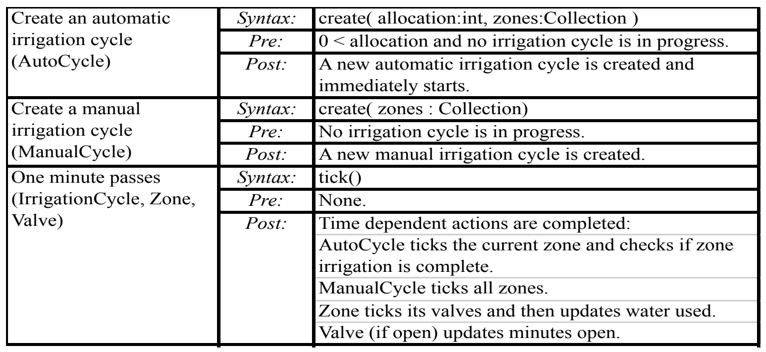
2.3.2 AquaLush must display the following data for each manually opened valve while it is open:
(a) Its identifier
(b) Its location
(c) How long it has been open
(d) How much water it has used
(e) The moisture level reported by its associated sensor

2.3.3 AquaLush must display the total water used in manual irrigation.

2.3.3.1 AquaLush must set the total water used in manual irrigation to zero when it starts up in manual mode or when it is switched to manual mode from automatic mode.

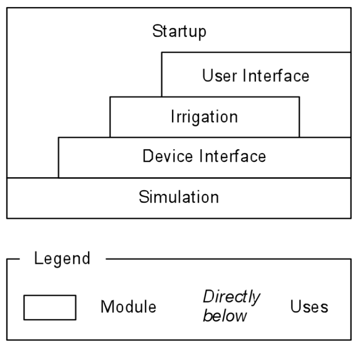
2.3.3.2 When no valve is open in manual irrigation mode, AquaLush must set the total water used in manual irrigation to zero.

(b) Functional requirements



Create an automatic irrigation cycle (AutoCycle)	*Syntax:*	create(allocation:int, zones:Collection)
	Pre:	0 < allocation and no irrigation cycle is in progress.
	Post:	A new automatic irrigation cycle is created and immediately starts.
Create a manual irrigation cycle (ManualCycle)	*Syntax:*	create(zones : Collection)
	Pre:	No irrigation cycle is in progress.
	Post:	A new manual irrigation cycle is created.
One minute passes (IrrigationCycle, Zone, Valve)	*Syntax:*	tick()
	Pre:	None.
	Post:	Time dependent actions are completed:
		AutoCycle ticks the current zone and checks if zone irrigation is complete.
		ManualCycle ticks all zones.
		Zone ticks its valves and then updates water used.
		Valve (if open) updates minutes open.

(c) Pre and Post conditions



(d) AquaLush layers

Figure 1: Examples of AquaLush Artifacts

3. THE BENCHMARK

3.1 Is it The Right Time for it?

Sim et al. [17] mention two conditions that need to be satisfied before making attempts for constructing a benchmark for traceability. First, the field of research needs to be mature enough so that the benchmark does not hold back the progress in the community. Second, there must be a willingness for collaboration within the community because this facilitates the acceptance of the benchmark and its use.

Attempts to compare traceability approaches indicates that the field is mature enough for the development of a benchmark. In the last years, there have been several studies comparing automated link generation approaches based on different information retrieval techniques [12] [1] [15] [16]. Recently, researchers began to use data and cases developed by other laboratories to evaluate their traceability techniques (e.g., in [15], the authors evaluated their approach using the data that were developed by other researchers in [2]). This facilitates the comparison of these techniques to a great extent.

Many events for traceability illustrate the willingness for collaboration within the community (e.g., the workshop on Traceability in Emerging Forms of Software Engineering and the International Symposium on Grand Challenges in Traceability).

An exploration of the state of research in the field of traceability shows that the discipline is beyond the cited preconditions. In fact, the community is already aware that a benchmark for traceability is needed. Indeed, this need has been explicitly mentioned and discussed by a number of researchers in the field [4] [7]. The traceability community is also engaged in establishing a research infrastructure that facilitates the running of traceability experiments [5]. This infrastructure is expected to include a number of traceability benchmarks.

