Benchmarks for Traceability?

Alex Dekhtyar
Department of Computer Science
University of Kentucky
Lexington, KY, USA
dekhtyar@cs.uky.edu

Jane Huffman Hayes
Department of Computer Science
University of Kentucky
Lexington, KY, USA
hayes@cs.uky.edu

Giuliano Antoniol
Department of Computer Engineering
École Polytechnique de Montréal
Montréal, Canada
giuliano.antoniol@polymtl.ca

ABSTRACT

This position paper discusses the need for and the organization of a traceability benchmark. We establish the basic principles of organization of such a benchmark. We then observe the nature of traceability tasks in three areas of Software Engineering: independent verification and validation, software maintenance, and reverse engineering. Based on this, we derive the desiderata for a traceability benchmark addressing the needs of all three areas.

1. INTRODUCTION

Tracing and traceability possess demonstrated importance in a number of disciplines such as software engineering. This is illustrated with a few common scenarios. Software maintainers rely on being able to recover the code components or methods that relate to a given bug report. A systems engineer needs to know the mapping between a legacy software system that is being enhanced and the developer's contract (doing the enhancing) to determine the impact of "mothballing" certain legacy code components. Unfortunately, the generation of, maintenance of, and/or assessment of traceability information (such as requirements traceability information) is largely manual, time-consuming, and error-prone.

Researchers have made great strides in inventing and validating methods and tools for addressing the traceability problem. Yet most researchers address specific traceability problems in the area of their expertise and/or of their funding organization. For example, Antoniol et al. addressed traceability recovery from code and user documentation to assist the software maintainer [1]. Hayes et al. sought to assist the independent verification and validation analyst in recovering traceability between requirements and design specifications [8]. Besides looking at different problem areas, researchers also examine different artifacts (code and user's documentation and requirements and design specs in the above example), use different approaches (semi-automated, fully automated; information retrieval methods, rule-based methods; etc.), and calculate different measures in order to evaluate their approaches. As a result, it is very difficult to: compare most of the studies performed by different research groups; adapt or apply ideas from the work of others; adapt or apply artifacts from the work of others; and adapt or apply measures from the work of others.

TEFSE'07 '07, March 2007, Lexington, KY

Other research fields have faced similar challenges and overcome them through the successful use of a benchmark. It is our position that a traceability benchmark would go a long way to alleviate the problems mentioned above. Specifically, a traceability benchmark is needed for:

- *Robust comparisons of different techniques for solving the same problem.* While there is an occasional apples-to-apples comparisons between traceability studies [1, 9], such work is just beginning to emerge.
- *View of the behavior of the same methods on different problems.* Standard IR techniques are applied with different success to various problems in Software Engineering. These problems, however, come from very different areas of Software Engineering, sprouting different researcher communities. There is very little communication across the problem boundaries, and very little analysis of the behavior of different approaches across the entire problem set in Software Engineering to which IR methods are applicable.
- *Comparison of emerging traceability methods and techniques with state-of-the-practice approaches.* As one of the stated goals of traceability research is technology transfer, apples-to-apples comparisons between new methods/approaches and techniques used in industry is needed to: (a) certify that new approaches provide "better" solutions, and (b) persuade industry to adopt them.

The recently published Grand Challenges in Traceability [4] specify *Measurement and Benchmarks* as one of the key challenge areas. In particular, the following Grand Challenges related to this area, have been outlined:

L-GC1 Define standard processes and related procedures for performing empirical studies during traceability research.

L-GC2 Build benchmarks for evaluating traceability methods and techniques.

L-GC3 Define measures for assessing the quality of individual and sets of traceability links.

L-GC4 Develop techniques to assess traceability methods and processes.

We believe that a well-designed, robust, feature-rich benchmark for testing methods, techniques, and processes for achieving traceability in the Software Engineering domain will go a long way toward addressing both issues above. It will establish the sorely needed "measuring stick" for testing uses of different traceability approaches within the same problem set. This paper begins the process of addressing specifically the challenge **L-GC2**.

