ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links

Leonardo G. P. Murta* André van der Hoek** Cláudia M. L. Werner*

*Federal University of Rio de Janeiro
COPPE - System Eng. and Computer Science
P.O. Box 68511
Rio de Janeiro, RJ 21945-970 Brazil
Phone: +55(21) 2562-8675
{murta, werner}@cos.ufrj.br

**University of California, Irvine
Department of Informatics
444 Computer Science Building
Irvine, CA 92697-3440 USA
Phone: +1(949) 824-6326
andre@ics.uci.edu

Abstract

Traditional techniques of traceability detection and management are not equipped to handle evolution. This is a problem for the field of software architecture, where it is critical to keep synchronized an evolving conceptual architecture with its realization in an evolving code base. ArchTrace is a new tool that addresses this problem through a policy-based infrastructure for automatically updating traceability links every time an architecture or its code base evolves. ArchTrace is pluggable, allowing developers to choose a set of traceability management policies that best match their situational needs and working styles. We discuss ArchTrace, its conceptual basis, its implementation, and our evaluation of its strengths and weaknesses in a retrospective analysis of data collected from a 20 month period of development of Odyssey, a large-scale software development environment. Results are promising: with respect to the ideal set of traceability links, the policies applied resulted in 95% precision at 89% recall.

1. Introduction

With the introduction of software architecture as a critical artifact in the software life cycle, a new problem has emerged: traceability between an architectural description and its corresponding source code must be maintained as they each evolve over time. Software architectures are currently used as a basis for run-time evolution [18], product selection in software product lines [6], new testing approaches [20], impact analyses [25], and numerous other activities that will not operate properly without a detailed and accurate mapping from an architectural description to relevant corresponding source code artifacts.

The scenario on which we focus in this paper consists of a conceptual architecture and its corresponding source code each evolving separately but needing to be accurately traced to each other. We know that an architecture simply is not static over time [12] and we certainly know that code evolves over time. Our objective is to make sure that, in the face of such evolution, proper traceability links amongst the two are maintained at all times so one can navigate from any version of any architectural element to its corresponding source code and, vice versa, can navigate from any version of a source code artifact and find in which version(s) of which architectural elements it is used.

To address this problem, we present a novel solution in the form of ArchTrace, a tool that relies on two critical observations: (1) rather than reconstructing traceability links after some significant amount of time has passed, it continuously updates traceability links in response to each and every change committed by a user, and (2) the specific update to be made is determined by an actively specified set of traceability management policies. The result is an approach that can be tailored to different user practices, takes advantage of the knowledge encoded in the policies, and accommodates incorporation of new policies.

The rest of this paper is organized as follows. Section 2 presents a motivating example to ground the ensuing discussion. Section 3 introduces the high level approach underlying ArchTrace, which is followed by a discussion of its implementation in Section 4. Section 5 evaluates the approach. Section 6 discusses related work and we conclude the paper in Section 7 with an outlook at our future work.

2. Motivating example

In this section, we provide an example that we will use throughout the paper to describe the features of ArchTrace. In this example, a simple architecture is defined for a word processing application. This architecture has three components: *Print*, *Toolbar*, and *Display*. All components exist in one version and the source code that implements these components is organized into three directories: *Model*, *View*, and *Controller*. These directories contain, respectively, *Printer.java* and *Action.java*, *EditingWindow.java*, and *CommandDispatcher.java*, as shown in the right hand side of Figure 1.

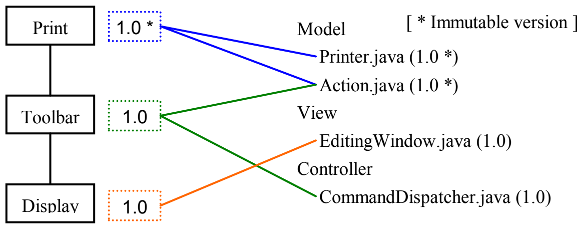


Figure 1: Initial scenario of the example

Figure 1 also shows that the first version of the *Print* component is immutable as it was already committed to the configuration management (CM) repository and can no longer be changed (unless, of course, a new version is created). Further, the first version of the *Print* component is implemented by two source files, *Printer.java* and *Action.java*; the first version of the *Toolbar* component is implemented by two source files, *Action.java* and *CommandDispatcher.java*; and the first version of the *Display* component is implemented by only one source file, *EditingWindow.java*. Note that the files that implement the *Print* component are also immutable.

The first step in our scenario consists of an architectural change, namely to create version 2.0 of the *Print* component. The new version inherits the traceability links of the previous version, which is expected since at this point nothing else has happened. Dashed lines represent these new traceability links in Figure 2.

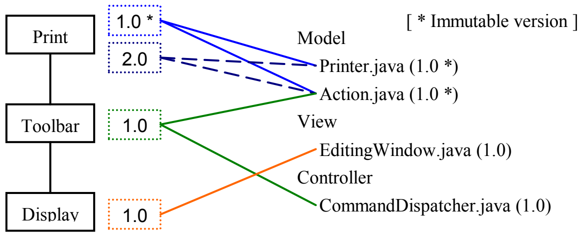


Figure 2: Component version creation

The second step consists of a series of changes to the code: (1) checking out *Action.java*, (2) modifying the checked out copy, (3) moving it to the *Controller* directory, and (4) in the process of checking in the new version, changing its name to *Command.java*. The set of traceability links should be updated accordingly. Specifically, architectural elements that used to link to version 1 of *Action.java* now should link to *Command.java*, which is version 2 since it represents an evolutionary step from *Action.java*. However, we should take into account the immutable state of the first version of the *Print* component. As an immutable version, its traceability links cannot be updated. Figure 3 shows the resulting set of traceability links. Two links, from the *Print* component (version 2.0) and *Toolbar* component (version 1.0), were redirected from *Action.java* to *Command.java*, and one traceability link, from version 1 of the *Print* component, was kept to point to *Action.java* due to immutability restrictions.