3.2 Creating the Benchmark

3.2.1 AquaLush

We constructed the data of our benchmark based on an existing case study: AquaLush [10]. AquaLush is an irrigation system that uses soil moisture sensors to control the irrigation of the soil. The AquaLush case study is an illustrative example from a book about software design. We chose AquaLush because it includes a rich set of documents that covers several artifacts produced during different development stages. The AquaLush artifacts that we included in our benchmark are the user-level requirements, the use case model, the software requirements specification, the software architecture document, the detailed design document and the source code. The documents are written in natural language and the source code is written in Java. The documents contain a number of diagrams (e.g. class diagrams or statecharts), tables and GUI screenshots. Figure 1 presents some examples of the AquaLush artifacts.

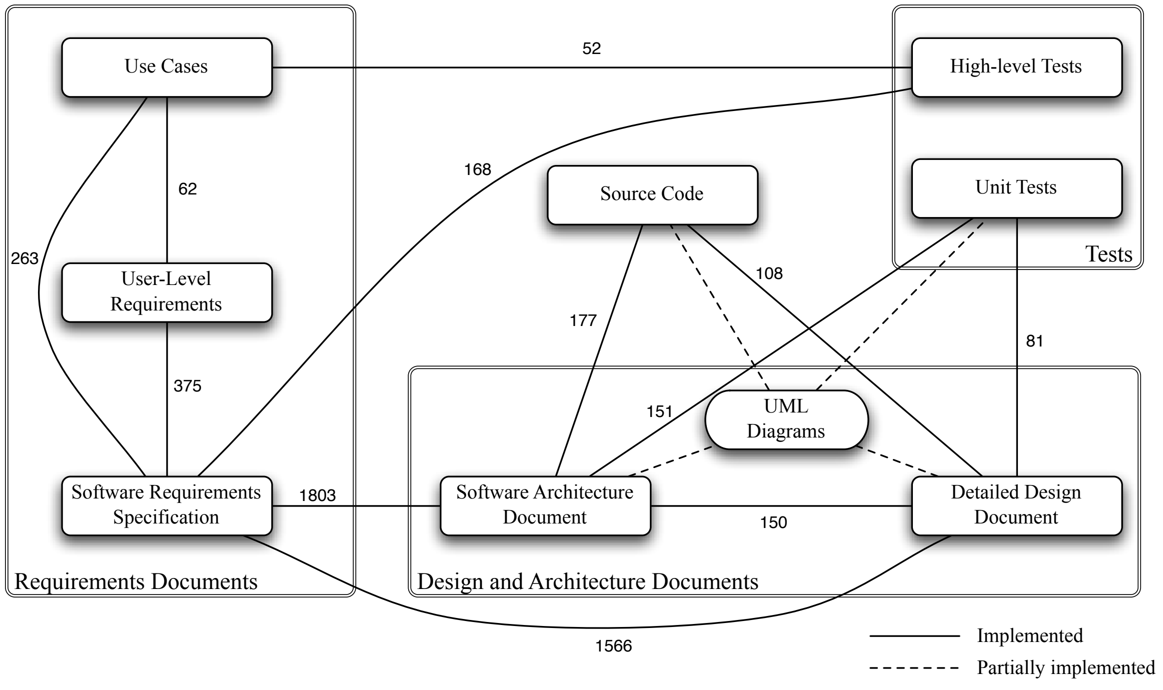


Figure 2: The data set and answer set of the benchmark

The AquaLush project was developed for pedagogical purposes, therefore it includes documents that reflect the practices lectured in software engineering courses. We found a few inconsistencies among some of the artifacts (e.g. methods mentioned in the design document but not implemented in the code) and a number of bugs in the implementation, but these are typical problems that are likely to be found in any software project. We did not fix these problems as we did not want to alter any of the project data.

3.2.2 Benchmark Components

The design of our benchmark was inspired by the work of Dekhtyar et al. [7] who decompose a benchmark for traceability into five components: data set, tasks, answer set, measures and software/data format. We detail each of these components in the following paragraphs.

Data Set. Our goal was to develop a complete data set that covers the main artifacts produced during the development of a software system. We developed the data set based on the AquaLush case study. We took the existing AquaLush documents, which already cover artifacts from the requirements, design and implementation stages and added tests in order to cover the testing stage. We developed two types of tests: unit tests and high-level tests. Both kinds of tests are automated and implemented using JUnit[1]. We used the following testing techniques to develop the unit tests [3]: equivalence partitioning, boundary-value analysis and branch coverage. We also considered the Liskov Substitution Principle [14] which states that if a type B is a subtype of another type A in an object-oriented program, then it should be possible to replace objects of type A with objects of type B each time an object of type A is used without having to change the rest of the program. The unit tests cover all classes except those in the user interface layer and in the start-up layer. The high-level tests have been developed according to the use cases available in the requirements documents. They cover both the basic and alternative flows of the use cases. Table 1 lists the documents in the data set.

