Autom Softw Eng (2008) 15: 75–107
DOI 10.1007/s10515-007-0020-6

Continuous and automated evolution of architecture-to-implementation traceability links

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

Received: 13 November 2007 / Accepted: 16 November 2007 / Published online: 27 November 2007

Abstract A traditional obstacle in the use of multiple representations is the need to maintain traceability among the representations in the face of evolution. The introduction of software architecture, and architecture-based development, has brought this need to architectural descriptions and corresponding source code. Specifically, the task is to relate versions of architectural elements to versions of source code configuration items, and to update those relations as new versions of the architecture and source code are produced. We present ArchTrace, a new approach that we developed to address this problem. ArchTrace distinguishes itself by continuously updating traceability relations from architectural elements to code elements through a policy-based extensible infrastructure that allows a group of developers to choose a set of traceability management policies that best match their situational needs and/or working styles. We introduce the high-level approach of ArchTrace, discuss its extensible infrastructure, and present our current set of ten pluggable traceability management policies. We conclude with a retrospective analysis of data collected from a twenty month period of development and maintenance of Odyssey, a component-based soft-

This paper is an extended version of the paper "ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links", which was published in the proceedings of the Twenty-first ACM/IEEE International Conference on Automated Software Engineering, 2006.

L. G. P. Murta (✉) · C. M. L. Werner
COPPE—System Engineering and Computer Science, Federal University of Rio de Janeiro, P.O. Box 68511, Rio de Janeiro, RJ 21945-970, Brazil
e-mail: murta@cos.ufrj.br

C. M. L. Werner
e-mail: werner@cos.ufrj.br

A. van der Hoek
Department of Informatics, University of California at Irvine, 221 ICS2 Building, Irvine, CA 92697-3440, USA
e-mail: andre@ics.uci.edu

ware development environment comprised of over 50,000 lines of code. This analysis shows that our approach is promising: with respect to the ideal set of traceability links, the policies applied resulted in a precision of 95% and recall of 89%.

Keywords Traceability · Software architecture · Configuration management · Software evolution

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 (Oreizy et al. 1998; Van der Hoek 2004), product selection in software product lines (Bosch 2000; Chen et al. 2003), new testing approaches (Richardson and Wolf 1996; Muccini and Van der Hoek 2003), impact analyses (Zhao et al. 2002), and numerous other activities that will not operate properly without a detailed and accurate mapping from an architectural description to relevant corresponding source code configuration items.

The fact that both the architecture and the source code can – and do – evolve independently represents a significantly complicating factor. It may be feasible for a developer to specify a proper mapping once, but it is not reasonable to expect the developer to continuously maintain and evolve that mapping manually, especially not when the software system under development is of significant scale and undergoes numerous changes.

Several different approaches already address the problem of maintaining traceability between an architectural description and corresponding source code configuration items. These approaches can be classified into two categories: *equality by definition* and *after the fact reconstruction*. Equality by definition refers to methods in which an architectural description and its source code configuration items are perfectly traceable because one is embedded inside the other. For instance, ArchJava (Aldrich et al. 2002) and XDoclet (Walls and Richards 2003) embed the definition of architectural elements in the source code. While this kind of solution is effective in maintaining 100% accuracy, it is not as realistic, as it is often the case that the architecture of a system is maintained in an architecture description that is separate from the source code, with different people using different tools and different notations maintaining the two.

Data mining (Shirabad et al. 2001; Ying et al. 2004; Zimmermann et al. 2004), information retrieval (Antoniol et al. 2002; Huffman Hayes et al. 2003; Marcus and Maletic 2003; Settimi et al. 2004), and syntactic analysis (Briand et al. 2003) techniques fall into the category of after the fact reconstruction. This category encompasses techniques which (re)discover traceability links. These techniques tend to be generic in nature, and do not take into account the special relationship between architecture and source code, nor do they leverage the structured way in which both tend to co-evolve. Because of their reliance on mathematical properties, and low tolerance for exceptions, their performance is suboptimal when applied to the problem of architecture-to-implementation traceability.

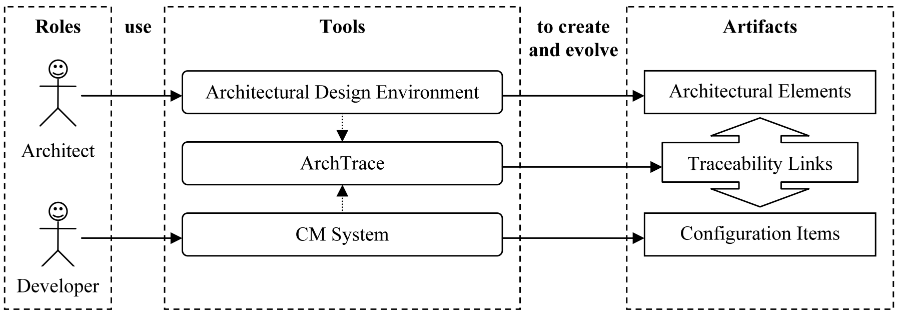


Fig. 1 ArchTrace context

In this paper, we present an alternative approach that falls in between equality by definition and after the fact reconstruction. This approach can be typified as *instant update*, and relies on two critical observations: (1) rather than reconstructing traceability links after some significant amount of time has passed, we continuously update the 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 regarding architectural and source code evolution, and accommodates the incorporation of new policies.

