A Framework for Managing Traceability Relationships between Requirements and Architectures

Susanne A. Sherba and Kenneth M. Anderson
University of Colorado
Department of Computer Science
Boulder, CO 80309-0430 USA
{sherba, kena}@cs.colorado.edu

Abstract

Traceability helps stakeholders to understand the relationships that exist between software artifacts created during a software development project. For example, the evolution of the relationships between requirements and the components to which they are allocated can provide insight into the maintainability of a system. Unfortunately, due to the heterogeneous nature of these artifacts, creating, maintaining, and viewing these relationships is extremely difficult.

We propose a new approach to traceability based on techniques from open hypermedia and information integration. Open hypermedia and information integration provide generic techniques for establishing, maintaining, and viewing relationships between software artifacts. Our approach allows the automated creation, maintenance, and viewing of traceability relationships in tools that software professionals are accustomed to using on a daily basis.

1. Introduction

Traceability can provide important insight into system development and evolution. Antoniol *et al.* maintain that traceability assists in both top-down and bottom-up program comprehension [4]. According to Jacobson, Booch, and Rumbaugh, "[t]raceability facilitates understanding and change [9, page 10]." Palmer asserts that "[t]raceability gives essential assistance in understanding the relationships that exist within and across software requirements, design, and implementation [13, page 412]."

Even if we limit our discussion to relationships between requirements and architectural artifacts, we are confronted by a large number of potentially useful relationships. The models of Ramesh and Jarke [16] and Pohl [14, 15] suggest possible relationship types between various elements and artifacts. Han [7] lists three categories of structural relationships: coarse-grained inter-document, fine-grained inter-document, and fine-grained intra-document. We divide inter-document relationships into two subcategories: relationships between different versions of the same artifact and relationships between different artifacts. Furthermore, we consider relationships between both consecutive and non-consecutive versions of the same artifact as well as relationships between relationships. Figure 1 depicts these five relationship types.

Relationships can exist between elements of a single artifact (relationship type 1 in Figure 1). For example, in a requirements specification, one requirement might *elaborate* or *depend_on* another. In an architectural diagram, a component might be *part_of* another component or *depend_on* another component. Across versions of the same artifact (relationship types 2 and 3) we may observe relationships such as *refines*, *replaces*, *based_on*, and *formalizes*. Between requirements and architectural components, relationships such as *satisfies* and *allocated_to* might be useful (relationship type 4). Furthermore, software engineers may be interested in how a particular type of relationship between artifacts evolves over time (relationship type 5). For example, a *number_of_instances* relationship might provide insight into component cohesion. If the number of *allocated_to* relationships between a requirements document and a component diagram explodes after an iteration in the design phase, this may indicate that one or more components have lost cohesion.

All of these relationship types might be useful to one or more stakeholders at some point in the software development project; however, the vast number of possible relationships makes the task of manually creating and maintaining these relationships daunting. Furthermore, different stakeholders have diverse information needs. Not all relationships will be of interest to all stakeholders. For example, a customer might only be interested in knowing that all requirements have been allocated to components whereas a