Table 1: Size estimation of the benchmark data set

Document	Size	
User-level requirements	49 statements	599 words
Use Cases (one use case includes several extensions)	8 use cases	2075 words
Software requirements specification	372 statements	7370 words
Software architecture document	109 statements	5497 words
Detailed design document	64 statements	3803 words
Diagrams	23 diagrams	
Source code	75 classes	11 KLOC
Tests	93 classes	15 KLOC

To manage the artifacts, we used two tools: DOORS[2] and Rhapsody[3]. Textual documents, tables and screenshots were entered in DOORS. Each message is entered as one element in DOORS. If a title (or a subtitle) has only one message below it then the title and the message are merged in one

[1]http://www.junit.org/

[2]http://www.telelogic.com/products/doors/

[3]http://www-01.ibm.com/software/awdtools/rhapsody/

element. In the opposite case, the title is considered as one element. The UML diagrams were extracted from the architecture and design documents and were entered in Rhapsody.

Tasks. There are various traceability related tasks that can be performed using the AquaLush data set, including the use of traceability links among the artifacts for identifying the impact of a change or for propagating changes among the artifacts. In this work, we focus on defining traceability links.

We intend to evaluate and compare the effectiveness of methods and tools in generating complete and correct traceability links among different types of artifacts. We are especially interested in the generation of end-to-end vertical traceability linking (1) the requirements to the design and architecture documents and(2) the architecture documents to the source code and tests (Figure 2). We also consider the generation of links between high-level tests and requirements because this high-level tests are meant to check that the requirements are satisfied and are therefore related to them. The resulting links allow tracing any artifact to any other either directly or by going through an intermediate artifact. For example, it is possible to trace elements from the requirements specification to elements in the code by first finding elements in the software architecture document related to the requirements and the finding elements related to these architectural elements. The proposed tasks do not depend on any specific traceability tool or technique and they can be achieved manually or automatically.

Answer Set. The answer set (or ground truth) is the set of correct traceability links that relate the AquaLush artifacts to each other. We defined these links manually among the main AquaLush artifacts as presented in Figure 2. The arcs in Figure 2 are labeled with the numbers of traceability links that we have defined. We used DOORS to define links among all the textual artifacts, tables and pictures. For UML diagrams, we used Rhapsody, which allows us to link internal elements of the diagrams (e.g. link a class or a method). We also used DOORS and Rhapsody to define links pointing to the source code. Links to the source code point either to a class or to a package.

We defined our links according to the following rule: an element B is related to another element A if B is derived from A or if B gives additional and useful information about A.

There are a few elements (methods, constructors and classes) that are mentioned in the architecture and design documents but which are not implemented in the code. As we did not want to modify any of the existing AquaLush artifacts, we did not link these elements to the code.

Measures. To evaluate the effectiveness and efficiency of a traceability techniques, we use three measures: precision, recall and time. Precision and recall (see Section 2.1) are frequently used to evaluate link generation techniques. The time measure is meant to quantify the time needed by a given technique for generating the traceability links.

Software/Data Format. The format of the benchmark data should be easy to use and also be independent of any specific tool. As DOORS and Rhapsody documents do not satisfy this property, we exported all textual data into HTML documents and all UML diagrams into XMI documents. There are two advantages for using HTML. First, it allows browsing the documents easily and navigating among related artifacts using simple clicks. Second, HTML documents are well structured and any specific information can easily be extracted from them. For example, it is possible to extract the traceability links and create a traceability matrix out of them using a small piece of code. We used XMI for UML diagrams because it is a standard format for exchanging UML diagrams. UML diagrams are available as images too. We also provide the Doors and Rhapsody documents for those who prefer to use these tools.

3.3 Desiderata for a Benchmark

To be successful, a benchmark should satisfy some properties. Sim et al. [17] identified seven desiderata for a benchmark in the field of software engineering: accessibility, affordability, clarity, relevance, solvability, portability and scalability. Dekhtyar et al. [7] identified five additional requirements for a traceability benchmark: support for traceability in multiple software engineering fields, independence of methodology, ground truth, accuracy testing and scalability testing. In this section, we discuss to which extent our benchmark meets these characteristics.

Accessibility. The benchmark data need to be easy to obtain and easy to use. To satisfy this property, we made the benchmark public[4]. The data includes all the AquaLush artifacts mentioned in the previous section and the traceability links relating the artifacts to each other. Anyone can get the data, use it and eventually extend it to meet other requirements.

