14 Tailoring Software Traceability to Value-Based Needs

Alexander Egyed

Abstract: Software development generates and maintains a wide range of artifacts, such as documentation, requirements, design models, and test scenarios; all of which add value to the understanding of the software system. Trace dependencies identify the relationships among these artifacts. They contribute to the better understanding of a software system as they link its distributed knowledge. Trace dependencies are also vital for many automated analyses including the impact of change and consistency checking. This chapter compares the problem of manual traceability versus automated traceability with the Trace/Analyzer approach. This chapter also explores how to tailor precision, completeness, correctness, and time-line to adjust the trace analysis to value-based needs.

Keywords: Traceability, software modeling, trace analysis, trade-off analysis, consistency, impact of change, change propagation, traceability uncertainties.

14.1 Introduction

Software development is a process that involves many stakeholders and generates a range of development artifacts. For example, the requirements are typically captured independently from the design/implementation although it has been recognized that there is a strong, intertwined relationship between them (Nuseibeh, 2001). The design, in turn, is often refined stepwise over several layers to explore the complexity of subsystems. Each such subsystem or layer may be the explored structurally (i.e., class diagrams) and/or behaviorally (i.e., sequence or statechart diagrams) (Rumbaugh et al., 1999).

Handling artifacts independently benefits the concurrent software development (Boehm, 2003) because it separates concerns, reduces complexity, and allows engineers to work independently. However, these artifacts (e.g., requirements, design) must be linked together to understand their combined impact on the software system. Trace dependencies explicitly describe the relationships among such separately recorded artifacts.

In some form, every software artifact has “some relationship” to every other artifact. We thus define a trace dependency to specifically identify whether two, separately recorded artifacts have the same/similar meaning (i.e., since traces tend to bridge artifacts of different modeling notations it is typically not possible to capture the same/similar artifacts in a uniform manner). However, there are many potential trace dependencies and value-based software engineering (Boehm, 2003; Boehm and Huang, 2003) recognizes that it is not always meaningful to capture all
of them without understanding their value. While this chapter does not discuss how artifacts differ in their value, it does stipulate that the quality of trace dependencies should reflect the value of the artifacts they bridge (better quality traces for higher value artifacts). It is thus beneficial to customize traces in terms of their precision, completeness, correctness, and timeliness.

Trace analysis is the process of finding and validating trace dependencies among artifacts. While finding trace dependencies alone is not sufficient to reconcile multiple perspectives, they are the foundation for any such mechanism. Trace dependencies are vital for documentation, program understanding, impact analysis, consistency checking, reuse, quality assurance, user acceptance, error reduction, cost estimation, and customer satisfaction (Antoniol et al., 2002; Biffl and Halling, 2003; Pohl, 1996; Gotel and Finkelstein, 1994; Ramesh, 1993). Their absence usually inhibits automation. This chapter discusses how to generate and validate trace dependencies and how to customize this process to value-based needs. That is, not all traces are equally important and this chapter demonstrates how the trace analysis can be tailored to the importance of the artifacts they bridge. It must be noted that this chapter does not discuss the many uses of trace dependencies (besides some examples).

Not understanding trace dependencies has many negative side effects. Most significantly, it increases the risk that changes are not propagated correctly. And it causes errors in that engineers, ignorant or unaware of the inconsistencies, make decisions on inaccurate information.

Trace analysis is well motivated in value-based software engineering (Boehm, 2003) due to the need to evolve the system and software concurrently. Concurrent engineering implies that changes can happen anytime and anywhere and traces help the engineer in identifying the impact of those changes across all development artifacts (e.g., requirements, design, and implementation). Traces are also vital for value-based monitoring and control (Boehm, 2003) because the engineer needs to understand the mapping between goals and solution. This value benefit has been recognized in the past as there are many standards that mandate trace analysis as a required activity (e.g., DOD Std 2167A, IEEE Std. 1219, ISO 15504, and SEI CMM).

On the downside, trace analysis is a complex activity. Standards encourage trace analysis but they generally do not tell how to do it (Lindvall, 1994; Lindvall and Sandahl, 1996). Also, existing tool support is typically limited to the recording of trace dependencies but not to their identification (Antoniol et al., 2002) (i.e., traceability matrix). As a result, thorough trace analysis is a predominantly manual activity (Card, 1992) that has to cope with many complexities:

- **Non-scalable growth:** up to $n^2$ trace dependencies for $n$ artifacts (Antoniol et al., 2002; Card, 1992)
- **Syntactic and semantic differences:** hard to identify traces exactly (Övergaard, 1998; Jacobson, 1987).
- **Informal/semiformal languages** (e.g., requirements, UML design): artifacts are described imprecisely (Finkelstein et al., 1991) and cause trace uncertainties (Egyed, 2004).