# On-Demand Automated Traceability Maintenance and Evolution

Muhammad Atif Javed, Faiz UL Muram, and Uwe Zdun

University of Vienna, Faculty of Computer Science,
Software Architecture Research Group, Vienna, Austria
{muhammad.atif.javed,faiz.ulmuram,uwe.zdun}@univie.ac.at

**Abstract.** After the painstaking process of traceability construction, a substantial evolution of a software system, such as a new major version leads to the decay of traceability links. To date, however, none of the published studies have considered the on-demand update of traceability links. This paper presents an on-demand automated approach for case-based maintenance and evolution of traceability links in the context of different versions of a software project. The approach focuses on the component-to-component features for identification and prioritization of previous traceability cases, which are then used to perform reuse and adaptation of traceability links based on the matches and mismatches, respectively. The adapted (i.e., newly constructed) traceability links can then be verified by a human analyst and stored in a case base. The approach has been validated using an open-source framework for mobile games, named Soomla Android store.

## 1 Introduction

The constructed traceability links need to be maintained continuously or on-demand as a project evolves so that up-to-date traceability links are available when needed. The need for continuous traceability maintenance is triggered by changes to any of the software artefacts (like any architecture component) that, in turn, can be triggered by changes to artefacts within a traceability chain (e.g., underlying requirements and the code classes that implement the component). The semi-automatic support for such maintenance is achieved in some existing approaches [2, 6, 10]. However, continuous traceability maintenance might not be a feasible solution in case of a substantial evolution of a software system, such as a new major version in large real-world project in which hundreds of developers are involved, maybe even at different distributed locations. For such new major versions, traceability links are subject to and undergo reconstruction in general.

The proposed on-demand traceability maintenance and evolution approach consists of four phases: First, the pre-existing traceability links need to be organized as cases. Second, the extracted features of a new component are used to identify and prioritize the stored traceability cases. Third, the new requirements and code classes are matched with the requirements and code classes in previously implemented components. The links concerning the matched requirements and code classes are reused; however, adaptation is performed for mismatches between the requirements, architectural components and code classes. Finally, the adapted (i.e., newly constructed) links can be verified by a

human analyst and stored in the dedicated case base for future use. The feasibility of the proposed approach is demonstrated by maintaining and evolving the traceability links of the Soomla Android store[1] Version 2.0 to the Version 3.6.17.

The rest of this paper is organized as follows: Section 2 discusses the related work. Section 3 describes the on-demand maintenance and evolution of traceability links. Section 4 presents a case study. Section 5 compares the results of the proposed approach to an information retrieval methods based tool. Section 6 concludes the paper and discusses future work.

## 2 Related Work

Hammad et al. [6] developed a tool, called SrcTracer that supports traceability from the source code to the design to maintain consistency with the design during code evolution. The approach examines the source code changes based on a lightweight analysis and syntactic differentiation to evaluate whether a particular change alters the design or not. The changes in the source code that lead to design changes include adding or removing of classes, methods and their relationships. Cleland-Huang et al. introduce a concept for the identification of change types that are applied to requirements in their event-based traceability approach [3]. The authors capture seven types of change activities to requirements, in particularly create, inactivate, modify, merge, refine, decompose and replace. However, the approach concentrates more on the manual construction of traceability links instead of maintaining them.

Mäder et al. [10] introduce a semi-automatic strategy, called traceMaintainer to determine changes in UML models to update pre-existing traceability relations. The approach records all changes to model elements and uses this information to find a match within a set of predefined patterns of recurring development activities. Buchgeher and Weinreich [2] introduce the LISA approach that captures traceability relations through observing the developer as she/he is working on the architecture design and implementation. These approaches, however, focus on the continuous maintenance of traceability links that might not be feasible when a large real-world project substantially evolve (i.e., a new major version). The research in Borg et al. [1] focus on the extraction of traceability information that has been specified as a by-product of completing impact analyses reports by using history mining and storage of extracted traceability information in a semantic network. They have not considered the use of extracted traceability as a way to either update or validate the traceability record.