Relevance. The benchmark tasks should be representative of the typical traceability operations that are performed in real life. This condition is satisfied because the AquaLush artifacts are representative of the main artifacts produced during the development of a software system. The task of generating traceability links is also a typical operation that has been addressed by many researchers in the traceability field.

Clarity. AquaLush is an illustrative example from a book about software design. Therefore, it easy to understand and self-contained. The benchmark task (generate traceability links among various artifacts) is a classical and clear task that has already been performed by many researchers. The benchmark measures (precision, recall and time) are also simple and classical measures that have been used by researchers in the field.

[4]The benchmark is published at:
http://www.ifi.uzh.ch/rerg/research/aqualush/

Affordability. The benchmark must not be difficult or expensive to run, because otherwise researchers will not use it. The AquaLush project is not large and the link generation task is clear, therefore the benchmark is easy to use. Before using the benchmark, people may need to modify the format of the artifacts and the traceability links in such a way that they can be used by the tools or approaches they are using. This task can easily be automated in most of the cases.

Solvability. It should be possible to produce a good solution for the benchmark task. Our traceability benchmark is solvable and the solution, which is the set of traceability links among the different artifacts, is provided within the benchmark.

Portability. The benchmark should be portable to different tools and techniques. Both the AquaLush artifacts and the traceability links among the artifacts do not depend on any specific tool or technique. The data is available in a standard format (HTML and XMI). Therfore, it can be used to evaluate any tracing tool or technique.

Scalability. Scalability might be the major limitation of this benchmark. While the AquaLush project includes many types of artifacts, it is not a large project and thus it is not representative of large systems. To improve the scalability of the benchmark, AquaLush may be extended with new features. This extension is left for future work.

Support of multiple SE fields. The benchmark should be rich enough to support various tasks related to tracing. Currently, we only consider the generation of links among various types of artifacts. However, as our benchmark includes a complete data set with end-to-end traceability, it can be used for evaluating tasks from other software engineering processes such as verification and validation or maintenance and evolution. For example, analyzing the impact of changing one requirement on the rest of the artifacts is a task in software maintenance that can be evaluated with our benchmark.

Independence of methodology. The benchmark should not depend on any specific tracing tool or technique. This requirement is satisfied by our benchmark. In fact, all tracing methods, whether manual, semi-automated or automated, can be used to solve the task of defining traceability links among the AquaLush artifacts.

Ground truth. The true answer for each of the benchmark tasks should be provided. In our case, the true answer is the set of traceability links that relate the AquaLush artifacts to each other. We have defined these traceability links and we provide them with the benchmark.

Accuracy testing. The benchmark should allow evaluating the accuracy of tracing techniques. In our benchmark, we assess the accuracy through the precision and recall measures.

4. PROOF OF CONCEPT

4.1 Experiment

As a first application of our benchmark, we used an information retrieval tool to generate traceability links among some artifacts in the benchmark and then compared the generated links with the ground truth (that is, the traceability links we defined manually). We then compared the results we obtained in term of precision and recall to the results published by other researchers who used the same traceability technique on other case studies. We considered two tasks: generating links from the user-level requirements (ULR) to the software requirements specification (SRS) and generating links from the software architecture document (SArch) to the code.

The goal of this experiment is to answer the following questions:

Q1: Are the results (precision/recall) obtained when generating traceability links for the AquaLush project similar to those obtained by other researchers using the same techniques on other cases? What conclusion can we derive from these results concerning the relevance of our benchmark?

Q2: Do we get similar results when we use the same traceability tool to extract links (1) between two documents written in natural language and (2) between a document written in natural language and source code?

To run our experiment we used a research tool for generating traceability links: RETRO (REquirements TRacing On target) [13]. RETRO provides a number of IR methods that can be used for recovering links. It also allows filtering candidate links by specifying a threshold value for the traces that should be considered: if the threshold is 0.15, then only the links having relevance greater than 0.15 are kept. Analysts can enter feedback into RETRO to improve the quality of generated links, but in our experiment we did not use the feedback feature.

RETRO takes two lists of textual files as input: the high-level documents and the low-level documents. Therefore, we had to split the documents (ULR, SRS and SArch) into small subdocuments, where each subdocument contains one element that should be traced. For the source code, we considered each class as a single document. As RETRO only considers textual documents, we removed all pictures and diagrams from the document. We also extracted the content of all tables into text files.