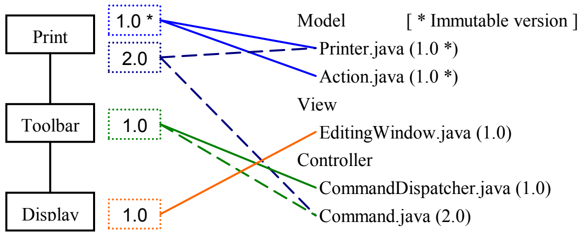


Figure 3: Expected scenario after the change

It is worth noting that, for illustration purposes, the example intentionally represents a simple scenario of evolving artifacts. It, however, provides concrete situations in which evolution of traceability links is difficult, even with automated tools: architectural versioning, immutability, renaming of artifacts, and selectively updating a set of traceability links.

3. Approach

The goal of ArchTrace is to support the evolution of already established traceability links. We are explicitly not concerned with creating an initial set of links, which tends to be the domain of the techniques of data mining [21, 24], information retrieval [4, 13], or syntactic analysis [7]. Generally speaking, the problem that we address in this paper can be stated as follows: given an initial set of traceability links, and given that both an architecture and its implementation can evolve independently, how can the traceability links be updated with the addition of new links, removal of existing links, and changes in existing links to ensure that each architectural element is at all times accurately linked to its corresponding source code artifacts (and vice versa)?

In support of this goal, we have designed our approach to consist of the following features: (1) a policy-based infrastructure, allowing the matching of policies to work practices; (2) policies that specifically take advantage of their knowledge of architectural and source code artifacts to make educated guesses on what to do upon architectural or source code change events; (3) policies that, when appropriate, request human input – but do so far less often than just maintaining all links manually; (4) and policies that act as either rules, deciding upon actions to take, or constraints, limiting the kinds of actions that can be taken.

In response to new "check in" events, we execute one or more policies that each governs a particular aspect of traceability link evolution. Policies are intentionally simple, each capturing one small behavior of traceability link evolution that matches potential actions that a user may take (a policy that deals with checking in a new architectural element, a policy that deals with removing a source file, a policy that deals with protecting immutable artifacts, etc.). Policies, thus, have a separate responsibility. But, because execution of one policy can result in the triggering of one or more other policies, the result is a set of closely collaborating policies that together are responsible for appropriately updating traceability links.

The policies are atomic elements that can be enabled and disabled individually. This is to support different work practices and different CM systems. Some developers establish certain practices on how to evolve their artifacts, and different CM systems establish different procedures [9]. Rather than attempting to build a single all-encompassing solution, we adopt a pluggable infrastructure that supports the addition of new policies (as long as they adhere to the programmatic interface of ArchTrace). A secondary, but as important benefit is that it becomes possible to disambiguate policies: when multiple policies are enabled, it is possible that multiple policies fire upon a check in. In some cases, this is desired, but in other cases it may be possible that conflicting policies are used or that certain policies apply to certain situations only (i.e., a developer may choose to use one set of policies during initial phases of development, when many new elements are added, and another set of policies during maintenance, when the set of elements stays relatively constant).

Our approach distinguishes four classes of policies: *architectural element evolution* policies, *implementation evolution* policies, *pre-trace* policies, and *post-trace* policies. Architectural element evolution policies fire when an architect makes modifications to an architecture, and implementation evolution policies fire when the source code evolves.

Pre-trace policies operate just before a new link is added or an old one is removed, acting as constraints. Their primary task is to detect the introduction of inconsistencies between the traceability link that is added or removed and the set of traceability links already existing. Should such an inconsistency arise, a pre-trace policy can veto the addition or removal, prohibiting the action to complete. An example of a pre-trace policy is one that prohibits changing links of immutable architectural elements: their traceability links generally should stay the same over time, so any suggested change should not be allowed.

Post-trace policies are executed after the creation or removal of traceability links has actually been completed. This allows the definition of policies that update additional traceability links when traceability links are added or removed. For example, when an architectural element needs to be updated with a newer version of a source file, an implementation element evolution policy adds the link, but a post-trace policy is responsible for removing the old link. This, in turn, may trigger other policies, in effect creating a rolling set of policies of different types that are executed.

Policies may request assistance from users; they are not meant to operate automatically or be "hidden" at all times. Rather, when it is pertinent that a user chooses one of two courses of action, or when additional human input is needed, a policy can leverage the interface of ArchTrace to get the input it needs. While, in our experience, it is relatively rare that this happens, it is critical to support this functionality. Should a "wrong" decision be made by a policy at some critical juncture, the set of traceability links can become significantly out of sync over time with those that actually should exist. Rather than automatically guessing an alternative, it is better to request user assistance. Note that the reason that this is relatively rare is because the users are involved in the selection of active policies in the first place: they already have selected a set of policies that describes how they operate and wish to be supported; only in exceptional circumstances will it be necessary to request clarification.

