Model-Based Traceability

Jane Cleland-Huang[1], Jane Huffman Hayes[2], J. M. Domel[1]
[1]*School of Computing*
DePaul University
{jhuang,jdomel}@cs.depaul.edu
[2]*Department of Computer Science*
University of Kentucky
hayes@cs.uky.edu

Abstract

Many organizations invest considerable cost and effort in building traceability matrices in order to comply with regulatory requirements or process improvement initiatives. Unfortunately, these matrices are frequently left un-used and project stakeholders continue to perform critical software engineering activities such as change impact analysis or requirements satisfaction assessment without benefit of the established traces. A major reason for this is the lack of a process framework and associated tools to support the use of these trace matrices in a strategic way. In this position paper, we present a model-based approach designed to help organizations gain full benefit from the traces they develop and to allow project stakeholders to plan, generate, and execute trace strategies in a graphical modeling environment. The approach includes a standard notation for capturing strategic traceability decisions in the form of a graph, and also notation for modeling reusable trace queries using augmented sequence diagrams. All of the model elements, including project specific data, are represented using XML. The approach is demonstrated through examples from a traffic simulator project composed of requirements, UML class diagrams, code, test cases, and test case results.

1. Introduction

Successful traceability processes generally incorporate a carefully devised planning stage to determine when, how, where, and why each traceability link will be created [5]. By aligning traceability tasks with project and organizational goals, and capturing these planning decisions in a trace strategy graph, project stakeholders can ensure that their traceability effort provides effective support for critical software engineering tasks. In prior work, several researchers have proposed the use of strategic traceability graphs [5,1] to define the traceability goals of a project. In its simplest form, such a graph defines a set of traceable artifacts, the traceability links between them, and the purpose of each of these links.

However, such graphs fall short of providing the level of support needed to fully automate the traceability process, especially when artifacts are stored in heterogeneous formats. Furthermore, there is no easy way to facilitate reuse of *trace strategies*, decisions to create and maintain a certain set of traceability links, across different artifacts within a project [4]. These limitations were illustrated through an informative discussion between one of the authors and a group of developers from a large, safety-critical project at a level 3 CMMI organization. In this project, complete traceability between requirements, design, and code was required as part of the process improvement initiative. Despite having a relatively accurate and complete set of traceability matrices, the developers claimed never to use these traceability matrices at all. There appeared to be a usability problem which meant that extracting useful information from the matrices was not seen as a cost-effective or useful exercise. One possible reason for this was the lack of supporting traceability tools with which to generate queries and review the results.

This paper addresses these problems through proposing a new model-based approach which builds on the underlying structure of the traceability strategy graph. In much the same way as a model-driven engineering environment focuses on abstracting the particulars of a domain from the underlying platform-level implementation details [6], model-based traceability separates out tracing strategies and reusable trace queries from their underlying document representations. This separation of concerns makes it much easier for traceability strategies and queries to be reused across projects. It also means that the traceability links become significantly more accessible and common queries are supported through the simple click of a button.

2. Traceability Model Representation

Our proposed model-based traceability approach includes four distinct layers:

- **The strategic layer** captures the artifacts and their associated traceability links in a model known as the *strategic traceability graph*. This takes the form of a traceability metagraph as previously defined by Ramesh [5] which defines the types of artifacts to be traced, and specifies link types as well as additional information defining who will use the links, when they will be used, and where the data is stored. Building the strategic traceability graph forces project stakeholders to make decisions about how much traceability they want in their

project and to analyze the purpose and projected benefits of their traceability plan.

- **The document management layer** records the names and locations of all project level software artifacts and traceability matrices. In early prototypes of our approach, these components are represented as XML documents; however, it is relatively straight forward to extract this information, periodically or on demand, from 3rd party CASE tools in order to keep the XML representations current [3].
- **The stored query layer** defines a set of queries that are facilitated by the lower-level strategic and document management layers. These queries are designed by a project manager or requirements analyst and used by other stakeholders in the course of their regular software engineering tasks.
- **The executable layer** is responsible for interpreting the queries in the stored query layer, transforming them into executable code, invoking them, and reporting or visualizing the results.

In our model, reuse of trace strategies and queries is facilitated through the separation of these four concerns. Continuing the analogy of model-driven software engineering, the strategic and stored query layers are similar to a platform independent model (PIM), while the document management layer relates to the platform specific model (PSM) [6]. Both the strategic and stored query layers are reusable across projects as they are mapped to project-specific physical files by the document management layer.