We generated links using the default tracing method in RETRO, which is the vector space retrieval with tf-idf (term frequency - inverse document frequency) term weighting. Then we filtered links with different threshold values and observed the effect of the filtering on precision and recall.

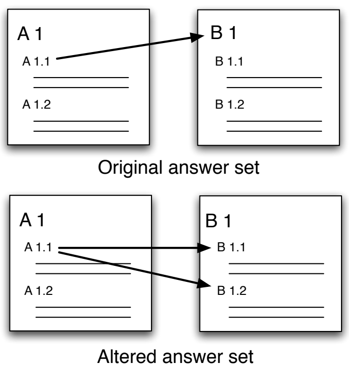


Figure 3: Splitting traceability links through the hierarchy

The resulting links were compared with the ground truth of our benchmark automatically.

When evaluating the links generated between the architecture document and the code, we were forced to alter our manual traceability links (i.e., the answer set). The reason is that RETRO does not consider the hierarchical structure of documents; so it only generates links pointing to classes. However, in AquaLush, 32 of our manual links point to whole packages instead of individual classes. For example, we linked general statements about the GUI to the UI package. To make the comparison of links possible, we split the links pointing to a package into several links pointing to each of the classes within the package as illustrated in Figure 3. We call the experiment we ran with these links *AquaLush (+)*. We also ran a second experiment (*AquaLush (-)*) where we neglected all the links pointing to packages.

Managing the titles and subtitles was also a challenge. Titles contain relevant keywords related to the statements under them. But existing traceability tools do not take the structure and titles of documents into consideration. Therefore, in this experiment, we deleted all the elements that contain a title only. We ran an additional experiment, which we do not report in this paper, while keeping the titles in the software architecture document. The results were very similar to those obtained from the experiment with no titles.

4.2 Results

In this section, we compare the precision and recall we obtained with AquaLush to those published in [16] and in [18]. We also compare the results obtained in the two link generation tasks.

In [18], Sundaram et al. used RETRO to generate traceability links between high and low-level requirements for two data sets: MODIS and CM-1. MODIS contains 19 high-level requirements and 49 low-level requirements. There are 41 links between high-level and low-level requirements. CM-1 has 220 high-level requirements, 235 low-level requirements and there are 361 links among these requirements.

We took the results they obtained using tf-idf and no analyst feedback and compared them to the precision and recall we obtained when generating links from the user-level requirements to the software requirements specification of AquaLush. As we used the same IR method, the same tool and similar types of artifacts as in [18], we expected to obtain results that are comparable to each other. The results are reported in Figure 4. The precision and recall obtained for CM-1 and for AquaLush are very similar.

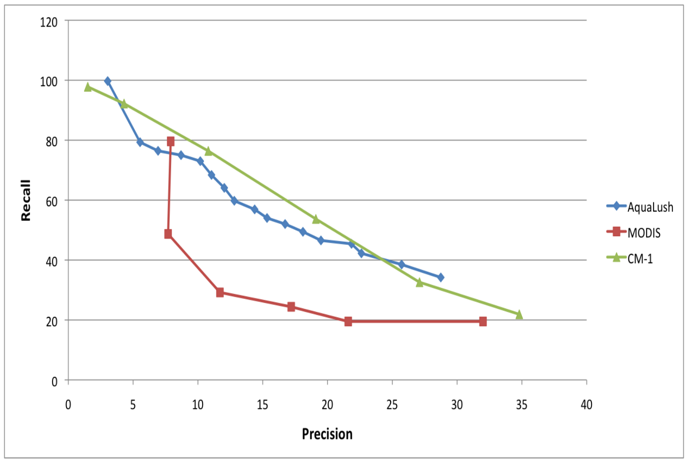


Figure 4: Precision and Recall for traceability links generated from the user-level requirements (ULR) to the software requirements specification (SRS)

Comparing the results obtained from generating links between source code and architecture documents with results obtained by other researchers was more challenging. In fact we could not find results for these types of documents, but we found results for using IR methods to get links between code and other types of artifacts.

In [16], Oliveto et al. compared different IR techniques for generating links between use cases and source code for two data sets: EasyClinic and eTour. EasyClinic has 30 use cases, 47 classes and 93 correct links while eTour has 58 use cases, 116 classes and 336 correct links. Among the various results presented in the paper, we considered those obtained with the Vector Space Model. These results are reported in Figure 5.

The recall obtained with both AquaLush experiments is lower than the one obtained with EasyClinic and eTour for precision values that are over 17% (no recall values were reported in [16] for lower precision). However, the difference is not huge for the case of AquaLush (-) and eTour.