We have implemented this approach in an extensible infrastructure, ArchTrace, which we explicitly designed to support policy-based traceability management between evolving architectural descriptions and evolving source code configuration items. ArchTrace operates through triggers that it inserts into external systems, most notably in the environment used to evolve architecture – typically an architectural design environment – and in the environment used to evolve source code configuration items – typically a Configuration Management (CM) system. These triggers monitor changes made by users to the architecture description or configuration items, and fire when those changes are committed. Upon firing, ArchTrace runs any applicable policies to update traceability links, as depicted in Figure 1. As an example, when a developer checks in a modification to a source file, ArchTrace runs a policy that adds a traceability link from the corresponding architectural element to the new version of the source file and another policy that removes the traceability link to the old version.

An important aspect of ArchTrace is that it is pluggable with respect to the set of traceability management policies that it uses. At this moment in time, we have implemented ten such policies, but other policies can easily be coded and used. For instance, current policies focus on tracing components, connectors, and interfaces, but we can extend the set of available policies to additionally trace other, perhaps more fine-grained elements.

We note that, while our goal is to investigate a new technique for automatically maintaining traceability of evolving architecture-to-implementation links, we do not directly compete with data mining, information retrieval, and syntactic analysis techniques. Rather, we see our work as an exploration of a complementary technique –

one that eventually may very well make use of these other techniques in providing its functionality. This is illustrated by our evaluations of ArchTrace. We performed a retrospective analysis of ArchTrace as applied to 20 months of data regarding the development of Odyssey (Werner et al. 2003), a component-based software development environment consisting of over 50,000 lines of code covering about 20 components. Initial results were promising, but they improved some when we incorporated an extra policy based on straightforward data mining to help in establishing an initial set of traceability links.

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. The example concerns a word processing application, the architecture of which is shown on the left hand side of Figure 2. This architecture has three components (*Print*, *Toolbar*, and *Display*), one connector (*Bus*), and two interfaces (*Input* and *Output*). All architectural elements exist in a single version and the source code that implements these architectural elements is organized into three directories: *Model*, *View*, and *Controller*. These directories contain,