In the rest of this paper, we describe a set of high-level (non-functional) requirements for a traceability benchmark (Section 2) and the desired organization of a traceability benchmark (Section 3). We then discuss the views of the benchmark from three different subfields of Software Engineering: Independent Verification & Validation, software maintenance, and reverse engineering (Section 4). For each area, we outline the origins and the nature of the tracing tasks, the artifacts used, and the means of determining "success." We conclude by merging this information together into an outline for a simple traceability benchmark.

2. REQUIREMENTS FOR A TRACEABILITY BENCHMARK

The requirements listed below outline our vision for an eventually emerging benchmark for traceability tasks. At this stage, we elected to concentrate on the benchmark for tracing/traceability in Software Engineering.

R1: Support for traceability in multiple fields of Software Engineering. The benchmark shall provide the ability to test tracing methods and procedures for tasks from different areas of Software Engineering.

R2: Independence of methodology. The benchmark shall be independent of a specific tracing methodology. That is, tracing tasks contained within the benchmarks shall be solvable by a wide range of methods and procedures: manual, semi-automated, automated.

R3: Ground truth. For each task within the benchmark, the benchmark shall establish the ground truth (i.e., true answer) against which to be compared.

R4: Accuracy testing. The benchmark shall provide the opportunity to test the accuracy of tracing techniques on its tasks.

R5: Scalability testing. The benchmark shall provide the opportunity to assess the scalability of tracing techniques of its tasks[1].

Discussion

The majority of the requirements selected follow common sense on what a benchmark should do and what problems it should address. As we have observed, tracing tasks are part of different Software Engineering processes: IV&V, reverse engineering, software maintenance, to name just a few. We would like for the traceability benchmark to be rich enough to contain tasks from (ideally) every area in Software Engineering where tracing occurs.

Moving on to the second requirement, we observe that we do not want to build the benchmark to *only* score the work of, for example, information retrieval methods. It should be possible to "solve" the tracing tasks contained within the benchmark in any way, including purely manual tracing. This makes sense because within the research community, we would like to have apples-to-apples comparison of the work of different automated methods. However, in order for these automated methods to transfer to practice, we have to demonstrate that they outperform current, often manual and labor-intensive, procedures. The requirement to have ground truth available for each tracing task can be viewed as a restriction that the benchmark creators should take upon themselves: do not add tasks to the benchmark for which you cannot, or are not willing to, provide the correct answer (or where correct answers do not exist). The fourth requirement states the main purpose of the benchmark: it has to allow for testing the accuracy of the methods. It is included in the list of requirements to provide symmetry with the fifth requirement, which is less obvious: in many circumstances, tracing techniques and procedures must be not only accurate but also expedient and scalable to large tracing tasks. The benchmark, in our view, must provide the opportunity to test the scalability of methods for such tasks.

[1] At least for the tasks where scalability is an issue.

3. ORGANIZATION OF TRACEABILITY BENCHMARKS

In light of the outlined requirements, we view a traceability benchmark as consisting of five main components: (i) data set, (ii) tasks, (iii) answer sets, (iv) measures, and (v) data representation formats/supplementary software.

We give a brief overview of each component below.

Data Set

The dataset of a traceability benchmark should be a collection of artifacts from a software project. To satisfy the first requirement, such a collection must be sufficiently rich and contain artifacts from different project stages and used in different software engineering processes. To satisfy the scalability requirement, the data set has to: (a) contain sufficiently large artifacts, and (b) provide the opportunity to scale artifact sizes up or down.

Perhaps the most difficult question is the origin of such a collection of data. Two possibilities exist: benchmark creators can take data from an already existing software project, or they can create the benchmark from scratch. The second option requires significant, perhaps, unrealistically so, effort on the part of the benchmark creators. The first option requires for there to be a software project that has a rich artifact collection which can become public.

Tasks

As stated earlier, tracing tasks originate from different underlying problems, which themselves come from different software engineering processes. To address the first requirement, the benchmark must complement the rich artifact collection with the rich collection of tracing tasks to be performed on the data. Individual researchers/research groups may concentrate on specific individual tasks, but the benchmark itself has to include enough tasks to address tracing needs from different areas of software engineering.