4. Implementation

ArchTrace is implemented in Java and assumes the use of xADL 2.0 [10] to describe software architectures and Subversion [8] to store source code. As we detail in the following, however, the architecture of ArchTrace is constructed to allow easy addition of other architectural tools and/or CM systems.

4.1. Overall architecture

Figure 4 presents the ArchTrace architecture. It consists of six components, four of which standard (shown as solid grey boxes) and two of which custom (shown as patterned boxes). The custom components depend on the particular architecture evolution environment and CM system used. As stated, we rely on xADL 2.0 and Subversion, but because the *Architecture Connector* and *Repository Connector* components are designed with abstract interfaces, the rest of ArchTrace is independent of the details of those two components.

Connector components insert tool-specific listeners. Upon receiving events (illustrated using dashed lines), they pass those on to the generic *Event Listening* component, which is responsible for interpreting the data contained in the events and invoking the appropriate part of the *Policy Triggering* component to begin the updating of traceability links.

The *Policy Triggering* component coordinates which specific policies are executed at what time in order to manage the set of traceability links and evolve them by adding and removing links. As discussed in Section 3, this kind of coordination is necessary because a policy may recursively trigger the execution of other policies, resulting in them together performing relatively complex tasks. For instance, in the specific case of the example in Section 2, the renaming and moving of a source file, a policy that updates the architectural element with the new link will trigger another policy that removes the older traceability link. Moreover, the policy that removes the older traceability link may trigger a third policy that prohibits this removal when the architectural element is marked as immutable.

Note that this architecture fully supports collaborative development. Because the CM system is responsible for resolving conflicts, perhaps with the help of the user performing some merges, traceability links simply evolve based on what is eventually checked in.

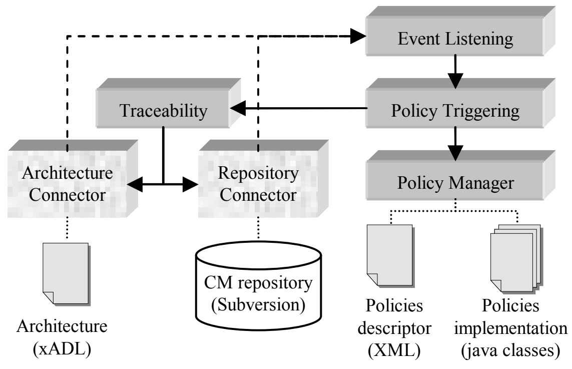
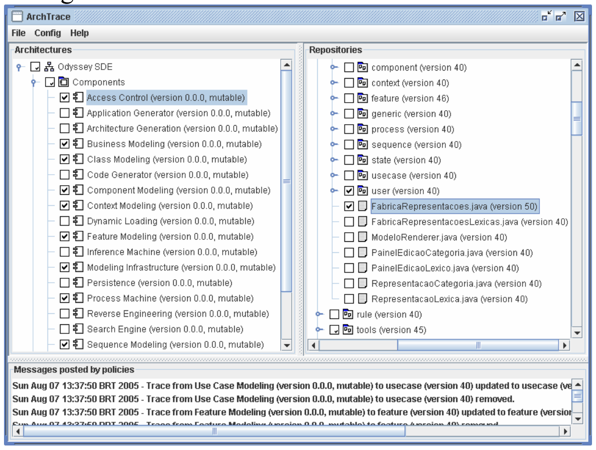


Figure 4: ArchTrace architecture

Actions that result in changes to the set of traceability links are actually enacted by the *Traceability* component. Since traceability links are typically stored either in the architecture description or in the CM system (by checking in a description of an architecture with the source code), this component is responsible for actually supporting the creation, removal, and querying of traceability links. It interacts with both the *Architecture Connector* and *Repository Connector* components to build upon their generic interfaces and operate independently.

Finally, the *Policy Manager* component is responsible for managing which policies are active at what time. During bootstrap of ArchTrace, this component loads all policies, instantiates them, and allows the user to activate and deactivate specific policies. Here is where the pluggability of ArchTrace comes into play: when new policies are created, these new policies, once loaded by this component, will act as any of the eight policies that we already built: they can be enabled, disabled, executed, triggered by other policies, etc.