In Figure 6, we report the results we obtained for both link generation tasks. Globally, the results we obtained when generating links from ULR (user-level requirements) to the SRS (software requirements specification) are comparable to those obtained when generating links between SArch (software architecture document) and code.

Answering Q1. The results we obtained when generating traceability links between ULR and SRS are very similar to those obtained by Sundaram et al. [18] when using the

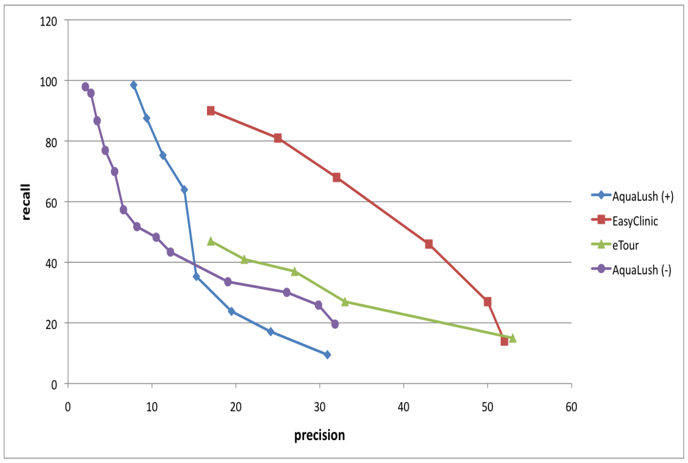


Figure 5: Precision and Recall for traceability links generated from the software architecture document (SArch) to the source code

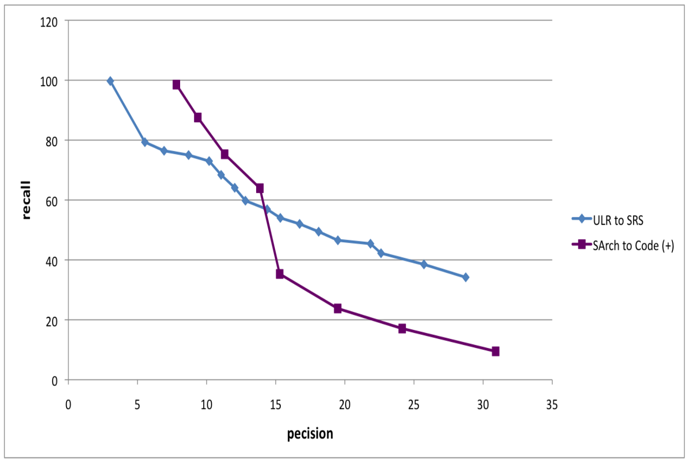


Figure 6: Precision and Recall for traceability links generated from the user-level requirements (ULR) to the software requirements specification (SRS) and from the software architecture document (SArch) to the source code

same tool on similar types of artifacts from the project CM-1. This similarity is a positive indicator for the fitness of our benchmark for evaluating traceability generation techniques, because the developers of the RETRO tool could have obtained similar results if they used our benchmark for validation.

The precision we obtained for generating links between SArch and code are not as good as those obtained by Oliveto et al. in [16]. There are various explanations for the lower precision we obtained in this experiment. First, the use of different cases with different types of documents (use cases in [16] vs. architecture document in our experiment) is likely to give different results. Second, we did not use the same tool (no tool was mentioned in [16]), so the technique they used for generating links may be different from ours. Moreover, they do not mention which term weighting was used, so it is possible that it is not the one we used (tf-idf). The text normalization may also be different, as code and textual documents can be normalized in different ways. Finally, our results are probably affected by the modifications made to the manual links (see Section 4.1) in order to make the comparison possible.

Answering Q2. RETRO performs similarly when generating traces between two documents written in natural language and when generating traces between a document in natural language and source code. While we can make no reliable conclusions about the performance of RETRO based on a single experiment, the experiment suggests that RETRO could achieve comparable results when used for tracing documents written in natural language and when used for source code.

4.3 Lessons Learned

In this section, we present a number of problems that we faced during the development of our benchmark and how we mitigated them.

Rules for defining links. When defining the traceability links, deciding whether or not two elements are related was sometimes difficult. In the literature, we could not find guidelines about how to define traceability links within a project. Therefore, we defined our own guidelines: we consider two elements A and B to be related only if B is derived from A or if B gives additional and useful information about A. There may be other ways to define traceability links among artifacts depending on the purpose behind their implementation. We encourage traceability researchers to explore what kinds of links are most useful and how these links should be defined.