2.1 The Strategic Layer

To illustrate and evaluate our approach, we first introduce a simple project consisting of 54 software requirements, and their associated UML classes, java code, JUnit test cases, user acceptance tests, and test results. These artifacts were taken from a small software project to create a traffic simulation model, developed as part of the coursework by a student at DePaul University. The strategic traceability graph showing artifacts and their associated traceability links is depicted in Figure 1. In this example, the traceability paths were chosen because they supported basic software engineering tasks such as determining if the design satisfied the requirements, or if the code implemented the design.

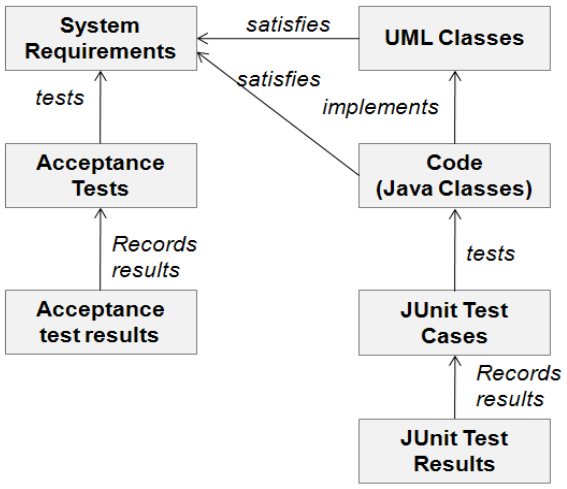


Figure 1. Strategic traceability graph for the traffic simulation project

The underlying graph structure is documented using a standard XML representation. In this very simple example, bidirectional traceability is facilitated between requirements and design, design and code, code and test cases, requirements and test cases, and test cases and their results. Trace relationships for a small subset of the artifacts are shown in the following XML specification:

```
<ProjectTraces>
    <Artifact>
      <ArtType>Requirements</ArtType>
      <ArtName>SystemRequirements    </ArtName>
      <XMLFile>XMLReqs1.xml</XMLFile>
    </Artifact>
    <Artifact>
      <ArtType>UMLClasses</ArtType>
      <ArtName>UMLClasses</ArtName>
      <XMLFile>XMLReqPro142.xml</XMLFile>
   </Artifact>
   <TracePath>
      <Source>UMLClasses</Source>
      <Sink>SystemRequirements</Sink>
      <TraceRep>Matrix</TraceRep>
      <TraceType>Satisfies</TraceType>
   </Tracepath>
</ProjectTraces>.
```

Although not included here due to space considerations, this XML file also documents layout coordinates of the strategic traceability graph's visual components. Actual file names and information about the traceability matrices are project specific and are therefore provided as part of the document management layer.

2.2 The Document Management Layer

Each individual set of artifacts is also represented using a standard XML format. For example, an XML representation for one of the requirements is shown below:

```
<Requirements>
    <Requirement>
        <ReqID>2.1</ReqID>
        <ReqText>The map layout shall be defined
        in a text file.</ReqText>
        <ReqType>Business Use Case</ReqType>
        <ReqParent>2.0</ReqParent>
        <Module>Maps</Module>
        <Status>Implemented</Status>
    </Requirement>
</Requirements>.
```

Noteworthy features of this XML document are that it includes specific attributes that are deemed strategic for performing the most common types of traces. For example, traces might be filtered by module name or requirements

status, and therefore these attributes are included in the XML document.

Other artifact types such as UML Classes, code, test cases, and test results are represented in similar ways using appropriate XML schemas. For example, source code is represented in a format known as srcML [2]. These other artifacts are not shown here due to space constraints. All trace matrices depicted in the strategic traceability graph are represented as follows:

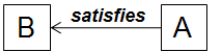
```
<TraceMatrix>
    <TraceType>Satisfies</TraceType>
    <Trace>
        <SourceID>13</SourceID>
        <SinkID>1102</SinkID>
        <Confidence>1</Confidence>
    </Trace>
    .... More traces here
</TraceMatrix>.
```

This file includes source ID and target ID of each link, as well as a confidence score. When just-in-time tracing tools based on information retrieval methods are used, this confidence score is computed automatically, while it is initially set to 1 for manually constructed trace matrices.