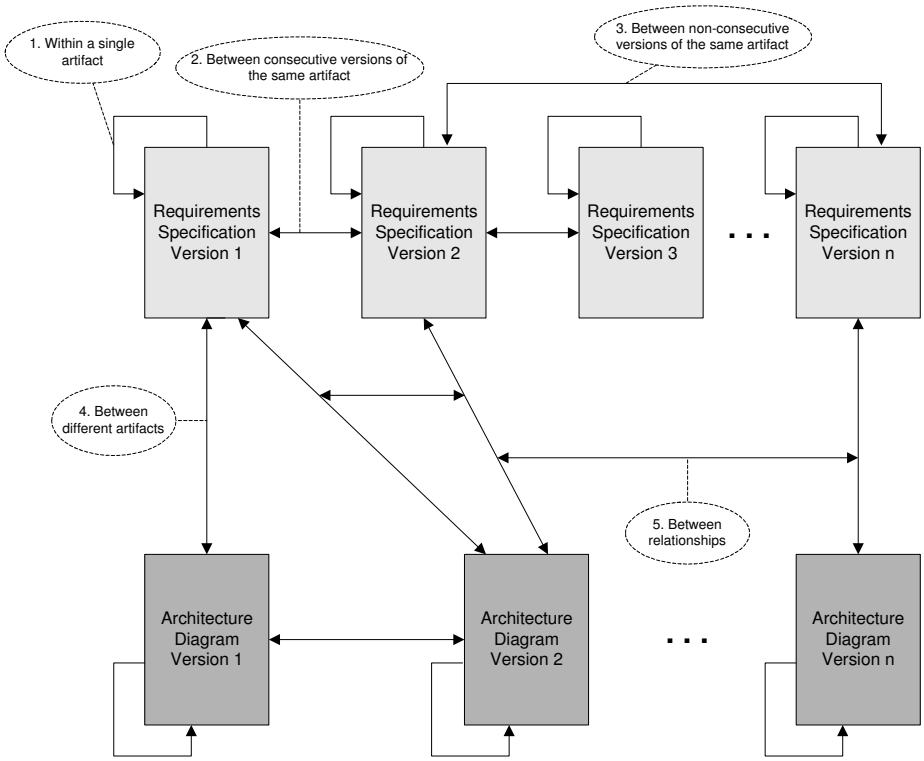

Figure 1. Implicit and Explicit Relationships in Software Artifacts.

software developer might need to understand the requirements in light of their allocated components and subcomponents as well as the dependencies between these components. Stakeholders need to be able to filter relationships to provide a view of the information space that conforms to their information needs.

Finally, users should not be required to use a specialized tool to view traceability information. Different stakeholders use different tools to produce requirements and architectural artifacts. It is more likely that stakeholders will make use of these relationships if they are able to view the relationships in the tools that originally created the artifacts. However, these tools are often not designed to interact. Thus, we must also overcome the problem of heterogeneous artifacts produced by different tools [3].

These problems lead to three requirements for creating, maintaining, and viewing traceability relationships:

- The creation and maintenance of traceability relationships must be automated.
- Stakeholders must be able to create a view of traceability relationships based on their information needs.
- Users should be able to create and view traceability relationships within common, familiar software tools.

In addition, we suggest that a traceability tool should provide support for evaluating the evolution of relationships between artifacts. This analysis can provide insight into the entire project.

2. Approach

We hypothesize that open hypermedia [12] and information integration [2] enable an approach to traceability that allows:

1. automation of the discovery, creation, and maintenance of traceability relationships.
2. customized views of these relationships based on the information needs of the stakeholders.
3. creation and viewing of these relationships in the tools that originally created the artifacts.

In this section, we provide a brief overview of open hypermedia and information integration and then present our conceptual framework. Next, we offer a motivating scenario and then describe our proposed prototype.

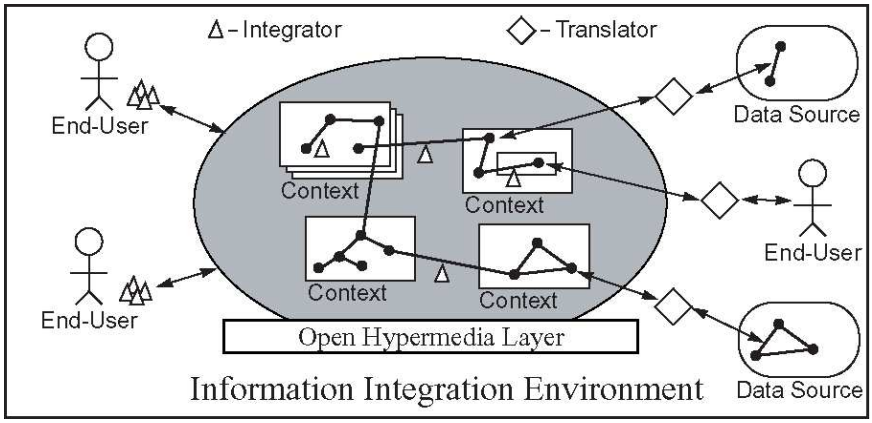

Figure 2. The InfiniTe Information Integration Environment [2].

2.1. Open Hypermedia

Open hypermedia systems [12] enable the creation and viewing of relationships in heterogeneous applications as well as the traversal of those relationships within and between applications. Open hypermedia services allow links (relationships) to be stored separately from an artifact. The open hypermedia data model supports complex relationships such as bi-directional relationships, relationships with multiple anchors (n-ary relationships), and relationships between relationships [3]. Open hypermedia systems also provide services to filter relationships based on type and to create collections of relationships (hyperwebs or, more commonly, linkbases).

2.2. Information Integration

Information integration [2] provides services to automate the discovery, creation, maintenance, and evolution of relationships between heterogeneous artifacts. Information integration uses the concept of a *data source* to model information outside the information integration environment. *Translators* are responsible for importing information from data sources into the information integration environment as well as exporting information from the information integration environment to data sources. *Integrators* work within the environment to automate the discovery and creation of relationships. Specific integrators can be developed to find different relationships within the environment. These relationships can be between artifacts, other relationships, or collections of artifacts and relationships. In addition, information integration uses *contexts* to model different views of the information space. A conceptual model of the InfiniTe information integration environment [2] is shown in Figure 2.