It should be noted that, while ArchTrace typically operates in the background, it is possible for architects or developers to query ArchTrace at any time in the software development lifecycle to visualize the traceability links among architectural elements and their implementation. For instance, this kind of feature is essential for performing some activities such as impact analysis. Shown in Figure 5, ArchTrace allows exploration of the set of links: one can see all the links for a given architectural element or choose a file for which one wants to know to which architectural elements it belongs.



```
Odyssey SDE
  Components
    ☑ Access Control (version 0.0.0, mutable)
    ☐ Application Generator (version 0.0.0, mutable)
    ☐ Architecture Generation (version 0.0.0, mutable)
    ☑ Business Modeling (version 0.0.0, mutable)
    ☑ Class Modeling (version 0.0.0, mutable)
    ☐ Code Generator (version 0.0.0, mutable)
    ☑ Component Modeling (version 0.0.0, mutable)
    ☑ Context Modeling (version 0.0.0, mutable)
    ☐ Dynamic Loading (version 0.0.0, mutable)
    ☑ Feature Modeling (version 0.0.0, mutable)
    ☐ Inference Machine (version 0.0.0, mutable)
    ☑ Modeling Infrastructure (version 0.0.0, mutable)
    ☐ Persistence (version 0.0.0, mutable)
    ☑ Process Machine (version 0.0.0, mutable)
    ☐ Reverse Engineering (version 0.0.0, mutable)
    ☐ Search Engine (version 0.0.0, mutable)
    ☑ Sequence Modeling (version 0.0.0, mutable)
```

```
    ☐ component (version 40)
    ☐ context (version 40)
    ☐ feature (version 46)
    ☐ generic (version 40)
    ☐ process (version 40)
    ☐ sequence (version 40)
    ☐ state (version 40)
    ☐ usecase (version 40)
    ☑ user (version 40)
      ☑ FabricaRepresentacoes.java (version 50)
      ☐ FabricaRepresentacoesLexicas.java (version 40)
      ☐ ModeloRenderer.java (version 40)
      ☐ PainelEdicaoCategoria.java (version 40)
      ☐ PainelEdicaoLexico.java (version 40)
      ☐ RepresentacaoCategoria.java (version 40)
      ☐ RepresentacaoLexica.java (version 40)
  ☐ rule (version 40)
  ☐ tools (version 45)
```

```
Sun Aug 07 13:37:50 BRT 2005 - Trace from Use Case Modeling (version 0.0.0, mutable) to usecase (version 40) updated to usecase (ve
Sun Aug 07 13:37:50 BRT 2005 - Trace from Use Case Modeling (version 0.0.0, mutable) to usecase (version 40) removed.
Sun Aug 07 13:37:50 BRT 2005 - Trace from Feature Modeling (version 0.0.0, mutable) to feature (version 40) updated to feature (versio
```

Figure 5: ArchTrace screenshot

4.2. Policies API

Each ArchTrace policy is implemented as a Java class that follows a specific interface provided by ArchTrace. Every policy must provide a short description and the rationale behind the policy. Moreover, a method called "execute" should be implemented. The arguments of this method vary depending on the type of policy. The *pre-trace* and *post-trace* policies receive the link that is being added or removed, as well as the action that is causing that link to be added or removed. An *architectural element evolution* policy receives the architectural element that is being evolved and the details of how its links are being evolved. Finally, an *implementation evolution* policy receives the configuration item and, once again, the details of how its links are being evolved. Using this information, as well as the querying capabilities of the *Traceability* component listed in Figure 4, policies should have sufficient information to make their decisions. If that is not the case, they can use the user interface of ArchTrace to request additional information from the user. Further details regarding ArchTrace internals can be found at [14].

4.3. Built-in policies

We have implemented an initial set of eight policies. We developed them based on informally observing ourselves and other developers in action. Table 1 presents a list of the policies together with their motivation and related policies ("REL" column).

During the design of ArchTrace, we simulated a set of hypothetical scenarios in which different changes were made to an architecture and its implementation and observed the effects the changes should have had on the traceability links among the elements. As a first observation, we noted that, when a new version of a source file is available, it is necessary to use this version for architectural elements that are under development. This led us to create three different atomic policies: addition of new traceability links when new versions of source files are available (policy 8), removal of old traceability links when new traceability links are created (policy 5), and denial of traceability links creation and removal to immutable architectural elements (policy 2). Together, these policies ensure traceability links are updated to newer versions, but that the links of immutable architectural elements are kept untouched.

Another common pattern that we observed was that, when a new version of an architectural element is created, it should inherit all traceability links from its ancestor. This led us to policy 7, which copies all traceability links from the previous version of an architectural element when a new version is created.

In addition, depending on the combination of the policies described above, a given architectural element may have traceability links assigned to more than one version of the same source code. This situation should be avoided depending on the underling programming language (i.e., compiling and running a system with two files in which the same Java class is defined is prohibited by the language); this led us to create policy 3. Additionally, when a source file undergoes a name change, users that are not aware of the name change may erroneously establish a traceability link to the original artifact. In the example of Figure 3, *Action.java* was renamed to *Command.java*. In this scenario, the user is warned by policy 1 if they try to establish a traceability link to *Action.java*, but can use the interface of ArchTrace to nonetheless establish the link.

Because most CM systems allow hierarchical organization of source files, a potential redundancy emerges when both the container and the contained are linked. To avoid this situation, both proactively and passively, we implemented policies 4 and 6. The policies simply link to the container, indicating that it and all of its contents belong to a particular architectural element.

4.4. Policy triggering example

We now revisit the example of Section 2 to describe ArchTrace's handling of the transformation from the initial scenario, shown in Figure 1, to the final scenario after the changes, in Figure 3.

After the first action is performed by the developer, namely the creation of a new version of the *Print* component, ArchTrace receives an architectural evolu-

Table 1: ArchTrace built-in policies

ID	TYPE	DESCRIPTION	REASONING	REL
1	Interactive constraint Class: *pre-trace*	Suggests traceability links to more recent configuration item version if the user creates a traceability link to older version.	When different versions of a configuration item have different names or paths, a traceability link should be created to its newer version.	
2	Automatic constraint Class: *pre-trace*	Denies traceability links creation or removal on immutable architectural elements.	In some circumstances, it is not desirable to evolve the traceability links of architectural elements that are marked as "immutable".	8
3	Automatic constraint Class: *pre-trace*	Denies traceability links creation to more than one version of the same configuration item.	Some programming languages do not support more than one version of the same configuration item in the same runtime environment.	5,8
4	Automatic constraint Class: *pre-trace*	Denies traceability link creation to sub configuration items if the composite configuration item is already traced.	If a composite configuration item is linked from a given architectural element, it is redundant to have traceability links to its parts.	6
5	Automatic rule Class: *post-trace*	Removes traceability links from old configuration item versions when a traceability link is created to a newer version.	Some programming languages do not support more than one version of the same configuration item in the same runtime environment.	3,8
6	Automatic rule Class: *post-trace*	Removes traceability links from sub configuration items if a traceability link is created to the composite configuration item.	If the composite configuration item is traced from a given architectural element, it is redundant to have traceability links to its parts.	4
7	Automatic rule Class: *arch. evol.*	Copies all existing traceability links to the new version of the architectural element.	Typically, new architectural element versions have the same traceability links of the version from which they were originated.	
8	Automatic rule Class: *impl. evol.*	Automatically updates traceability links when a new version of a configuration item is available.	Architectural elements that have traceability links to a specific configuration item should be updated with links to newer versions.	2,3,5