The nature of the tasks needs to be addressed here as well. Most traceability research now is concentrated on the tracing of textual artifacts (we consider code to be a textual artifact). However, tracing is needed for non-textual artifacts as well. At some stage of the benchmark development, non-textual artifacts must become part of the data set, and, consequently, the tracing tasks for non-textual requirements and tracing tasks between textual and non-textual requirements will have to enter the benchmark as well.

Another aspect of traceability tasks concerns the nature of recovered links. In some tracing tasks, there is only one type of link to be recovered, whereas for other tasks, it is important not just to recover the link, but to recover the type of the link correctly. Tracing tasks included in the benchmark must reflect these needs.

Answer Sets

Following the third requirement, the benchmark must include the answer set, i.e., the ground truth for each task. The reasons for this are straightforward and follow from our fourth requirement: there is a need to assess the accuracy of each method/technique/process used to solve a tracing task. This can only be achieved reliably if ground truth, i.e., the correct answers for each task, are known and can be compared against the answers returned by different methods.

As such, this becomes one of the most serious restrictions on building the benchmark. Only the tasks for which answer sets can be built (by benchmark creators) can be included. There are two aspects of this: (i) the task must actually have a correct answer set, and (ii) benchmark creators must be able to establish it and arrive at a consensus.

Measures

Requirements **R4** and **R5** demand inclusion of measures for assessing the accuracy and scalability of methods. Standard IR measures for accuracy, *precision*, *recall* (both macro- and micro- variants), *average expected precision*, *f-measure*, etc., certainly can, and will, be used as part of the benchmark. A standard measure for scalability is time.

In addition to accuracy and scalability, enshrined in our requirements are other quantities that can be measured. In particular, to facilitate technology transfer, it is important to develop a set of measures for effort.

Software/Data Formats

This, perhaps, is the only optional part of the benchmark. Yet, we observe that development of convenient data formats and inclusion of software that understands these formats in the benchmark can go a long way in both making the benchmark popular and in establishing standard means of encoding "real" artifacts.

4. THREE VIEWS OF A BENCHMARK

In this section, we provide brief accounts of three subdisciplines of Software Engineering in which traceability figures as a major factor: Independent Verification & Validation (IV&V), software maintenance, and reverse engineering. For each subdiscipline, we outline the data used, the tracing tasks performed, the measures used to determine the accuracy of the results, and other aspects that can influence formation of the benchmark.

4.1 Independent Verification & Validation

Independent Verification & Validation is a process that is performed in parallel with the development process when risk reduction is of the utmost importance. IV&V is often applied by independent, third party analysts when the software system being built is safety- or mission-critical. The IV&V analysts concentrate on ensuring that the right system is built (validation) and that it is built according to the specifications of each lifecycle phase (verification).

Toward that end, the IV&V analyst has a number of responsibilities, such as: ensuring that all requirements have been addressed by the software design, ensuring that the source code has correctly implemented the software design, ensuring that test cases have been developed to validate that the requirements have been implemented, and executing those test cases to ensure passing. As one can imagine, it is thus important to know the mapping of the requirements to the design, of the design to the code, and so on. This mapping is referred to as the requirements traceability matrix (RTM) and should be delivered by the system developer. However, the RTM is often not delivered, or is not developed to the proper level of detail, or is not kept up to date. Thus, it often falls on the IV&V agent to build such an RTM. Even if an RTM is delivered by the developer, the IV&V agent has the job of assessing the correctness of the RTM. So it is clear that traceability is an important activity in IV&V.

Let us look closer at the typical data/artifacts that an IV&V analyst will utilize in performing traceability or traceability assessment. It is common for requirements specifications to be delivered as textual artifacts with some embedded graphics. The requirements are often expressed in free form narrative text. The design specifications are also often textual, but may have more graphical information, such as in UML diagrams. Source code will be available in digital form. Test cases will often be free text, but may contain embedded "pointers" to requirements, design elements, or source code methods. The size and number of specifications varies greatly based on the project. Some projects may have one system level specification, a half dozen software requirements specifications, a dozen design documents, and millions of lines of source code. Smaller projects may have only one software requirements specification, one or two design specifications, and merely thousands of lines of code.