## 3 Building a Case-Based Reasoner for Traceability Maintenance and Evolution

An overview of the proposed on-demand traceability maintenance and evolution approach is shown in Figure 1. It describes the procedure used for case-based representation of software traceability (Section 3.1), similarity assessment and retrieval of stored traceability cases (Section 3.2), reuse and adaptation of traceability links at the architectural level (Section 3.3), and revision and retention of traceability links (Section 3.4).

[1] http://soom.la/.

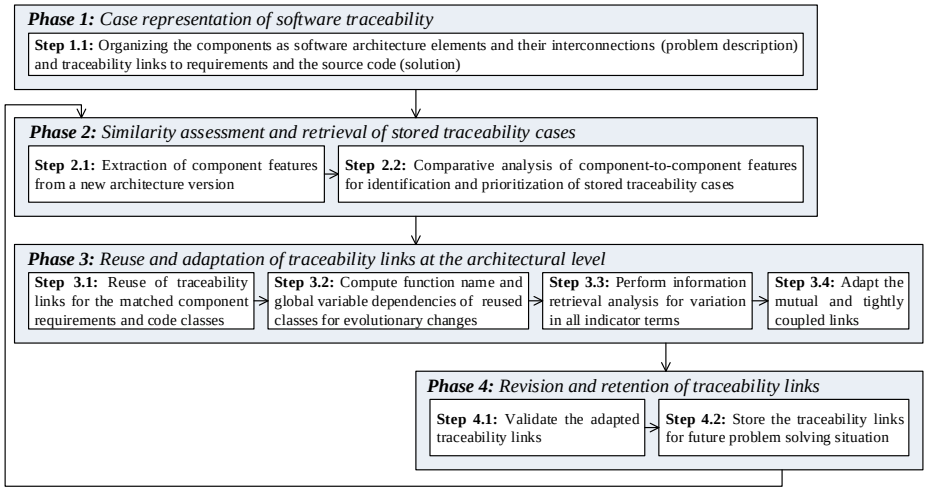


**Fig. 1:** The steps of the on-demand traceability maintenance and evolution approach

### 3.1 Case Representation of Software Traceability

In case-based reasoning paradigm, a problem situation is organized as a case which consists of two parts, as shown in Listing 1.1: problem description and solution. The former describes the components as software architecture elements and their interconnections, while the later contains the traceability links to artefacts produced in the other activities of the development process, such as requirements and implementation.

The idea with the problem description part is the selection of substitutable component(s) among a variety of candidate components that might solve a given problem. The information about an architectural component might be enclosed inside particular XML tags and attributes. The component identification number distinguishes a component from others components, the name represents a name of the component and the description specifies functionalities offered by a component. A port describes an interaction point; it would have provided or required interfaces that provide specific information about the services offered or required by a component. A connector defines a pathway of interaction between components and have source and target ends. The solution part of a case comprises of traceability links to the underlying requirements that can be distinguished by an identification number and the code classes that implement the particular requirement within the component. For efficient reuse and adaptation, it was decided to separate each requirement and its corresponding implementation classes so that unaffected traceability links shall be avoided from adaptation.