tion notification. This notification triggers policy 7, which is responsible for copying all traceability links from the first version of the *Print* component to the second version of the same component. After the execution of policy 7, both versions of the *Print* component have equivalent sets of traceability links. However, the first version is immutable, meaning that its traceability links will never change. On the other hand, the second version may have its traceability links evolved in the future. Figure 2 shows the scenario after the execution of policy 7.

The developer performs a second action, which consists of first changing the code of *Action.java*, then moving it to the *Controller* directory, and finally changing its name to *Command.java*. When this overall change is committed, a notification is sent to ArchTrace, which triggers policy 8, creating a new traceability link from the *Toolbar* component (version 1.0) to *Command.java* (version 2.0). However, the execution of policy 8 triggers policy 5, which is responsible for removing the old traceability link from the *Toolbar* component (version 1.0) to *Action.java* (version 1.0).

Policy 8 is triggered two more times for the same notification event. The second triggering of policy 8 tries to create a traceability link from the *Print* component (version 1.0) to *Command.java* (version 2.0). However, policy 2 denies the creation of this traceability link because the *Print* component (version 1) is marked as immutable. Finally, the third triggering of policy 8 creates a traceability link from the *Print* component (version 2.0) to *Command.java* (version 2.0). This is allowed by the pre-trace policy 2, which is triggered, but does not undertake action since version 2.0 of the *Print* component is not immutable. Because the action is allowed, the creation of this traceability link triggers post-trace policy 5, which removes the old traceability link from the *Print* component (version 2.0) to *Action.java* (version 1.0).

5. Evaluation

To evaluate the effectiveness of ArchTrace and its current set of policies, we executed a retrospective study of an existing system. The system, Odyssey [23], is a large-scale software development environment being developed at COPPE/UFRJ since 1997.

To perform the study, we gathered the Odyssey versioning data produced during the period of July 9, 2003 until March 1, 2005. We used and reorganized the data to replicate the original check-ins that took place, and then replayed those check-ins anew into a CM repository instrumented with ArchTrace. The result was that, during playback, we received all the events that would have taken place had ArchTrace been used in the first place, allowing us to reproduce the original scenario of development and maintenance, covering both major architectural changes and a host of source code changes. This strategy made it possible to look back in time and understand whether our policies would have operated properly in establishing and evolving the right set of traceability links.

5.1. Study planning

The study consists of four steps. The *first step* consists of the initial detection of the proper traceability links between the Odyssey architecture and its source code on July 9, 2003. This initial set of traceability links was manually identified by Odyssey developers by examining the architectural definition and its realization as components, connectors, and interfaces in the source code.

The *second step* is the evolution of the traceability links during 20 months of Odyssey development and maintenance. Replaying the set of check-ins that were originally performed in this period of development and maintenance, the initial set of traceability links was transformed, step-by-step as triggered by each check-in, into a new set of traceability links. This evolved set of traceability links is named T_e.

The *third step* consists of the detection of the traceability links that should exist on March 1, 2005 among the Odyssey architecture and source code. This set of ideal traceability links, named T_i, was manually created by Odyssey developers by examining the actual architecture as evolved over the period of time and identifying the source files that implement each architectural element.

Finally, the *fourth step* consists of the comparison of the set of ideal traceability links (T_i) with the set of actual traceability links produced by ArchTrace (T_e). This comparison illustrates the effectiveness of the ArchTrace policies in evolving traceability links.

5.2. Environment Preparation

Table 2 shows some Odyssey statistics. We note that the system is non-trivial, consisting of over 2700 files, and that the study also represents a significant set of data with a total number of commits during the study period of 307 and a total number of revisions to individual artifacts (both architectural and at the implementation level) of close to 8500.

At the beginning of the playback, we turned on all policies except 1 and 3. Policy 3 is not designed to operate concomitant with policies 5 and 8; as the effect is either preventive (policy 3) or proactive (policies 5 and 8) and we chose a proactive approach (others may choose a more cautious route, in just using policy 3). Policy 1 is designed to operate in an interactive manner, at times requesting user input. We turned off any policies involving interactivity to avoid ourselves giving potentially "better" input than original developers would have given; our results, thus, form a lower bound of what is theoretically possible.

Table 2: Odyssey statistics

Files	2703	**Repository size**	40158 KB
Revisions	8463	**Total commits**	307
Unique tags	13	**First revision date**	July 9, 2003
Unique branches	7	**Last revision date**	March 1, 2005

5.3. Statistics Gathering

This retrospective study aims to analyze different statistics gathered from the ArchTrace execution. To allow this automatic gathering, we implemented a statistics gathering aspect and weaved it into ArchTrace. The aspect is composed of 19 pointcuts that collect the following 27 metrics for each of the 307 configurations: the configuration number, author, and date; the number of configuration items added, removed, and modified; the number of executions of each policy; the number of traceability links added and removed manually; the number of traceability links added and removed automatically; the number of traceability link additions and removals lost; the number of indirect traceability links added and removed manually; the number of indirect traceability links added and removed automatically; and the number of indirect traceability link additions and removals lost.