The tracing tasks that the IV&V analyst must perform are largely driven by the artifacts that are delivered by the developer. For example, if the developer delivers an RTM, the agent's task is to assess that RTM rather than build it. If no RTM is delivered, the agent must build an RTM for the various artifacts delivered. If a developer delivers a full host of artifacts (requirements specifications, design specifications, source code, test plan, test cases, user's manual, administrator's manual, etc.), the IV&V agent has a host of tasks to perform, such as:

- trace requirements specification to the design specification;
- trace design specification to source code (files, methods, classes);
- trace source code to test cases;
- trace requirements to test plan;
- trace requirements to user's manual; and
- trace design specification to test plan and/or test cases (depends on level of detail in the test plan).

In IV&V, it is of utmost importance to construct the correct RTM. To ensure that this has happened, the agent may manually validate that certain links in the RTM are correct, and will manually ensure that all high level elements have been satisfied by their children (so they may ensure that all requirements have been satisfied by what is listed as their children in the RTM). Since we want to automate as much of the tracing tasks as possible, we need measures to help us evaluate the quality of such automated methods (whether we are using them to help build RTMs and/or assess RTMs). Specifically, we need to make sure that the automated methods are accurate. We want to make sure that they find all the links that exist (recall) and do not retrieve things that are not links (precision). We want to ensure that the techniques do not require as much time on the part of the analyst as manual techniques (effort). We want to ensure that the techniques work well on large projects as well as small ones (scalability). It is also important to allow the IV&V analyst to have the "final say" on what is a link or on whether or not a parent element has been satisfied by its children. So the techniques need to incorporate the input of the human analyst (feedback).

With this in mind, let us next examine how a software maintainer might use traceability in her everyday work.

4.2 Software Maintenance

Software Maintenance is the set of activities and processes carried out on deployed software to correct defects and deficiencies, to add new functionality, to improve or enhance existing features, or to prevent defects and, in general, quality degradation. Software maintenance is highly human-intensive and risky, since changes in any software system of a realistic size risk degrading software quality and may produce unwanted or unexpected side effects. Successful software systems operate for decades, and often outlive the hardware and operational environments for which they were originally conceived, designed, and developed. As a software system is enhanced, modified, and adapted to new user needs, the code becomes increasingly complex, often drifting away from its original design. Furthermore, very often source code evolves, but documentation is not updated; maintaining consistency and traceability information between software artifacts is a costly and time-consuming activity, frequently neglected due to the pressure to reduce costs, to reduce time to market, or to move on to the next software change. Thus, the system itself is often the only reliable source of information and source code browsing is the most commonly performed activity during maintenance because obsolete or missing documentation forces maintainers to rely on source code only. Unfortunately, source code browsing becomes very time- and resource-consuming as the size and complexity of programs increase. An alternative to source code browsing is automatic or semi-automatic design recovery. Central to this is the recovery of "higher-level abstractions beyond those obtained by examining a program itself" [3].

For example, corrective maintenance tasks are activated by discovered defects; maintainers receive short, informal descriptions of problems and are asked to change the software thus producing a new, hopefully, defect-free, release. They have to build mappings between defect descriptions, domain and application concepts and, ultimately, associate domain and application concepts with code fragments and vice-versa. It is worth noting that the process of concept location and abstraction building is preliminary to any software change and, thus, it is not limited to corrective maintenance. Adding new features or enhancing existing ones must be done in such a way as to avoid unwanted side effects. This, in turn, requires the building of a mental model of features and feature interactions. In performing this mapping process, maintainers rely on available documentation; typical tasks include:

- trace the problem description to domain and application concepts;
- trace domain and application concepts to requirements and/or features; and
- trace requirements and/or features to design and/or code regions.

Here the term feature is defined as a requirement, documented or not, of a program that a user can exercise and which produces an observable behavior. With this in mind, it is clear that to accomplish any maintenance task, traceability links between code and other sources of information are crucial and preliminary to any actual software change.