2.3. Conceptual Framework

Our conceptual framework builds on techniques from open hypermedia and information integration. The main elements in our framework are *tool*, *artifact*, *relationship*, and *metadata*. These concepts are illustrated in Figure 3. A *tool* is something that a stakeholder uses to perform a task. Examples of tools include word processors, UML diagramming tools, integrated development environments (IDEs), mail programs, version control systems, and issue tracking systems. An *artifact* is produced by a tool. A *relationship* is a semantic association between artifacts, portions of artifacts, or relationships. *Metadata* allows a method engineer (or someone familiar with the software project) to describe the artifacts and relationships that are created and used during the project.

As can be seen in Figure 3, stakeholders are able to use their original tools. These tools produce heterogeneous artifacts. Artifact and relationship metadata provides information about artifact and relationship types. The metadata includes information such as which translators can be applied to specific artifact types or which integrators can be used to find particular relationship types. The artifacts are translated into the information integration environment where traceability relationships can be automatically discovered and created by appropriate integrators as determined from the metadata. These relationships are forwarded to the open hypermedia system, which provides services to display the relationships in the tools that originally created the artifacts. The traceability system provides services to schedule integrators and translators, to "chain" relationships together to form new relationships, to register new artifact and relationship types, to create customized views by filtering both artifacts and relationships, and to provide insight into system evolution based on the traceability relationships that the system has discovered.

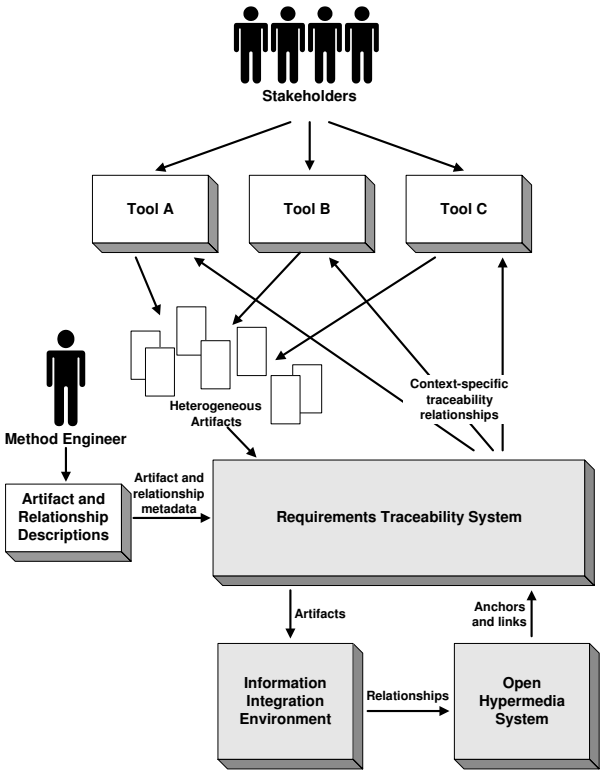

Figure 3. Conceptual Traceability Framework.

2.4. Motivating Scenario

This section provides an example of the use of relationships to provide insight into the evolution of requirements and a system architecture. The hypothetical product is an income tax program. An initial analysis determined that much of the functionality of the system would remain stable (e.g, user interface, data import and export). However, the tax calculation component of the software would need to be frequently updated to reflect current state and federal tax laws. Thus, maintainability has been identified as one of the important quality requirements of the software architecture.

In the initial architecture, the system architect chose to allocate all requirements related to the calculation of taxes to one component. The product is now in its fifth release and the project manager has observed that it is taking software developers more time to update the software to reflect the current tax laws. In addition, these changes are adversely affecting other functionality in the system.

The system architect is called in to review the evolution of the architecture. To analyze the problem, he requests that he be given a summary of all *allocated_to* relationships between tax computation requirements and the components that satisfy those requirements. From this information, he is able to discern that in the first three product releases, all tax computation requirements were indeed satisfied by the tax component. However, in the fourth and fifth releases, the relationships show that some of the tax requirements are being satisfied by a component that also satisfies data import requirements. For more details, the architect opens the requirements specification associated with the fourth product release. He requests that the system display all *allocated_to* relationships. By traversing these relationships, he is able to view the component that currently satisfies the requirement. Thus, the system architect is able to identify the point at which the architecture began to lose its conceptual integrity [5] and to suggest changes to restore the cohesion of the tax component.