In this context, indirect traceability links are traceability links implicitly detected when a given traceability link is established to a composite artifact. For example, if a traceability link is established to a directory, all files and subdirectories inside this directory are also implicitly linked (even though no links exist since our policies handle this recursive traceability). The effect of losing a traceability link to a composite artifact, then, can have significant effects on the functioning of the policies. Hence, we monitored both direct and indirect links in our study.

5.4. Study Execution

Execution of the study comprised two major steps: (1) playback of existing check-ins and (2) analysis of lost traceability links. The first step is performed through a tool that we explicitly wrote to submit, check-in by check-in, the accumulated version history of Odyssey. The tool simply goes through each check-in, recreates a workspace, populates it with the known changes, and commits the workspace. The tool pauses after each step, waiting for manual confirmation that it is okay to move to the next check-in in order to provide time for the analyses in step two.

The second step is performed after each individual check-in has been performed and ArchTrace has responded by evolving the traceability links. We then manually checked if there were any lost traceability links. We kept track of two kinds of lost traceability links: lost additions (i.e., traceability links that ideally exist, but were not added by ArchTrace), and lost removals (i.e., traceability links that ideally do not exist, but were not removed by ArchTrace).

It is important to reiterate that the kinds of changes that we replayed were both at the source code level and the architectural level. Though architectural changes took place less frequently (as one would expect in any kind of project), the architecture of Odyssey went through three major iterations: 1.0.0, 1.1.0, and 1.2.0. With each release, we checked in the architectural elements, triggering architectural element evolution policies. Generally, we allowed ArchTrace to update the traceability links itself, except one time when the architecture evolved with the addition of four new components. An initial set of traceability links was established manually at that time for those components.

We have made available our complete results, both raw and processed, at *http://www.cos.ufrj.br/~murta/ArchTrace/odyssey.html*.

5.5. Qualitative Analysis

During the 20 months of Odyssey development and maintenance, 77 versions of 21 architectural elements were created. Moreover, 3031 configuration items were added, renamed, or moved, 154 configuration items were removed, and 1563 modifications were applied to existing configuration items. Most configuration items were added in July 2003, as shown in Figure 6. This reflects the beginnings of our study. After November 2003, most activities were related to modifications of existing configuration items, with just a few configuration item additions and removals.

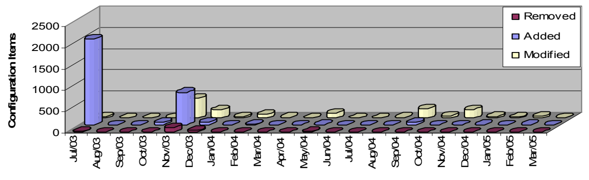


Figure 6: Configuration items evolution

The results of which policies were active during the study are shown in Figure 7. As expected, policies

2, 5, and 8 were used most often, as they represent responses to the normal evolution of configuration items (e.g., links from architectural elements are updated to reflect newer versions of the files).

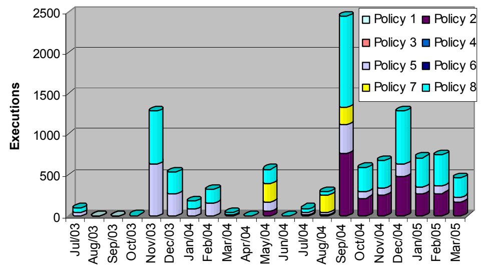


Figure 7: Execution of different policies

The spike in November 2003 indicates an important event. At that time, a major reorganization of the Odyssey source code was performed. This significantly affected the names of packages and the locations of existing classes. Policies 5 and 8 dealt successfully with this situation by updating traceability links to reflect the new organization of the source code. Figure 7 and Figure 8 further illustrate the effects of this event. Figure 7 shows that only policies 5 and 8 were needed to support the reorganization, and Figure 8 shows that those two policies automatically added and removed many traceability links while losing a few.

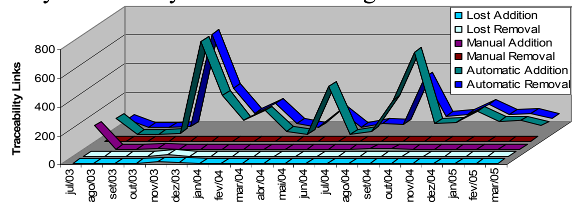


Figure 8: Traceability links evolution

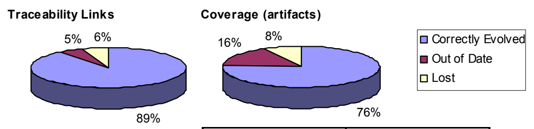
Policy 7, which is responsible for copying existing traceability links to new versions of architectural elements, was triggered on May, August, and September, 2004, meaning the three updates to the Odyssey architecture. Policy 2 was frequently triggered to deny the evolution of traceability links related to immutable architectural elements, since those are now checked in, frozen, and should no longer change.