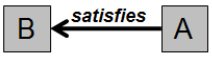
2.3 The Stored Query Layer

The stored query layer composes queries from different types of primitive links. It also constructs composite queries out of lower-level more primitive ones. This section describes these composition methods and also introduces a possible visual representation to be used in our modeling notation. As traces are dynamic, we define them using sequence-style diagrams that we call *trace query sequence diagrams*, and then map them onto the strategic traceability graph in much the same way as a use case map maps a use case scenario onto a class diagram. Note that strategic traceability graphs can be hierarchical. If more than one such graph exists, such as for a large project with many artifacts, we map the trace query sequence diagrams onto the high level strategic traceability graph. The actual notation shown here is just an example of the notation that will used in the final modeling language.

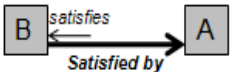
Primitive traces occur between two adjacent artifact types in the strategic traceability graph. This is the simplest form of trace for which three primary factors can vary independently. The first factor is the direction of the link. For example, the trace *UMLClasses satisfies Requirements* could support forward traces such as "is requirement R satisfied in the design?" or backward traces such as "does this design element trace back to a requirement?" A *satisfies* trace between two artifacts such that *A satisfies B* in the original strategic traceability graph is modeled as follows:



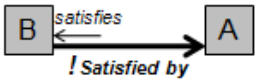
A trace query that uses this link as-is is shown in bold, as follows:



A reverse direction trace is depicted in a similar way. However, the original name and direction of the trace is depicted as an annotation above the reverse trace, as shown below:

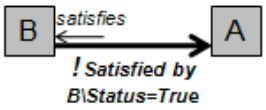


Another common case of tracing across primitive links defines whether the trace should return a set of matching (i.e., traced) artifacts, i.e., "*Return a list of B elements and the A elements that satisfy them*" or instead return the missing set with the query "*Return a list of B elements that are not satisfied by one or more A elements*." In our proposed modeling notation, we represent the case of the missing set query as:



in which the **!** symbol represents the missing trace query described above.

Primitive traces can also be constrained according to filters. For example, we might issue a trace from B to A, in which B is filtered according to the value of some attribute. Each filter is set on one specific artifact type. For example, in the following diagram, artifact B is filtered to only include records for which Status = True. This is represented in our modeling notation as follows:



Composite traceability paths occur between two non-adjacent artifact types in the traceability graph. They are used to compose two or more primitive traces. Composite traceability paths can be visualized on the strategic traceability graph, as depicted in Figure 2, which maps the trace query "return a list of requirements that trace to code which has failed its test case." However, as trace queries can be fairly complicated, we have adopted the technique of modeling them in sequence-style diagrams, as shown in Figure 3. There are several different ways (called *traceability paths*) in which this trace query could be implemented. For example, we could start at requirements, and trace via code, to test cases, and then to test case results and then filter out the results to include only those requirements with failed test cases. Alternately, we could start at the failed test cases and then trace back to find corresponding code and related requirements. Generally, most trace queries can be performed in more than one way. In this case the results should be the same, but the second approach is more efficient. On the other hand, the second approach would not catch the case of untested code, as it is

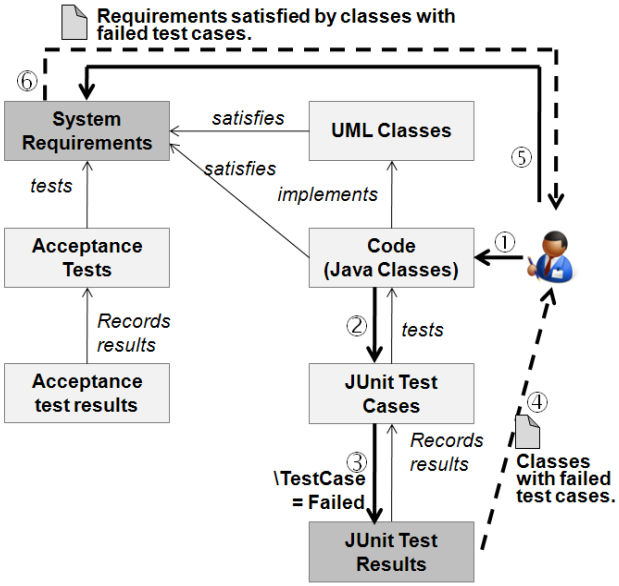


Figure 2. Trace query mapped onto the strategic traceability graph

limited to detecting failed test cases. The purpose of the trace must therefore be carefully considered and modeled. One of the purposes of modeling a trace is so that it can be automatically transformed into an executable query. This is discussed in the following section on the executable layer.