This scenario shows the need to be able to create and view relationships between different versions of different artifacts in a customized context. In addition, it suggests that the evolution of relationships over various versions and product releases can provide valuable insight into system evolution.

2.5. Prototype

To evaluate the feasibility of our approach, we plan to build a traceability system that implements the conceptual framework described in Section 2.3. The traceability system will provide an infrastructure for the automated

discovery, creation, and maintenance of traceability relationships between heterogeneous artifacts. The system will be built using services from both open hypermedia and information integration as well as our own traceability-specific services. To provide open hypermedia services, we have chosen the Chimera open hypermedia system [3]. We will use the InfiniTe information integration environment [2] for information integration services.

InfiniTe will provide services for the automated creation of typed traceability relationships. As described in Section 2.2, translators will be used to import information to and export information from InfiniTe; integrators will be used to discover and create relationships. Chimera will provide services for viewing and traversing relationships created by InfiniTe in the tools that originally created the artifacts. To utilize all open hypermedia services, a tool will need to be integrated with Chimera (a Chimera "client").[1] If a tool is not integrated with Chimera, the user will be able to request an HTML summary of the artifact's relationships. The relationships created by InfiniTe will be typed to facilitate filtering these relationships to create customized contexts. Chimera allows the selective viewing of relationships based on type.

Our customized traceability services will be based on information provided in the artifact and relationship metadata. A metadata definition tool will allow a method engineer to define the artifacts and relationships that are required for an organization's software development process. Medvidovic *et al.* [10] describe strategies for modeling architectures in UML. Assuming that a project creates its requirements specification in Microsoft Word [11] and creates architecture diagrams in a UML diagramming tool, we can create specific translators to translate these artifacts into InfiniTe. We can then write one or more integrators to find relationships of interest, for example the *allocated_to* relationship between requirements in the requirements specification and components in the architecture diagram. In addition, we can create one or more integrators to analyze the evolution of the relationships between these artifacts. These translators and integrators can be invoked automatically or as needed, depending on the definitions in the metadata.

Since it is impossible to anticipate the needs of every software development team, we do not propose to build a comprehensive set of translators and integrators for every artifact and relationship type. We do, however, propose to implement a system that will manage artifact and relationship metadata, inform users of available options for integrators and translators, and invoke translators and integrators when appropriate. The system will also provide information about explicit and implicit relationships in the system as well as provide filtering based on both artifact and relationship types. Users of the traceability system will be able to customize the system to their traceability needs by defining artifact and relationship metadata; they then can create and register translators and integrators that translate the defined artifacts and create the required relationships.

To evaluate the utility of our approach, we plan to apply it to artifacts from an existing research project. We will develop translators and integrators and then use the traceability system to automatically generate traceability links between representative artifacts. Whenever possible, we will make use of techniques described in the literature to find these relationships.

The prototype will also allow us to perform a preliminary evaluation of the user interface. We will engage 2–5 computer science graduate students to perform several tasks (e.g., invoking a translator and an integrator, displaying different types of relationships). This preliminary evaluation will help us to detect useability problems so that they will not influence later evaluation.

In the next phase of our evaluation, a developer familiar with the research project will manually create traceability links in a commercial tool. We will record and compare the time involved in creating these relationships in the commercial tool and in our tool; we will also compare the number and types of links created by each of these approaches.

For the final phase of our evaluation, we will seed the research system with several defects. We will engage 5–10 software developers of similar programming experience who have no experience with the research project. The test subjects will receive training on our traceability system and the commercial tool as well as an overview of the research project and development environment. We will then ask them to locate and correct defects using three different sets of information:

1. No explicit traceability links represented—developers are free to use any tools with which they are familiar (e.g., grep and find).
2. Traceability links created and represented in the commercial tool.
3. Traceability links generated by our traceability system and represented in the tools that originally created the artifacts.

We will collect data on the number of defects found using each approach, the time required to find and fix each defect, and the ability to correct a defect without introducing new problems. We will use post-evaluation surveys to record the programmers' experiences in using the different tools and to solicit their opinions on the various approaches. We will then analyze this data to determine the utility of our system in locating and fixing defects as compared with the other two approaches.

[1] Fortunately, the open hypermedia community has developed techniques to facilitate application integration [6, 17, 18].

3. Related Work

Han [8] describes an information model for requirements and architecture management. The model provides the structure for two templates, a *System Requirements Document* and a *System Architecture Document*; a tool, TRAM, facilitates the creation of documents based on these two templates. TRAM's use of templates differs from our approach in that with a template approach the data must conform to a prescribed format. Our approach allows a user to create custom translators and integrators to handle different artifact formats and relationship types.