**Listing 1.1:** Features and traceability links of the StoreAssets component in the Soomla Android store Version 2.0

```
<Case ID = "C53">
  <Problem Part>
    <Feature List>
      <Component ID="SC1" Name="StoreAssets" Description="The StoreAssets component impleme-
          nts the virtual currencies, virtual goods, as well as their classification and pr-
          ice models." />
      <Port Name="Assets" Kind="Prov" Interfaces="" />
```

```
    <Port Name="AssetsInfo" Kind="Prov" Interfaces="" />
    <Connector Source="Assets" Target="ControlAssets" TargetComponent="StoreController" />
    <Connector Source="AssetsInfo" Target="InfoStorage" TargetComponent="DatabaseServices"
        />
  </Feature List>
 </Problem Part>
 <Solution Part>
  <Trace Links>
    <Requirement ID="1.2.0" Description="The SOOMLA Android store supports consumable ite-
       ms. The user is expected to consume virtual goods and repurchase them. Tokens, co-
       ins and gems are some examples of virtual currencies. To purchase the virtual goo-
       ds, the static and balance driven price models should be implemented. After some
       time, when the virtual currency is insufficient, the user would have to purchase a
       virtual currency pack, such as 10 coins pack or 20 coins pack. The pack holds the
       virtual currency and its corresponding price, i.e., the cost of the pack." />
    <Code Classes="VirtualCurrency.java, VirtualCurrencyPack.java, VirtualItemNotFoundExc-
       eption.java, AbstractPriceModel.java, StaticPriceModel.java, BalanceDrivenPriceMo-
       del.java, JSONConsts.java" />
  </Trace Links>
  <Trace Links>
    <Requirement ID="2.2.0" Description="SOOMLA also supports non-consumable items that are
       expected to last forever. This type of goods will be used to implement additional
       levels, a remove ads feature, or upgrading to a premium version of the game." />
    <Code Classes="NonConsumeableItem.java, VirtualGood.java, VirtualCategory.java, Google-
       MarketItem.java, AbstractVirtualItem.java" />
  </Trace Links>
 </Solution Part>
</Case>
```

### 3.2 Similarity Assessment and Retrieval of Stored Traceability Cases

The assessment of component-to-component features for case retrieval includes two steps. First, all the components together with their features are extracted from the given software architecture. Second, the stored cases are traversed in order to find the similar cases, i.e., a set of candidate components whose characteristics/features match with the new architectural component.

**Algorithm 1** Algorithm for retrieval of traceability cases

**Input:** A new case-problem $Cpn$ would have a set of features $f = \{f_1, f_2, ..., f_n\}$, where $n$
refers to the number of new component features
**Output:** The degree of similarity between the versions of an architectural component
**Local variables:** The component features $f' = \{f'_1, f'_2, ..., f'_m\}$ within in the problem part of
a previous case $Cpr$, where $m$ concerns the number of features
The weight factor $W$ is assigned to the components' features so that $w = \{w_1, w_2, ..., w_n\}$
**Begin**
  start from $i = 1$ –> select one case $Cpn_i$ $(0 \leq j \leq m)$
  **Begin**
    for each $f'_j$ of $Cpr$ do $(0 \leq j \leq n)$
    **Begin**
      compare individual features $f$ and $f'$:
      do Equation 2
    **End**
    calculate the similarity of overall features using Equation 1
  **End**
**End**

Let $Sim1$ and $Sim2$ be the similarity values of two previous cases $Cpr1$ and $Cpr2$, respectively. Of course, for a new case problem $Cpn$, a case $Cpr1$ is more preferable

if $Sim1$ is higher than $Sim2$, because $Cpr1$ is more similar to the new case problem. The global similarity *GSIM* (Equation 1) concerns the overall features while the local similarity *LSIM* (Equation 2) focuses on matching $f_i$ ($ith$ feature) of a new component with the $f'_j$ ($jth$ feature) of a previous component incorporated in a traceability case. It was decided to assign the similarity weights ($W_i$) ranging from 1.0 – 3.0. In particular, the highest weight is assigned to component interfaces or otherwise its description; whereas the component name, port name and port kind are weighted 2.0, 1.5 and 1.0, respectively. Algorithm 1 illustrates the case retrieval mechanism.

$$GSIM([f_1, f_2, ..., f_n][f'_1, f'_2, ..., f'_m]) = \sum_{i=1}^{n} \sum_{j=1}^{m} W_i . LSIM(f_i, f'_j) \tag{1}$$

$$LSIM(f, f') = \begin{cases} 1, & \text{if } f = f' \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

### 3.3 Reuse and Adaptation of Traceability Links at the Architectural Level

The artefacts produced in the other activities of the development process, such as requirements and source code are matched with the solution part of retrieved traceability case(s) that underlies requirements and code classes implemented in a previous component. If component requirements and code classes are matched, the relevant traceability links are reused. However, transformation adaptation is performed for mismatches between the requirements, architectural components and the code classes.