When recovering traceability information, for example, locating features or tracing requirements to low level artifacts, it is of utmost importance that maintainers are not overloaded by information. At the same time, some of the methods, functions, classes, and, in general, code regions relevant to the problem under study must rank high in the recovered traceability links. Precision and recall, although relevant, do not fully characterize the maintainer's point-of-view. Browsing hundreds or thousands of traceability links will just not help and will make traceability recovery not very useful. This is especially true when maintenance has to be performed in large software systems, applications of millions of lines of code. Reducing code browsing to a few percent of the entire system will be considered a success from a researcher's point-of-view, but it will be considered a failure from the maintainer's perspective. In conclusion, we believe that maintenance tasks require one to strive for a balance between precision and recall in order to provide developers with as fewer traceability links as possible while ensuring that important links are not forgotten (and thus to ease developers' understanding). Indeed, when maintenance is considered, the success of tracing tasks is not determined solely by precision and recall, but also by the relative position of correct traceability information in the retrieved set.

4.3 Reverse Engineering

As underlined in the previous subsection, very often, change and evolution activities focus only on fixing defects. While the source code is evolving, the architecture, design, and documentation are not updated. Reverse engineering and maintenance activities are tightly related; in the absence of reliable documentation, reverse engineering practices aim at recovery of high level abstractions supporting program understanding, concept location, and feature location. Reverse engineering dates back to the late 1980s and the seminal work of Chikofsky and Cross who, in 1990, introduced a taxonomy for reverse engineering and design recovery [3]. They defined reverse engineering to be "analyzing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information."

Several successful technologies such as program slicing [11], feature location [5, 2], concept location [12], and architecture reverse engineering [6, 10] have been developed. These, and other techniques, recover different abstractions. We note that manual work is required to establish traceability links between and within extracted representations for most every technique. Great savings can be obtained if automatic or semi-automatic approaches are used to assist in the recovering and assessing of those traceability links.

A second challenging issue is related to traceability of behavioral information. Traceability links must be established not only between artifacts describing the software structure, but also between artifacts detailing the interaction of components and the component semantics. We hold that traceability must be extended to documents such as UML sequence, activity, and state diagrams. This, in turn, requires that as-is information be recovered or validated on an existing system.

Traceability links can be subsequently exploited in software change and evolution to evaluate the impact of changes, to define the priority and schedule the order of changes, to ensure that changes didn't affect expected behavior, and to support automatic test data generation.

In performing these reverse engineering tasks, programmers rely on static and dynamic information. Static information is extracted form source code while dynamic data are collected by executing the software under different scenarios. Typical tasks include:

- extract abstractions and logical views and build traceability links to domain and application concepts;
- trace outdated representations to as-is abstractions and views;
- establish and validate traceability links between and within extracted views;

- establish and validate traceability links semantics; and
- establish and validate traceability links between extracted views and requirements.

In many reverse engineering tasks, completeness and accuracy of extracted representations have been considered a key success factor. Being that the activity is rarely performed, the cost of discarding a few false positives is not considered a major issue. Once views and abstractions have been validated, traceability information has to be recovered and validated. Reverse engineering is not a stand-alone activity. Rather, it aims at supporting other tasks, mainly software change and evolution. In other words, costs and accuracy must be compatible with the actual subsequent in-the-field use of extracted information. For example, if the goal is to support IV&V, correctness will be of utmost importance; if the goal is to help in program understanding and feature location, the effort required to locate a correct chunk of information into retrieved traceability links is more important than correctness or completeness. So, the success criteria inherently depends on the subsequent tasks. Furthermore, any benchmark will be required to be flexible and extensible to ease the task of incorporating new views, new abstractions, or add new link semantics.

5. TOWARD THE BENCHMARK

In Section 3, we have described the five components of an ideal, universal, traceability benchmark. In Section 4, we briefly described the specific needs of three subareas of Software Engineering w.r.t. traceability. We recognize that the list of traceability needs and tasks presented above is far from complete – for example, problems and tasks associated with traceability between different versions of the same artifact have not been raised, while they too comprise an important collection of traceability tasks.

At the same time, we observe that any attempt to begin the construction of a traceability benchmark must start somewhere. Also, the attempt must have as its immediate goal the construction of the components that address specific concerns of a well-defined subset of areas. In this section, we attempt to establish the specific organization of such a benchmark We follow the guidelines we established in Sections 2 and 3 for the three areas discussed in Section 4.