The Unified Software Development Process [9] defines trace dependencies between elements of its various models. For example, "[a] use-case realization [in the analysis model] ... provides a straightforward trace to a specific use case in the use-case model [9, page 186]." The Unified Software Development Process differs from our approach in that it prescribes specific artifacts and relationships. In our approach, we allow users to create and maintain artifacts and relationships of interest to the project (by locating or creating appropriate translators and integrators). Thus, with appropriate metadata definitions, our approach can be adapted to various software development processes.

Pohl *et al.* [15] describe six meta-models for requirements and architectural artifacts. They then define dependencies between the meta-models. The introduction of explicit dependencies between use case and architecture scenarios allows dependencies between other requirements and architectural models to be "derived". The derivation of dependencies is the same as "chaining" of relationships in our system. Thus, these meta-models can be realized in our system. A user would need to create or locate appropriate translators for the requirements and architectural artifacts. Integrators to create the explicit relationships would need to be developed as well. Our system could then manage the automatic generation and representation of Pohl's derived relationships.

4. Conclusion

We believe that the services of open hypermedia and information integration can be leveraged to provide an approach to traceability that facilitates the automated discovery, creation, maintenance, and viewing of relationships in tools that originally created the artifacts. Furthermore, these services can provide a customized view of the information space.

This research is still in its early stages. To evaluate our hypothesis, we plan to build a prototype traceability system along with representative translators and integrators to find relationships between requirements and architectural artifacts. We have demonstrated the feasibility of the cycle represented in Figure 3 [1] and have already built several translators and integrators for text, HTML, and source code artifacts [2].

Although we believe that our approach can be successfully applied to requirements and architectural artifacts and relationships, our approach is not limited to these artifacts and relationships. We envision that, by developing an appropriate set of translators and integrators, the approach can address traceability concerns throughout a software development project.

References

[1] K. M. Anderson and S. A. Sherba. Using Open Hypermedia to Support Information Integration. In *OHS-7/SC-3/AH-3*, pages 8–16, 2001.

[2] K. M. Anderson, S. A. Sherba, and W. V. Lepthien. Towards Large-Scale Information Integration. In *Proceedings of the 24th International Conference on Software Engineering*, pages 524–535, Orlando, FL, USA, May 2002.

[3] K. M. Anderson, R. N. Taylor, and E. J. Whitehead, Jr. Chimera: Hypermedia for Heterogeneous Software Development Environments. *ACM Transactions on Information Systems*, 18(3):211–245, July 2000.

[4] 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, October 2002.

[5] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1995.

[6] H. C. Davis, S. Knight, and W. Hall. Light Hypermedia Link Services: A Study of Third Party Application Integration. In *Proceedings of the Sixth ACM Conference on Hypertext*, pages 41–50, Edinburgh, Scotland, September 1994.

[7] J. Han. Specifying the Structural Properties of Software Documents. In *Proceedings of the 1994 International Conference on Computing and Information*, pages 1333–1351, 1994.

[8] J. Han. TRAM: A Tool for Requriements and Architecture Management. In *Proceedings of the Australasian Computer Science Conference*, Gold Coast, Queensland, Australia, 2001.

[9] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[10] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, January 2002.

[11] Microsoft®Word®. http://www.microsoft.com/office.

[12] K. Østerbye and U. K. Wiil. The Flag Taxonomy of Open Hypermedia Systems. In *Proceeding of the Seventh ACM Conference on Hypertext*, pages 129–139, Washington, DC, USA, 1996.

[13] J. D. Palmer. Traceability. In R. H. Thayer and M. Dorfman, editors, *Software Requirements Engineering, Second Edition*, pages 412–422. IEEE Computer Society Press, 2000.

[14] K. Pohl. *Process-Centered Requirements Engineering*. Research Studies Press Ltd., 1996.

[15] K. Pohl, M. Brandenburg, and A. Gülich. Integrating Requirements and Architecture Information: A Scenario and Meta-Model Based Approach. In *Proceedings of the Seventh International Workshop on Requirements: Foundation for Software Quality (REFSQ'01)*, Interlaken, Switzerland, 2001.

[16] B. Ramesh and M. Jarke. Toward Reference Models of Requirements Traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, January 2001.

[17] E. J. Whitehead, Jr. An Architectural Model for Application Integration In Open Hypermedia Environments. In *Proceedings of the Eighth ACM Conference on Hypertext*, pages 1–12, Southhampton, UK, April 1997.

[18] U. K. Wiil and J. J. Leggett. The HyperDisco Approach to Open Hypermedia Systems. In *Proceedings of the Seventh ACM Conference on Hypertext*, pages 140–148, Washington DC, USA, March 1996.