The Journal of Systems and Software 108 (2015) 178–192

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

# Exploiting traceability uncertainty among artifacts and code

Achraf Ghabi[*], Alexander Egyed

*Johannes Kepler University, Altenbergerstraße 69, 4040 Linz, Austria*

ARTICLE INFO

*Article history:*
Received 14 April 2014
Revised 29 May 2015
Accepted 16 June 2015
Available online 26 June 2015

*Keywords:*
Traceability
Artifacts to code mapping
Analysis

ABSTRACT

Traceability between software development artifacts and code has proven to save effort and improve quality. However, documenting and maintaining such traces remains highly unreliable. Traceability is rarely captured immediately while artifacts and code co-evolve. Instead they are recovered later. By then key people may have moved on or their recollection of facts may be incomplete and inconsistent. This paper proposes a language for capturing traceability that allows software engineers to express arbitrary assumption about the traceability between artifacts and code – even assumptions that may be inconsistent or incomplete. Our approach takes these assumptions to reasons about their logical consequences (hence increasing completeness) and to reveal inconsistencies (hence increasing correctness). In doing so, our approach's reasoning is correct even in the presence of known inconsistencies. This paper demonstrates the correctness and scalability of our approach on several, large-scale third-party software systems. Our approach is automated and tool supported.

## 1. Introduction

Traceability is very important during software development, especially for change impact analysis (Briand et al., 2006; Mäder and Egyed, 2011) during the maintenance stage (Haumer et al., 1999). Empirical evidence suggests that requirements to code traces can make bug fixes and features extensions 20–30% faster and over 50% more correct (Briand et al., in press; Mäder and Egyed, 2011). These benefits are substantial and accentuate that traceability should play a major role in the software engineering life cycle. Existing commercial tools typically support the recording of traces but not necessarily their creation or maintenance.

It is presumed that software engineers 'know' the traces between software artifacts (e.g., requirements or model elements) and code. Existing tools merely record them (Egyed and Grunbacher, 2002) – typically using a trace matrix (TM) that cross-reference artifacts at the level of granularity the engineers chose (e.g., requirements to classes vsrequirements to methods traces). The engineers' job is to manually fill in the fields of the matrix by deciding for each cross-reference whether or not the element on the one side, say a requirement, is implemented by the element on the other side, say a method. A trace matrix thus reveals that traceability is of quadratic complexity: $a*c$ for $a$ artifacts (e.g., requirements) and $c$ code elements (e.g., methods). Each cell in a trace matrix requires a non-trivial, human decision. Consider, for example, the Gantt Project system (GAN, 2014) (one of our study systems) with hundreds of artifact elements and thousands of Java methods. A complete traceability matrix for the Gantt Project system requires tens of thousands of decisions; one for every model element/Java method pair. The scalability implication is daunting (Bianchi et al., 2000). Once established, the traceability must be kept up-to-date while the software artifacts and/or the code changes (Clarke et al., 1999) – to remain consistent and useful.

Yet, traceability cannot be captured or maintained by a single engineer because in any complex engineering effort engineers have partial knowledge only. Traceability is thus a collaborative process that involves many engineers. Moreover, traceability is a mostly manual process (automation are mostly limited to information retrieval discussed later). Given that traceability is also of non-linear complexity, it should not surprise that there is never a guarantee of correctness or completeness. Naturally this is a problem because the aforementioned studies on the benefits of traces (Briand et al., in press; Mäder and Egyed, 2011) presume correctness and completeness. Now consider that today most engineering projects do not even capture traceability (upfront). Rather, they capture it at later stages (after system completion) or never in which case this knowledge remains in the heads to the engineers who built the system. Unfortunately, during the development of a system and after its delivery to the client, key personnel may move on. Even if they stay, it is well documented that the engineers' recollection of artifacts and code fades over time – and with it the memory of traceability (Gotel and Finkelstein, 1994). However, it is exactly here that traceability is most needed.

[*] Corresponding author. Tel.: +4373224684388.
*E-mail addresses:* a@ghabi.net (A. Ghabi), alexander.egyed@jku.at (A. Egyed).

http://dx.doi.org/10.1016/j.jss.2015.06.037
0164-1212/© 2015 Elsevier Inc. All rights reserved.

Explicit traceability capture is thus a pre-requisite to principled software engineering. This paper introduces a language and approach that allows engineers to express traceability at any level of detail, completeness, certainty, and correctness. An example of a traceability uncertainty is if the engineer remembers that a given requirement is implemented in some set of classes but not exactly which ones of them. It would be wrong for a trace capture tool to force a precise input from an engineer in the face of such uncertainty. Yet, if multiple engineers input partially uncertain traceability then it is possible to combine this knowledge for a more complete understanding. We will demonstrate that it is possible to automatically reduce, even resolve, some uncertainty by automatically inserting logical consequences of the input provided by the engineers. As this example shows, our approach is most useful for situations where multiple engineers provide input about traceability. Yet, traceability provided by different engineers may not be consistent. We will demonstrate that it is possible to automatically identify incorrectness where the input provided by engineers is contradictory. But most significantly, we will demonstrate that this automation is correct even in the presence of inconsistent input.