5.6. Quantitative Analysis

To conclude the study, we compared the set of traceability links evolved by ArchTrace (T_e) with the set of ideal traceability links detected by Odyssey developers (T_i). T_e comprises 222 traceability links and has coverage of 638 artifacts. On the other hand, T_i comprises 235 traceability links and has coverage of 691 artifacts.

Figure 9 presents the summative results of the analyses, illustrating that, at the end of the 20 month evolution, the set of traceability links evolved by ArchTrace (T_e) has 12 out of date traceability links, affecting 113 artifacts. Moreover, 13 traceability links were lost ($|T_i\text{-}T_e|$), affecting 53 artifacts due to the fact that some of the lost links pointed to compound artifacts (i.e., directories). Overall, ArchTrace correctly identified 89% of the ideal set of traceability links and traced 76% of the source code to corresponding architectural elements in the context of the Odyssey project.

To put these figures in perspective, we borrow two metrics from the information retrieval field [5]: precision (the fraction of retrieved documents which are known to be relevant) and recall (the fraction of known relevant documents which were effectively retrieved). These two metrics apply here in the sense that we can use precision to show the percentage of actually identified traceability links that are correct ($|T_i \cap T_e| \div |T_e| = 95\%$; showing that 5% of the traceability links that were found are inaccurate) and recall to show the percentage of ideal traceability links that was actually identified ($|T_i \cap T_e| \div |T_i| = 89\%$; showing we missed merely 11% of the traceability links that should have been found).



	Traceability Links		Coverage (artifacts)	
	#	%	#	%
Correctly Evolved	210	89%	525	76%
Out of Date	12	5%	113	16%
Lost	13	6%	53	8%
Total	235	100%	691	100%

Figure 9: Quantitative analysis summary

5.7. Final Remarks

The data shows that ArchTrace largely operated correctly, even during the reorganization of Odyssey. Traceability links to one directory were lost, however, during this step. This problem occurred because of an interesting situation: a directory was erroneously deleted during the reorganization and had to be reintroduced some revisions later. Not surprisingly, this is a situation with which ArchTrace cannot deal at present. We note, however, that the traceability links of the old versions were fully available, so it would be easy for the developer to reestablish them by hand.

At other times, some traceability links were lost when new artifacts were introduced completely out of context of the existing artifacts. A possible solution to address this problem is the construction of a policy that employs information retrieval techniques [11] or syntactical analysis [7] to detect traceability links. These techniques do not depend on the history of an artifact, so they have the potential to enhance the current set of policies.

Finally, we observe that the study was performed over a relatively stable system and begun after some years of development had taken place. It is unclear how the current set of policies would perform on a new project. We plan on performing further studies and developing additional policies to understand and enhance ArchTrace's behavior in this regard.

6. Related work

Some approaches integrally combine the architecture definition with the source code, avoiding the need for traceability links. For instance, ArchJava [2] enhances the Java programming language with special keywords to integrate an architecture description inside the source code. Similarly, XDoclet [22] uses source code annotations to define EJB components. Clearly, these kind of approaches have their value. However, many situations require architectural representations separate from the source code [17]. In these situations, our approach represents an important contribution.

In the traceability research area, existing approaches are mainly concerned with traceability detection. For instance, De Lucia et al. [11] employ information retrieval techniques to detect traceability links from source code to use cases and test cases. While useful in and of themselves, for our problem they are inadequate. At best, it is necessary to rerun the entire algorithms to redetect proper traceability links. Because this ignores any previous information, the results obtained are typically not as strong as one would with ArchTrace. Nonetheless, we view this technique complementary to ArchTrace and believe this kind of approach can be used *together with* ArchTrace, helping to detect initial traceability links that will subsequently be evolved using ArchTrace.

Work in the consistency checking research area helps to detect inconsistencies among different software representations. Reiss [19], Nentwich et al. [15], and Abi-Antoun et al. [1] map specific representations of software artifacts into a generic representation: relational database, XML, and tree structured data, respectively, and then allow the construction of syntactical constraints among these representations, such as well-formedness rules and direct transformations. ArchTrace differs from these approaches. First, ArchTrace is a proactive tool, which evolves traceability links due to changes in software artifacts, not only reporting but also trying to avoid possible inconsistencies. Moreover, ArchTrace uses the history dimension to detect the evolution of traceability links over time. Finally, ArchTrace deals with architectural elements, which are coarse grained and cannot have all their traceability links directly detected via syntactical constraints. Nevertheless, we once again believe that these approaches can work together with ArchTrace, reporting syntactical inconsistencies between architectural elements and source-code elements, i.e., helping to detect when the automated policies may have done something wrong. By utilizing these techniques in some constraint policies, thus, we believe our approach can be made more powerful.

The research area of hypertext can be useful as an infrastructure for our work. This research area contributes mechanisms to manage the versioning of links among objects (e.g.: Chimera [3] and Molhado [16]). Instead of storing the links in xADL 2.0, we could store them in a hypertext tool. However, by themselves these tools are not sufficient to address our problem as they lack the policy-based enactment that is at the heart of ArchTrace.

7. Conclusion

This paper has presented a new approach for managing the evolution of traceability links between a software architecture and its implementation. Existing traceability approaches have focused on creating one-time snapshots of traceability links. While useful, the next problem is to evolve these snapshots. This is the focus of the work presented here: policy-based evolution of traceability links. The idea is that, by staying in lockstep with architectural and source code changes, it is much easier to solve small incremental problems of maintaining traceability. Through our policies, this is exactly what we do – and we achieve high quality results in both precision and recall.