Artifact Categories

We consider the following simple taxonomy of software artifacts which may be represented in a benchmark. At the top layer of the hierarchy, we identify two major categories of artifacts: textual and non-textual. Within each of the major categories, we establish a number of distinct subcategories and briefly describe them below.

Artifact/Textual/Free-form. This category includes textual project artifacts written in free-form, unstructured text. Examples include requirements documents and design documents.

Artifact/Textual/Structured. The key representative of this category is code.

Artifact/Textual/Semistructured. This category includes textual artifacts in which individual elements contain both structured and unstructured parts. A typical representative is a collection of bug reports entered through a bug tracking system. Often, for the purposes of the tracing tasks, these artifacts are treated in the same way as free-form textual artifacts.

Artifact/Non-Textual/Unparsed. This category includes a wide range of traditional non-textual artifacts created during the software project lifecycle. Their key distinction is the need to include parsing/recognition/interpretation as part of the solution of the tracing task. Examples include UML diagrams, state-transition diagrams, entity-relationship diagrams, and more.

Artifact/Non-Textual/Parsed. This category includes non-textual artifacts that are already parsed, or which do not require interpretation. The most important representative of this category is an RTM.

Task Categories

Two major task categories related to traceability were observed in the fields described in Section 4: *recovery of traceability information* and *assessment of traceability information*.

Recovery of traceability information, i.e., building traceability information from scratch, is a common task category for all three fields under consideration. A typical structure of such a task involves two artifacts broken into individual elements. The task itself is to build a mapping between elements of one artifact and the elements of the other artifact. In all three fields, traceability is to be established between *different artifacts* (not different versions of the same artifact).

Assessment of traceability information, or evaluation of given traceability mappings, plays a major role in IV&V, less so in the other two areas, although establishing semantics of traceability links (a task from reverse engineering) is an assessment task as well. A typical structure of an assessment task involves a pair of artifacts and a mapping between their elements. For each link in the mapping, the task is to classify it, either using a binary classification scheme (link vs. not a link) or using a more complex classification scheme, which establishes specific link semantics (e.g., "traces to", "depends on", "part of", "not a link"). The task may also involve recovery of links missing in the original mapping.

A third task category, *concept extraction*, is featured prominently in reverse engineering. Here, a single (usually textual) artifact is provided as input, and the task consists of building an abstract view/concept model of the artifact. Such tasks can be viewed as pre-cursors to subsequent *trace recovery tasks*, which involve the extracted higher-level artifacts. Therefore, we include this task category in our view of the benchmark.

The majority of the *trace recovery* tasks described in Section 4 involve tracing textual artifacts: tracing between free-form documents (e.g., between requirements and user's manuals), or tracing between a free-form/semistructured and a structured document (e.g., between design and code, or between bug reports and code). Some tasks do involve tracing between textual free-form/semistructured and non-textual artifacts (e.g., between use case diagrams and test cases documented as text). *Assessment* tasks found in IV&V typically involve two textual artifacts and an RTM, which we classify as a parsed non-textual artifact.

Dataset

There is significant overlap in the data used by IV&V analysts, software maintainers, and reverse engineers. The core textual artifacts of requirements, design, and code are present in tasks in all three fields, albeit, their importance is different. All three artifacts are crucial to IV&V tasks, but code becomes the key artifact in software maintenance and reverse engineering. The dataset must support tasks from all three fields, and it must support tasks involving artifacts of different types. As such, we derive the following list of artifacts to be included in the benchmark:

1. Requirements document. (*Artifact/Textual/Free-Form*)
2. Design document. (*Artifact/Textual/Free-Form*)
3. Code. (*Artifact/Textual/Structured*)
4. Test Cases. (*Artifact/Textual/Semistructured*)
5. Bug Reports. (*Artifact/Textual/Semistructured*)
6. Domain and application concepts. (*Artifact/Textual/Structured*)
7. Abstractions/Logical views.[2] (*Artifact/Textual/Structured*)
8. RTMs.[3] (*Artifact/Non-Textual/Parsed*)
9. Use Cases. (*Artifact/Non-Textual/Unparsed*)
10. System state-transition diagrams. (*Artifact/Non-Textual/Unparsed*)

Additionally, the benchmark can contain the following artifacts:

- Test Plan. (*Artifact/Textual/Semistructured*)
- User Manual. (*Artifact/Textual/Free-form*)
- UML Diagrams for different system aspects. (*Artifact/Non-Textual/Unparsed*)

Each artifact in the dataset must come with at least one predetermined way of separating it into individual elements. For some artifacts, such as bug reports and use cases, such separation is straightforward. Other artifacts, e.g., requirements and design, require some explicit separation into individual elements. Finally, yet other artifacts, such as code, may come with more than one notion of an element. It is possible that some tasks in the benchmark may actually involve determination or correct/convenient separation of an artifact into elements. However, we also want to ensure that the most basic tracing tasks are well-defined for the artifacts that they involve. In the remainder of the benchmark description, we will assume that each artifact is indeed broken into individual elements.

To address the *scalability* requirements, the benchmark has to have the following:

- Some artifacts must be large. As a minimum, at least two artifacts traceable to each other should contain over 1000 elements.
- Some artifacts must be broken into "subsets," from very small to the entire artifact. This is done to allow for artifacts of increasing size. For example, a small requirements document artifact can be a single section of the entire requirements document. A somewhat larger requirements document would incorporate the small document and add a few more sections to it.

Because at this stage we are primarily interested in scalability of the traceability of textual artifacts, the best artifacts to use for these purposes are requirements, design, code, and bug reports.

[2]These can be viewed as incoming data for traceability recovery tasks, and as answer sets for extraction tasks.

[3]We note that, generally speaking, RTMs prepared for assessment tasks have to contain errors in them, and therefore, should be different from the RTMs that represent proper answer sets to the tracing tasks in the benchmark.

Tasks

The benchmark must include tasks from each of the categories discussed above.

Recovery tasks. Generally speaking, a traceability recovery task between any pair of artifacts described above is feasible. However, since 72 traceability recovery tasks yield the need for 36 answer sets (each pair of artifacts can be traced in two directions, but a single answer set suffices for both directions), we elect to include only the most important and frequent traceability tasks. Table 1 illustrates the tracing tasks we consider for inclusion.

Assessment Tasks. Typical assessment tasks in IV&V include assessment of Requirements-to-Design and Design-to-Code traceability matrices.

Concept Extraction Tasks. These tasks typically involve extraction from Code.

Measures

As specified above, the same task performed from different perspectives yields different measures of success. The following properties of the tasks and their results need to be evaluated:

Accuracy. Accuracy, generally, measures the amount of the retrieved links that are correct. Traditional IR measures for accuracy are *precision* (micro precision, operating on an element-by-element level, macro precision, measuring the ratio of correct links in the entire RTM), and measures derived from it, such as *expected precision* and *average expected precision*.

Coverage. Coverage measures the amount of correct links that were retrieved. The traditional IR measure for this is *recall* (micro recall assesses recall for individual high-level elements, macro recall measures the total percentage of correct links that was retrieved).

Precision and recall are often combined in a single measure, *f-measure*, a harmonic mean of precision and recall. Depending on the perceived importance of precision and recall to a specific task, f-measure can be modified with a parameter β, which indicates how the computation of the harmonic mean can be skewed.

Scalability. Two approaches to measuring scalability can be considered. In the first approach, the *time* it takes to complete the tracing task is measured, and the increase in time is compared to the increase in the size of the task. In the second approach, together with the time it takes to complete the task, we monitor the changes in the accuracy and coverage of the solutions, as the problem size increases.

Effort. Because the benchmark can be used with automated, semi-automated, and manual tracing processes, there is not a unique, coherent way to compare effort, without developing models of effort in advance. In general, we would like to measure the effort of a human analyst throughout the process. There are two basic measures that can be used: absolute number of links the analyst (or a simulated analyst) had to examine, and *selectivity*, i.e., the percentage of all possible links that the analyst (simulated analyst) had to examine.

Answer Sets

We use this section to comment on the amount of effort that we anticipate in order to create even this scaled-down benchmark. As seen from Table 1, we have elected to include 23 different (46, if counted both ways) trace recovery tasks between nine different artifacts in the benchmark. This mandates 23 full mappings between the artifacts. Our experience with creating ground truth for even moderate-size datasets indicates that a very significant effort (perhaps unachievable by a single research group) is required.