Composite trace queries occur when multiple traceability paths are combined to increase the reliability or the knowledge of the trace result. For example, Figure 4 shows two different traces which are combined to answer the query of whether there are any missing links between requirements and code which could be discovered by exploring indirect links from requirements, via design, to code. In Figure 4, sub-query 1 returns the set of requirements which do not have traceability links to code, while sub-query 2 uses these untraced requirements as input, and traces them via the UML design to look for indirect traces to code. The results can be used to help an analyst update the requirements to code traceability matrix. This is an example of a composite trace.

Sub-queries can therefore be combined together in several different ways including: (i) Composition, (ii) reporting multiple results as separate entities, (iii) displaying multiple results in a single physical report without making any attempt to reconcile the results, and finally (iv) defining a set of heuristics or formulas for merging the results into a single consolidated result set. As a very simple example, a trace could be returned either if both result sets show it to be correct or if at least one of them does. Alternately, a voting scheme could be employed to generate a final set of results.

2.4 The Executable Layer

The executable layer provides a relatively simple user interface which displays all pre-defined queries for the project. A non-technical project stakeholder could therefore select a query, execute it, and analyze the results either using a special visualization tool, some kind of word processing document, or a spreadsheet.

The executable layer is responsible for transforming a query in the model into the underlying XQuery representation. Our modeling notation is designed to support this kind of transformation. For example, the first trace in Sub-query 1 of Figure 4 represents a request to retrieve the list of system requirements that are not satisfied by code. To accomplish this, a complete list of system requirements is retrieved from Requirements.xml (as defined in the CodeToReqs.xml matrix document). The *diff* between this list and the list of traced requirements in the trace matrix is computed, and a list of requirements with no matching code links is returned.

The executable layer is also responsible for various visualizations and reporting formats of the delivered results. For this reason, the sequence diagram in Figure 3 also shows a "Visualizer" object.