Based on our experiences and observations from various empirical investigations and open source software systems, we have noticed that a main driver for trusted and cost-effective traceability construction is the achievement of highest precision for the initial traceability links [7, 9]. It is therefore decided to use the unchanged traceability links, if available, as an active countermeasure to arbitrarily making traceability decisions and to maintain and preserve the trust in further traceability construction [8]. The reuse and adaptation is performed in four steps: (i) the links for matched parts are reused, (ii) the function name and global variable dependencies of reused classes are computed, (iii) the information retrieval analysis based on the indicator terms for variation in architectural components, requirements and undetected classes is performed, and (iv) the mutual and tightly coupled classes are linked with the corresponding development artefacts. The availability of reused links demonstrated better results for on-demand traceability maintenance and evolution.

### 3.4 Revision and Retention of Traceability Links

The validation by a human analyst is strongly required in automated traceability tools. Nevertheless, the problems in automated traceability might not be completely eliminated when extensive set of traceability links are candidates for validation [5]. The focus on reuse and adaptation of traceability links would reduce the burden on a human analyst, in particularly the already verified traceability links from the past (i.e., reused links) in both matched and partially matched cases might be omitted from validation.

However, only adapted traceability links for evolutionary changes would be made available to human analyst for validation. The revised solution part is combined with the problem description part in order to be stored as a new case so that the new traceability case becomes available for future problem solving situation.

## 4 Case Study

The Soomla Android store allows mobile game developers to easier implement virtual currencies (e.g., tokens, coins, gems), virtual goods and in-app purchases. Due to the added support for virtual goods, rewards, store events and payment mechanisms, the migration from older to later versions ($\geq 3.6.X$) of the Soomla Android store is recommended. The complete maintenance and evolution of traceability links cannot be discussed due to space limitations and similar technical details. Therefore, the rest of this section describes one of the affected component, named as StoreAssets.

**Listing 1.2:** Upgradtion of previously constructed traceability links of the StoreAssets component to the Version 3.6.17

```
<Case ID="75">
  <Problem Part>
    <Feature List>
      <Component ID="SC1" Name="StoreAssets" Description="The StoreAssets component implemen-
          ts the virtual currencies, virtual goods, rewards as well as their classification."
          />
      <Port Name="Assets" Kind="Prov" Interfaces="" />
      <Port Name="AssetsInfo" Kind="Prov" Interfaces="" />
      <Connector Source="Assets" Target="ControlAssets" TargetComponent="StoreController" />
      <Connector Source="AssetsInfo" Target="InfoStorage" TargetComponent="DatabaseServices"
        />
    </Feature List>
  </Problem Part>
  <Solution Part>
    <Trace Links>
      <Requirement ID="1.3.6.17" Description="The SOOMLA Android store supports consumable
          items. The user is expected to consume virtual goods and repurchase them. Tokens,
          coins and gems are some examples of virtual currencies. After some time, when the
          virtual currency is insufficient, the user would have to purchase a virtual curr-
          ency pack, such as 10 coins pack or 20 coins pack. The pack holds the virtual cu-
          rrency and its corresponding price, i.e., the cost of the pack. In addition, the
          user should be able to earn a specific reward after achieving certain criteria in
          game progress." />
      <Code Classes="VirtualCurrency.java, VirtualCurrencyPack.java, VirtualItemNotFoundEx-
          ception.java, BadgeReward.java, RandomReward.java, Reward.java, SequenceReward.-
          java, VirtualItemReward.java, Schedule.java, SoomlaEntity.java, JSONConsts.java,
          JSONFactory.java" />
    </Trace Links>
    <Trace Links>
      <Requirement ID="2.3.6.17" Description="SOOMLA also supports non-consumable items that
          are expected to last forever. This type of goods will be used to implement additi-
          onal levels, a remove ads feature, or upgrading to a premium version of the game."
          />
      <Code Classes="VirtualGood.java, VirtualItem.java, VirtualCategory.java, EquippableVG.-
          java, LifetimeVG.java, SingleUseVG.java, SingleUsePackVG.java, UpgradeVG.java, Mark-
          etItem.java, PurchasableVirtualItem.java" />
    </Trace Links>
  </Solution Part>
</Case>
```