	Reqts.	Design	Code	Test Cases	Bugs	Concepts	Views	Use Cases	Diagrams
Requirements document	-	**X**	o	**X**	**X**	o	o	**X**	**X**
Design document	**X**	-	**X**	**X**	**X**	o	o	**X**	**X**
Code	o	**X**	-	**X**	**X**	**X**	**X**	**X**	**X**
Test Cases	**X**	**X**	**X**	-	**X**	o	o	**X**	o
Bug Reports	**X**	**X**	**X**	**X**	-	**X**	**X**	**X**	o
Domain and application concepts	o	o	**X**	o	**X**	-	**X**	o	o
Abstractions/Logical views	o	o	**X**	o	**X**	**X**	-	o	o
Use Cases	**X**	**X**	**X**	**X**	**X**	o	o	-	**X**
System state-transition diagrams	**X**	**X**	**X**	o	o	o	o	**X**	-

Table 1: Traceability recovery tasks. "X": mandatory task, "o": optional task.

Conclusion

Traceability is an important activity that permeates many disciplines, notably software engineering. Advances have been made in the automation of traceability, but progress is hindered by the lack of standard artifacts, measures, data formats, etc. Benchmarks have proven useful in other research fields, and it is our position that traceability can also benefit from a benchmark.

This paper does not describe an existing benchmark. Rather, it establishes the properties and the structure for a yet-to-be-built benchmark. We have made an effort to describe a benchmark which would address traceability challenges in more than one area of software engineering. Our approach was to balance the complexity of the benchmark and its applicability. As a result, we arrived at the description, which, as we have observed, requires substantial effort to implement. It is our hope that this paper will spark interest and lead to a community-wide initiative to build such a benchmark.

Acknowledgments

The work of the first two authors is funded, in part, by NASA under grants NNG05GQ58G8G and NNX06AD02G and by NSF under grant CCF-0647443. Giuliano Antoniol was partially supported by the Natural Sciences and Engineering Research Council of Canada (Canada Research Chair in Software Change and Evolution #950-202658). We thank Marcus Fisher, Stephanie Ferguson, Jane Cleland-Huang, and all the participants of the workshop on Grand Challenges in Traceability, which took place on August 4-5 in Fairmont, WV. We also thank Andrian Marcus for an impassioned discussion on why this cannot be done.

6. 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] G. Antoniol, Y. Gueheneuc An Epidemiological Metaphor for Feature Identification. *IEEE Trans. Software Eng*, 32(9) pp 627-641 (2006)

[3] E. Chikofsky and J. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, Jan 1990.

[4] J. Cleland-Huang, A. Dekhtyar, J. Huffman Hayes (Eds.). Grand Challenges in Traceability, *draft*, *Center of Excellence for Traceability tech. report* COET-GCT-06-01-0.9, http://www.traceabilitycenter.org/downloads/documents/GrandChallenges/, September 10, 2006.

[5] D. Edwards, S. Simmons, and N. Wilde, "An approach to feature location in distributed systems," Software Engineering Research Center, Tech. Rep., 2004, *SERC-TR-270*.

[6] R. Fiutem and G. Antoniol and P. Tonella and E. Merlo ART: An Architectural Reverse Engineering Environment *Journal of Software Maintenance Research and Practice*, Num. 11, 1999, pages:1-25

[7] J. Huffman Hayes and A. Dekhtyar. Humans in the Traceability Loop: Can't Live with 'Em, Can't Live Without 'Em, (2005), in *Proceedings, 3d International Workshop on Traceability in Emerging Forms of Software Engineering*, pp. 20-23, Long Beach, CA, November 7, 2005.

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

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

[10] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. "Programmable reverse engineering." *International Journal of Software Engineering and Knowledge Engineering*, pages 501-520, December 1994

[11] M. Weiser Program slicing *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[12] Xie, X., Poshyvanyk, D., and Marcus, A. "3D Visualization for Concept Location in Source Code" in *Proceedings of 28th IEEE/ACM International Conference on Software Engineering(ICSE'06)* May 20-28, Shanghai, China, pp. 839-842