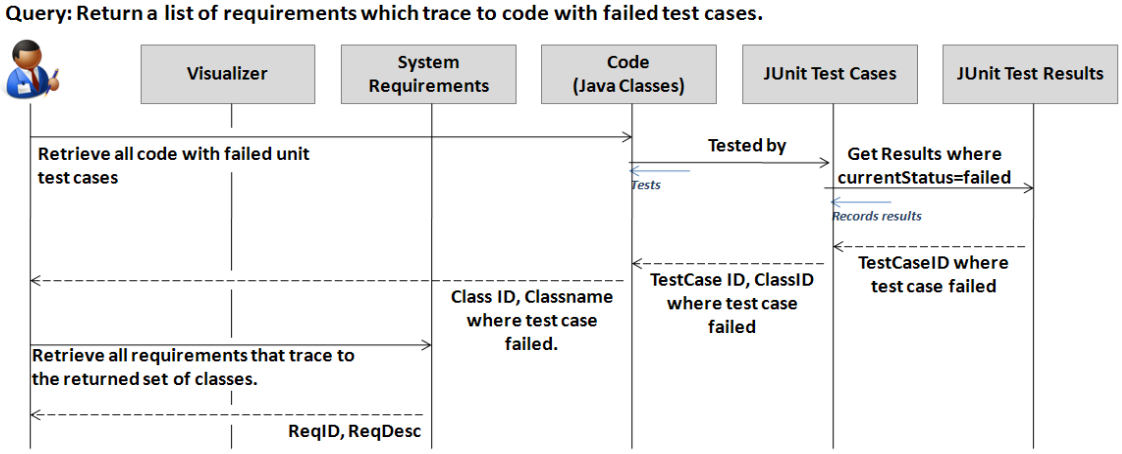


Figure 3. Defining a trace query as a sequence diagram

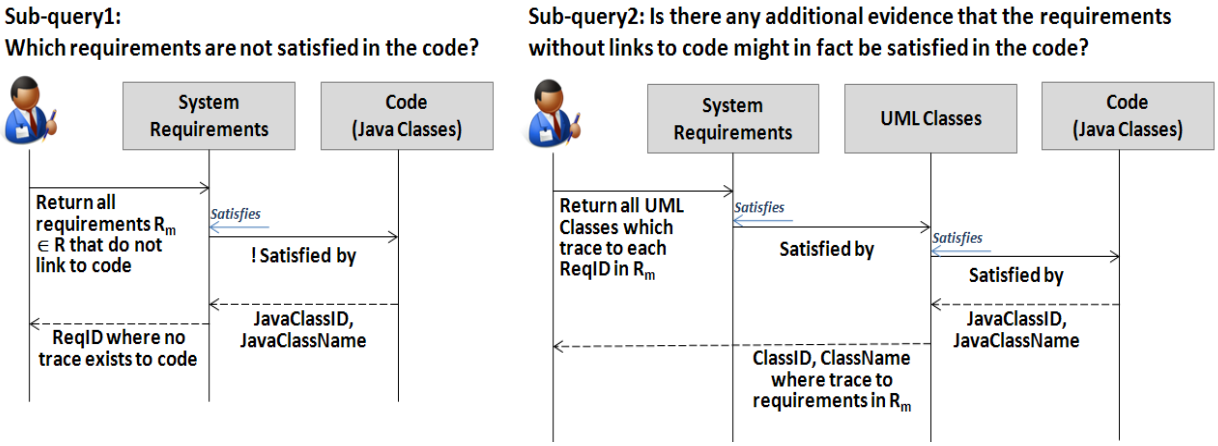


Figure 4. A composite query in which the output of one sub-query provides the input for another

3. Reuse across Projects

In addition to easing the task of developing and issuing traceability queries, several parts of our approach are easily shared across projects. For example, an organization could define a generic strategic traceability graph and associated trace queries for use across multiple projects. These would then be mapped to specific documents at the individual project level. Our approach also allows for trace queries to be organized by role, so that a tester sees one specific subset of queries, while a project manager sees another.

4. Conclusions

This position paper has proposed a new approach for managing traceability strategies and queries in which project stakeholders can plan, generate, and execute trace strategies in a modeling environment. The four layered model builds on existing techniques for defining traceability metagraphs, and then introduces a new technique for modeling reusable traceability queries. The proposed approach requires little additional effort above and beyond the non-trivial effort already needed to create and maintain traceability links; however its benefits are realized when traceability queries are made more accessible to more stakeholders, and when these queries and associated strategic traceability graphs are re-used in subsequent models.

The closest related work by Sousa et al. [7] introduced a traceability based framework for product line development, in which traces were executed through instantiating abstract classes. However their approach is very development centric and not designed for general stakeholder use. Our approach has the potential to significantly improve the benefits that can be achieved by establishing traceability matrices for a project. We have currently implemented a prototype of our model using a set of XML documents and associated queries executed using a tool named BASEX.

Current work is focused on developing and comparatively evaluating several candidate modeling notations to determine which approach is most intuitive for an average user while supporting a wide variety of queries. Our current work involves building the GUI prototype outlined in this paper. Future work will test the utility of the modeling notation and the overall approach against much larger and more varied sets of software artifacts.

Acknowledgments

This work is funded in part by the National Science Foundation under NSF grants CCF-0811140 and 0810924.

References

[1] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova. "Best Practices for Automated Traceability," *Computer*, 2007, vol. 40, no. 6, pp. 9.

[2] M.L.Collard, J.I. Maletic, and A. Marcus, (2002), "Supporting Document and Data Views of Source Code," in Proceedings of ACM Symposium on Document Engineering (DocEng'02), McLean VA, November 8-9, 2002, pp. 34-41.

[3] J. Lin, J. Cleland-Huang, R. Settimi, J. Amaya, G. Bedford, B. Berenbach, O.B.Khadra, C.Duan, and X. Zou, (2006), "Poirot: A Distributed Tool Supporting Enterprise-Wide Traceability," in Proceedings of IEEE Intn'l Conf. on Requirements Engineering (RE'06) - Tool Demo, Sept. 2006.

[4] R. Oliveto, A. Marcus, J. Huffman Hayes, "Software Artefact Traceability: the Never-Ending Challenge." ICSM 2007: 485-488

[5] B. Ramesh, and M. Jarke, (2001), "Towards Reference Models for Requirements Traceability," *IEEE Trans.on Software Engineering*, vol. 27, no. 1, Jan, pp. 58-93.

[6] D.C.Schmidt. "Model-driven engineering," IEEE Computer, vol. 39, no 2, February, 2006, pp. 25-31.

[7] Sousa et al, "A Model-Driven Traceability Framework to Software Product Line Development," *ECMDA Traceability Workshop*, June, 2008.