The adaptation of traceability links is performed in three steps. First, the function name and global variable dependencies of reused classes are computed. This led to the

identification of eleven classes for the first underlying requirement. In case of second requirement, five classes are identified out of which four classes are strongly linked as means of «extends» relationship. Second, the information retrieval analysis based on the indicator terms is performed, which leads to the identification of nine and twelve classes for the requirements, respectively. The variation in first requirement text covers the variation in component description. Besides that, the undetected classes are not used for adaptation as the indicator terms in excluded requirement text are matched with the deleted classes. The mutual terms, if available, would be used for the recovery. Finally, the mutual and tightly coupled classes are linked with the requirement in Version 3.6.17, as shown in Listing 1.2. The adaptation process correctly identified all the classes realizing the particular requirement within a StoreAssets component.

## 5 Experimental Evaluation

In circumstances of a new major version, the information retrieval techniques are commonly used to reconstruct the traceability links. As means of comparison, the traceability links for Soomla Android store Version 3.6.17 are reconstructed using an information retrieval based tool, called Traceclipse[2]. Particularly, the latent semantic indexing is used for traceability construction.

| Factors | On-Demand Maintenance and Evolution (OME) | | | | Traceclipse Tool Weight >=0.1 (TWF) | | | Traceclipse Tool All Links (TAL) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Reused | Adapted | Irrelevant | Missed | Generated | Irrelevant | Missed | Generated | Irrelevant | Missed |
| StoreAssets | 6 | 16 | 0 | 0 | 84 | 67 | 5 | 170 | 149 | 1 |
| StoreController | 11 | 43 | 8 | 1 | 24 | 13 | 36 | 85 | 38 | 0 |
| DatabaseServices | 4 | 3 | 0 | 0 | 40 | 37 | 4 | 85 | 78 | 0 |
| Billing | 0 | 43 | 36 | 0 | 32 | 27 | 2 | 83 | 76 | 0 |
| CryptDecrypt | 3 | 1 | 0 | 0 | 8 | 7 | 3 | 84 | 83 | 3 |

**Table 1:** Summary of achieved results for the Soomla Android store Version 3.6.17

| | Recall | | Precision | | F-measure | |
|---|---|---|---|---|---|---|
| | OME vs. TWF | OME vs. TAL | OME vs. TWF | OME vs. TAL | OME vs. TWF | OME vs. TAL |
| Cliff's delta | -0.5158 | -0.1548 | -0.5995 | -0.6319 | -0.6019 | -0.5859 |
| Cohen's delta | -2.8780 | -0.6614 | -2.1561 | -2.1100 | -2.6114 | -2.0095 |
| Standard deviation of delta | 0.1792 | 0.2340 | 0.2780 | 0.2994 | 0.2304 | 0.2915 |
| z/t score of delta | -4.5505 | -1.0458 | -3.4091 | -3.3362 | -4.1290 | -3.1774 |
| Confidence interval low | -0.8301 | -0.5656 | -1.0435 | -1.0863 | -0.9867 | -1.0131 |
| Confidence interval high | -0.2014 | 0.2559 | -0.1555 | -0.1774 | -0.2170 | -0.1587 |
| Degrees of freedom | 4.0112 | 4.0066 | 5.3214 | 6.5339 | 4.5955 | 7.7932 |
| p-value | **0.0103** | 0.3545 | **0.0172** | **0.0138** | **0.0108** | **0.0134** |