This paper combines our findings from three conference papers where we described the traceability language for model-to-code traceability (Egyed, 2004), an effective reasoning mechanism that is able to check correctness (Ghabi and Egyed, 2012), and managing inconsistencies in SAT problems with HUMUS (Nöhrer et al., 2012). The added value is in (a) providing a scalable, precise basis for reasoning based on SAT solvers; (b) more numerous and larger empirical evaluations; (c) a broadened scope that covers requirements, model elements and code; and (d) the integration of HUMUS and SAT for correct reasoning in context of potentially inconsistent traceability.

## 2. Illustration

We use the illustration of a video-on-demand system (VOD) (Dohyung) throughout this work to explain many of the uncertainty and incompleteness issues that characterize artifact-to-code traceability. In Fig. 1 we depict a state transition diagram on the left side and a table of requirements on the right side. The state transition diagram models the behavior of the VOD system. The table of requirements on the right side of Fig. 1 is an abbreviated documentation of the requirements implemented in VOD. Together, these two diagrams depict the many artifacts that engineers may want to trace to the code. For example, each requirement (i.e. row) in the table is an artifact that should be implemented somewhere in the code. The same is true to for the state transitions. For the sake of brevity we will be referring to the requirements and state transitions by their IDs: e.g., R1 or S4.

The VOD is a real albeit smaller system implemented in Java. For the sake of brevity we abstract the implementation into five pieces of code – labelled by their short acronyms {A, B, C, D, E}. Each of them stands for a set of Java classes.

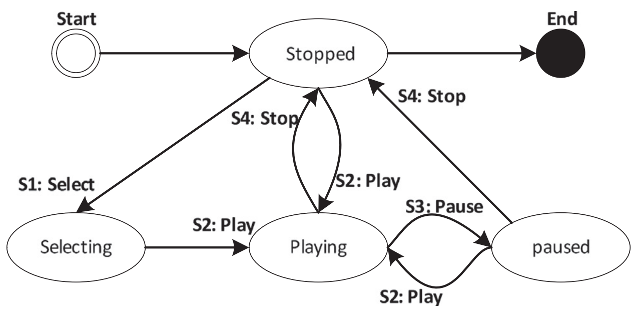
## 3. Artifacts and code relationships

While it is common that engineers create and use artifact descriptions, it is still not common to document where exactly each artifact is implemented in the source code or how it is related to other software development artifacts. Knowing about traceability is important for understanding complex systems and understanding the impact of a change (e.g., if a part of the requirements changes how would it impact the implementation?). The goal of this work is to help the engineer explore this kind of relationship between software development artifacts and the code. A software development artifact could be any common artifact used during the development and/or maintenance of a software project such as UML model, use cases, or requirements definition.

We refer to a piece of source code as a code element where the granularity of the code element is entirely user-definable. A code element could be a line of code, a method, a class, a package, or any other logical grouping (e.g., architectural component). We will discuss the implications of different granularity choices later. We presume that the code elements are disjoint in that the same line of code may not belong to more than one code element.

We refer to individual requirements, states transitions, etc as artifact elements. Here also the granularity is arbitrary user definable. For example, we could trace the entire state transition diagram to code or we could trace its individual states and transitions. The relationship between artifact elements and code elements is bidirectional. We expect that a single artifact element is implemented in multiple code elements (one-to-many mapping) because artifact elements are typically higher-level descriptions of the implementation of the system. Hence they are expected to require larger amounts of code to implement them. However, a single code element may also implement multiple artifacts (particularly, if the granularity of code elements if coarse). Moreover, it is not correct to assume that every code element must implement an artifact element. This assumption is true only if the artifact elements describe the entire software system. Artifacts (e.g. models) can be incomplete either by choice or by omission. For example, the state transition diagram in Fig. 1 is by no means complete and hence not all code will trace to it.

Fig. 1 includes a state transition diagram (which is a behavioral model) and it includes also a list of requirements. Those artifacts provide independent perspectives onto a software system – we speak of multiple perspectives or views (Antoniol, 2001; Gotel and Finkelstein, 1994; Parnas, 1972). Each perspective describes the software system from a different point of view. For example, the state transition perspective describes the software system independently from the requirements but there are clearly overlaps. R4 about stopping the playback, for example, is also implemented in the state transition diagram through various transitions. Perspectives may be at different levels of abstraction (i.e., separating the structure from the behavior). A code element may thus implement artifact elements of different perspectives. For example, whatever code implements the stopping of a playback implements both the stop transitions and the stop requirement.



| Req. ID | Name of Requirement |
|---|---|
| R1 | Select a Movie |
| R2 | Play a Selected Movie |
| R3 | Pause Playing |
| R4 | Stop Playing |

**Fig. 1.** Illustration System: Video On Demand (VOD).