We therefore view this paper as a successful existence proof of our technique and anticipate it to open a range of additional issues, questions, and refined approaches. While promising, much more work remains to be done. First and foremost, we recognize that, ideally, we should achieve 100% precision and recall. This, however, is unrealistic. No set of policies can anticipate every single potential change. However, with careful choosing by the user of which policies are active at which time, with carefully designing the policies to be interactive when needed, and by integrating some of the data mining techniques described in the previous section, we believe ArchTrace can be turned into a highly effective and practical solution for maintaining accurate traceability among an evolving architecture and its evolving code base.

An additional issue that we would like to address is branching. Our current policies do not quite handle this correctly. While architectural branches are handled correctly, at the source code level some side effects take place (inadvertent removal of “older” links). We plan on implementing a workaround for this problem in the form of a pre-trace policy that denies removal of traceability links to source code for which a new link to a branch is added.

8. Acknowledgments

This work is sponsored in part by NSF grants CCR-0093489 and IIS-0205724, and CAPES grant BEX0323/04-7. We wish to thank the students at UC Irvine and COPPE/UFRJ for their contributions.

References

[1] M. Abi-Antoun, et al., "Semi-Automated Incremental Synchronization between Conceptual and Implementation Level Architectures", WICSA, Pittsburgh, PA, November, 2005.

[2] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting Software Architecture to Implementation", International Conference on Software Engineering, pp. 187-197, Orlando, USA, May, 2002.

[3] K. M. Anderson, R. N. Taylor, and E. J. Whitehead, "Chimera: hypertext for heterogeneous software environments", Conference on Hypertext and Hypermedia, pp. 94 -107, Edinburgh, Scotland, September, 1994.

[4] G. Antoniol, et al., "Recovering Traceability Links between Code and Documentation", *IEEE Transactions on Software Engineering*, vol. 28, n. 10, pp. 970-983, October, 2002.

[5] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, ACM Press, 1999.

[6] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, Addison Wesley, 2000.

[7] L. C. Briand, Y. Labiche, and L. O'Sullivan, "Impact Analysis and Change Management of UML Models", International Conference on Software Maintenance, pp. 256-265, Amsterdam, Netherlands, September, 2003.

[8] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control with Subversion*, O'Reilly, 2004.

[9] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management", *ACM Computing Surveys*, ACM Press, vol. 30, n. 2, pp. 232-282, June, 1998.

[10] E. Dashofy, A. Hoek, and R. N. Taylor, "A Highly-Extensible, XML-Based Architecture Description Language", Working IEEE/IFIP Conference on Software Architectures, pp. 103-112, Amsterdam, Netherlands, August, 2001.

[11] A. De Lucia, et al., "Enhancing an Artefact Management System with Traceability Recovery Features", International Conference on Software Maintenance, pp. 306-315, Chicago, Illinois, September, 2004.

[12] IEEE, "Std 1471 - IEEE Recommended Practice for Architectural Description of Software-Intensive Systems", Institute of Electrical and Electronics Engineers, 2000.

[13] A. Marcus and J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing", International Conference on Software Engineering, pp. 125-135, Portland, Oregon, USA, 2003.

[14] L. G. P. Murta, A. van der Hoek, and C. M. L. Werner, "ArchTrace: A Tool for Keeping in Sync Architecture and its Implementation", Brazilian Symposium on Software Engineering (SBES), Tools Session, Florianópolis, Brazil, October, 2006.

[15] C. Nentwich, et al., "Flexible Consistency Checking", *ACM Transactions on Software Engineering and Methodology*, ACM Press, vol. 12, n. 1, pp. 28-63, January, 2003.

[16] T. N. Nguyen, E. V. Munson, and J. T. Boyland, "The molhado hypertext versioning system", Conference on Hypertext and Hypermedia, pp. 185-194, Santa Cruz, USA, August, 2004.

[17] R. v. Ommering, et al., "The Koala Component Model for Consumer Electronics Software", *IEEE Computer*, vol. 33, n. 6, pp. 78-85, March, 2000.

[18] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-Based Runtime Software Evolution", International Conference on Software Engineering (ICSE), pp. 177-186, 1998.

[19] S. P. Reiss, "Constraining Software Evolution", International Conference on Software Maintenance (ICSM), IEEE Press, pp. 162-171, Montreal, Canada, October, 2002.

[20] D. J. Richardson and A. L. Wolf, "Software Testing at the Architectural Level", International Software Architecture Workshop (ISAW), pp. 68-71, San Francisco, USA, October, 1996.

[21] J. S. Shirabad, T. Lethbridge, and S. Matwin, "Supporting Software Maintenance by Mining Software Update Records", International Conference on Software Maintenance, pp. 22-31, Florence, Italy, November, 2001.

[22] C. Walls and N. Richards, *XDoclet in Action*, Manning Publications, 2003.

[23] C. M. L. Werner, et al., "OdysseyShare: an Environment for Collaborative Component-Based Development", IEEE Conference on Information Reuse and Integration, pp. 61-68, Las Vegas, USA, October, 2003.

[24] A. T. T. Ying, et al., "Predicting Source Code Changes by Mining Change History", *IEEE Transactions of Software Engineering*, vol. 30, n. 9, pp. 574-586, September, 2004.

[25] J. Zhao, et al., "Change impact analysis to support architectural evolution", *Journal of Software Maintenance: Research and Practice*, John Wiley & Sons, Inc., vol. 14, n. 5, pp. 317-333, New York, USA, September, 2002.