**Table 2:** Cliff's $\delta$ Test for the Soomla Android store Version 3.6.17

Table 1 summarizes the achieved results for the Soomla Android store. For statistical analysis of the retrieved results, the Cliff's $\delta$ [4] – a robust non-parametric test – is used to evaluate the significance of the found results. The results from the Cliff's $\delta$ test are shown in Table 2. It is noticeable to see that significant differences emerged for all comparisons except with the recall of all traceability links. The main reason behind particular result is the rather higher number of generated links by the Traceclipse tool. The results for the Soomla Android store Version 3.6.17 provide strong evidence that our on-demand maintenance and evolution approach improved the precision and f-measure of traceability links for the new version of a software project, compared to an IR-based tool called Traceclipse.

[2] http://www.cs.wm.edu/semeru/traceclipse/

## 6 Conclusions and Future Work

This paper has proposed an on-demand traceability maintenance and evolution approach in which the component-to-component features are used for identification and prioritization of stored traceability cases, which are then used to perform reuse and adaptation of traceability links based on the matches and mismatches, respectively. The previous traceability links for the exactly matched parts are not only reused, but also not considered for validation by a human analyst. However, the adapted solution (i.e., newly constructed traceability links) can be verified by a human analyst and stored in the case base for future problem solving situations. The proposed approach demonstrated reliable results than the information retrieval based tool, called Traceclipse. In the future, our plan is to build a catalogue of guidelines as best practices for traceability reuse and adaptation across projects, organizations, domains, product lines and supporting tools.

## References

1. M. Borg, O. C. Z. Gotel, and K. Wnuk. Enabling traceability reuse for impact analyses: A feasibility study in a safety context. In *7th International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '13, pages 72–78, 2013.
2. G. Buchgeher and R. Weinreich. Automatic tracing of decisions to architecture and implementation. In *9th Working IEEE/IFIP Conference on Software Architecture*, WICSA '11, pages 46–55, 2011.
3. J. Cleland-Huang, C. K. Chang, and Y. Ge. Supporting event based traceability through high-level recognition of change events. In *26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, COMPSAC '02, pages 595–600, 2002.
4. N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *The Psychological Bulletin*, 114:494–509, 1993.
5. A. Dekhtyar, O. Dekhtyar, J. Holden, J. Hayes, D. Cuddeback, and W.-K. Kong. On human analyst performance in assisted requirements tracing: Statistical analysis. In *19th IEEE International Requirements Engineering Conference*, RE '11, pages 111–120, 2011.
6. M. Hammad, M. L. Collard, and J. I. Maletic. Automatically identifying changes that impact code-to-design traceability during evolution. *Software Quality Control*, 19(1):35–64, Mar. 2011.
7. M. A. Javed, S. Stevanetic, and U. Zdun. Cost-effective traceability links for architecture-level software understanding: A controlled experiment. In *24th Australasian Software Engineering Conference*, ASWEC ' 15 Vol. II, pages 69–73, 2015.
8. M. A. Javed, S. Stevanetic, and U. Zdun. Towards a pattern language for construction and maintenance of software architecture traceability links. In *21st European Conference on Pattern Languages of Programs*, EuroPlop '16, pages 24:1–24:20, 2016.
9. M. A. Javed and U. Zdun. On the effects of traceability links in differently sized software systems. In *19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, pages 11:1–11:10, 2015.
10. P. Mäder and O. Gotel. Towards automated traceability maintenance. *J. Syst. Softw.*, 85(10):2205–2227, Oct. 2012.