Support for hierarchical documents. Current techniques for generating traceability links do not support the hierarchical structure of documents. For example, they do not allow linking one element to a whole section or subsection in a document. They also do not take titles and subtitles into consideration. In the answer set of the benchmark, there are links pointing from elements in the architecture document to packages in the code. It was not possible to compare these links directly to those generated by RETRO because the links generated by RETRO only point to classes. To overcome this problem, we split the links pointing to a certain package into several links pointing to every class within the package (see section 4.1).
Supporting the hierarchical structure of documents is an interesting direction for future research.

5. THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our benchmark.

Construct Validity. To evaluate the effectiveness of traceability approaches, we use two classical measures that have been extensively used to evaluate IR based approaches: precision and recall. However, these measures may not cover all the strengths and weaknesses of a tracing method or tool. Therefore, it would be interesting to consider additional measures that assess other dimensions of the traceability tool or method, such as the usefulness of the generated links for program comprehension.

Internal validity. The AquaLush artifacts used for creating the benchmark were developed by a third party who was not aiming at using them for traceability purposes. This reduces the risk of having artifacts that are tailored for facilitating the generation of traceability links among them or artifacts that are adapted to some special traceability tool or technique.

External validity. AquaLush is a relatively small project which has high-quality artifacts. It is therefore not representative of large real-world software projects having incomplete and low-quality artifacts.

The evaluation of the effectiveness of a tool or technique depends on the example used for the evaluation. In most cases, we get different results when we apply the same tool or technique on different case studies. Therefore, a single case study is not enough to draw generalizable conclusions. A good benchmark should include documents that are representative of different types of projects. Thus, it is important to expand our benchmark in the future by adding other types of projects.

Other limitations. A major limitation of our benchmark is that the traceability links we defined (i.e., the ground truth) have not been validated other than through a careful inspection by us. To strengthen the validity of the traceability links, we intend to ask the original developer of AquaLush to evaluate the adequacy of our links. Currently, our answer set also lacks direct traceability links from requirements to the source code. We only trace from requirements to code via the software architecture (see Figure 2). We might add the direct requirements-to-code links in a future version of our benchmark.

6. NEXT STEPS

We are currently finalizing the answer set by completing the traceability links among the artifacts as presented in Figure 2. For future work, we intend (1) to use the benchmark to compare different traceability links and tools and (2) to extend the benchmark to cover other traceability related tasks.

Using the benchmark. We will use our benchmark to compare the effectiveness of different traceability link generation techniques and tools. The goal of the experiment will be to find which tool or technique is most efficient for each type of document. Each technique/tool will be used to generate links among the different AquaLush documents and the results will be compared to each other.

Extending the Benchmark. We intend to support the following three tasks in the future: Analyzing the *impact of change*, tracing *bug reports* and *updating* traceability links. The goal of the impact analysis task is to identify all the artifacts affected by a given change. To cover this task, we will define a number of changes (like bug fixes, changes in the external behavior of the system or changes in the design) and identify all the artifacts affected by the change. We will also define measures that estimate the time needed for performing the analysis and the correctness of the obtained results.

The bug-tracing task is about generating traceability links between bug reports and source code. For this task, we will extend AquaLush with some bug reports and traceability links from the bug reports to the source code.

Updating traceability links is a challenging task that can be evaluated using our benchmark. We will create a second release of the AquaLush artifacts and define traceability links among them. The task will then be to identify all links affected by the changes in the artefacts and update these links.

7. RELATED WORK

To the best of our knowledge, none of the existing case studies used for evaluating traceability technique cover all the artifacts that are covered by our benchmark and provide end-to-end traceability among the artifacts.

Hayes at al. [12] use data sets obtained from two NASA projects: CM-1 and MODIS. These data sets only cover high-level and low-level requirements. In [16], Oliveto et al. used data sets from the eTour and the EasyClinic projects. eTour only includes use cases and code classes. EasyClinic includes uses cases, a textual representation of interaction diagrams, source code and test cases. However, it does not contain a software requirements specification, an architectural document nor a design document.

iTrust[5], a medical application, has also been used as a traceability case. It includes a requirements specification, source code and a testing plan. Again, we did not find any design or architecture document.

Antoniol et al. [1] used two case studies (LEDA and Albergate) to evaluate their traceability recovery techniques. They only used the source code and the manual pages of LEDA (a C++ framework that is freely available). No other artifacts were mentioned.

Albergate is a software system that has been developed by students based on 16 functional requirements. The Albergate case study is not published. According to [1], Albergate includes all the documentation related to the entire software development process, but traceability links were only defined between the 16 requirements and the classes implementing them. Furthermore, the documentation of Albergate is written in Italian, which is problematic for many researchers.

[5]http://agile.csc.ncsu.edu/iTrust

8. CONCLUSION

In this paper, we presented a candidate benchmark for traceability based on a software for an irrigation system. Among other features, our benchmark includes all the typical artifacts that are produced during the development of a software system and it provides end-to-end traceability linking among these artifacts. The benchmark data, which are publicly available, do not depend on any specific traceability tool or technique. Therefore, researchers can use our benchmark to easily assess the effectiveness of their traceability methods. The benchmark is also convenient for the comparison of traceability methods.

The benchmark we are proposing is a first piece of an envisaged set of benchmarks that the traceability community needs. Our benchmark covers traceability tasks that require a rich set of artifacts with end-to-end traceability linking. We hope and envisage that other researchers will contribute further benchmarks featuring other characteristics such as very large data sets or traceability links evolving over time.

9. ACKNOWLEDGMENTS

Our work is partially funded by the Swiss National Science Foundation. We thank Professor Christopher Fox for providing us with the AquaLush artifacts. We also thank Dr. Jane Cleland-Huang for giving us useful information about existing traceability data sets.

10. REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[2] G. Antoniol, G. Canfora, A. de Lucia, and G. Casazza. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of the International Conference on Software Maintenance*, ICSM'00, pages 40–49, 2000.

[3] David Caspar. A benchmark for software traceability. Bachelor's thesis, University of Zurich, 2011.

[4] J. Cleland-Huang, A. Dekhtyar, J. Hayes, G. Antoniol, B. Berenbach, A. Eyged, S. Ferguson, J. Maletic, and A. Zisman. Grand challenges in traceability. Technical Report COET-GCT-06-01-0.9, Center of Excellence for Traceability, 2006.

[5] J. Cleland-Huang, A. Czauderna, A. Dekhtyar, O. Gotel, J.H. Hayes, E. Keenan, G. Leach, J. Maletic, D. Poshyvanyk, Y. Shin, A. Zisman, G. Antoniol, B. Berenbach, and P. Maeder. Grand challenges, benchmarks, and tracelab: Developing infrastructure for the software traceability research community. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE'11, 2011.

[6] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker. A machine learning approach for tracing regulatory codes to product specific requirements. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE'10, pages 155–164, 2010.

[7] A. Dekhtyar, J.H. Hayes, and G. Antoniol. Benchmarks for Traceability? In *Proceedings of Traceability in Emerging Forms of Software Engineering*, TEFSE'07, 2007.

[8] M. Eaddy, A.V. Aho, G. Antoniol, and Y.-G. Gueheneuc. CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Proceedings of the 16th International Conference on Program Comprehension*, ICPC'08, pages 53–62, 2008.

[9] A. Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, 29(2):116–132, 2003.

[10] C. Fox. *Introduction to Software Engineering Design: Processes, Principles and Patterns with UML2.* Addison Wesley, 2006.

[11] J.H. Hayes, A. Dekhtyar, and S.K. Sundaram. Improving after-the-fact tracing and mapping: supporting software quality predictions. *IEEE Software*, 22(6):30–37, 2005.

[12] J.H. Hayes, A. Dekhtyar, and S.K. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[13] J.H. Hayes, A. Dekhtyar, S.K. Sundaram, E.A. Holbrook, S. Vadlamudi, and A. April. REquirements TRacing On target (RETRO): improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering*, 3(3):193–202, 2007.

[14] B. Liskov and J. Wing. A new definition of the subtype relation. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP'93, pages 118–141, 1993.

[15] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE'03, pages 125–135, 2003.

[16] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *Proceedings of the 18th International Conference on Program Comprehension*, ICPC '10, pages 68–71, 2010.

[17] S. Elliott Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE'03, pages 74–83, 2003.

[18] S. K. Sundaram, J. H. Hayes, and A. Dekhtyar. Baselines in requirements tracing. In *Proceedings of the 2005 workshop on Predictor models in software engineering*, PROMISE'05, pages 1–6, 2005.

[19] W.F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32 –40, 1998.

[20] R. Watkins and M. Neal. Why and how of requirements tracing. *IEEE Software*, 11(4):104–106, 1994.

[21] A Zisman G Spanoudakis. *Software Traceability: A Roadmap.* Handbook of Software Engineering and Knowledge Engineering, World Scientific Publishing, 2005.