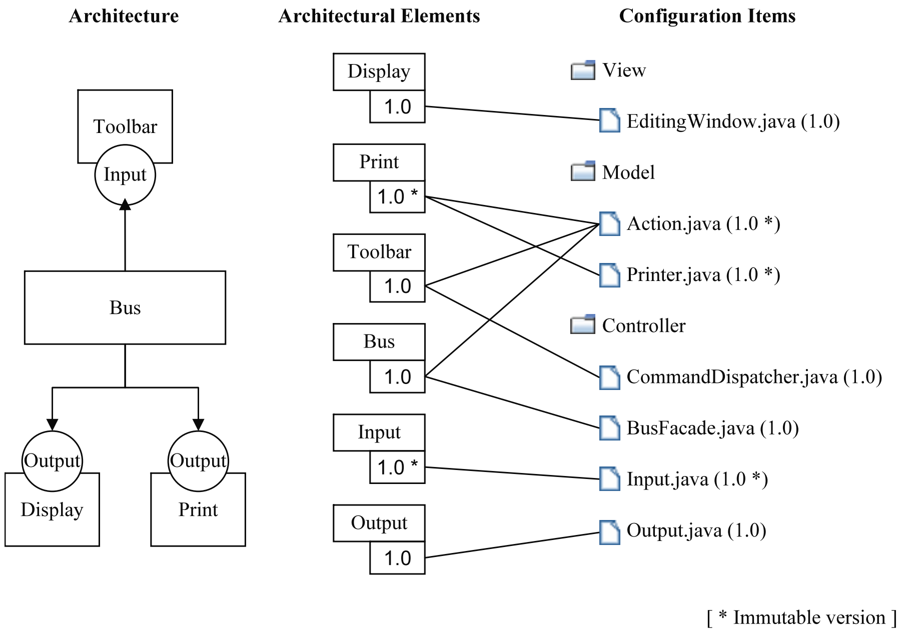


Fig. 2 Starting situation for our example scenario

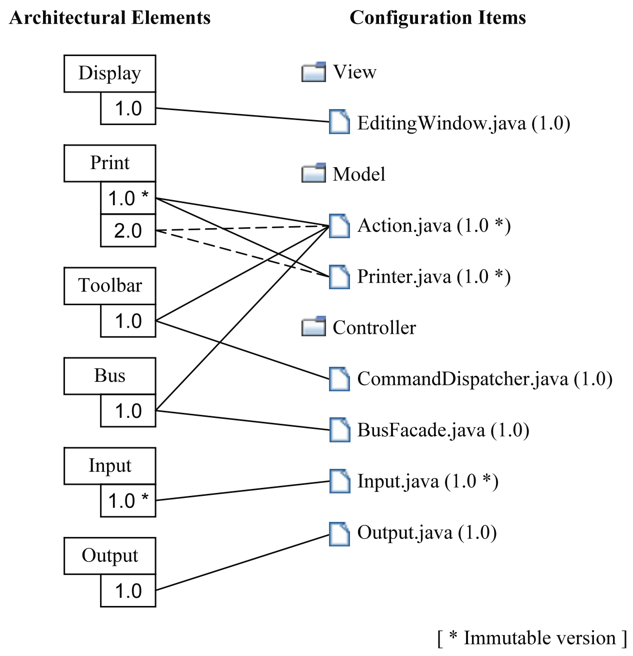


Fig. 3 Situation after *Print* version 2.0 is created

respectively: *Printer.java* and *Action.java*; *EditingWindow.java*; and *CommandDispatcher.java*, *BusFacade.java*, *Input.java*, and *Output.java*, as shown on the right hand side of Figure 2.

Figure 2 also shows that the *Print* component and *Input* interface are immutable, as they were already committed and can no longer be changed (unless, of course, new versions are created). Further, the *Print* component is implemented by *Printer.java* and *Action.java*; the *Toolbar* component by *Action.java* and *CommandDispatcher.java*; the *Display* component by *EditingWindow.java*; the *Bus* connector by *Action.java* and *BusFacade.java*; the *Input* interface by *Input.java*; and the *Output* interface by *Output.java*. Note that the source files that implement the *Print* component and *Input* interface are immutable already.

The first step of our scenario consists of an architectural change, namely to create version 2.0 of the *Print* component. As a result, 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 3.

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 now needs to be updated to reflect these changes. Specifically, architectural elements that used to link to version 1 of *Action.java* should now 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

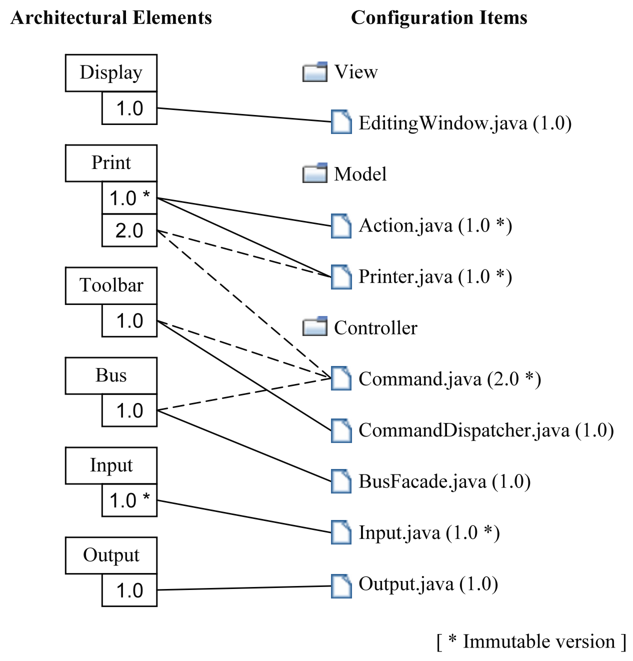


Fig. 4 Situation after *Action.java* is checked out, modified, moved, renamed to *Command.java*, and checked in

version, its traceability links should not be updated to allow history to remain intact. Figure 4 shows the resulting set of traceability links. Three links, from the *Print* component (version 2.0), *Toolbar* component (version 1.0), and *Bus* connector (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.

The final step consists of two interrelated changes to the architecture and source code: (1) a new version of the *Input* interface is created as the result of a check out, modification, and check in, and (2) the *Input.java* file is checked out, modified, and checked in. Again, the set of traceability links must be updated to reflect these changes, with the result shown in Figure 5. It is important to note that these updates must be independent of the order in which the two changes are committed (i.e., regardless of whether the architectural change is checked in first or whether the source code change is checked in first, the set of traceability links that eventually results must be exactly the same).

It is worth noting that, for illustration purposes, the example intentionally presents a simple scenario. However, it provides concrete situations in which evolution of traceability links is difficult, even with automated tools: source-code elements being moved and renamed, traceability links being updated selectively due to immutability, and interrelated modifications requiring consistent results regardless of the order of commits.

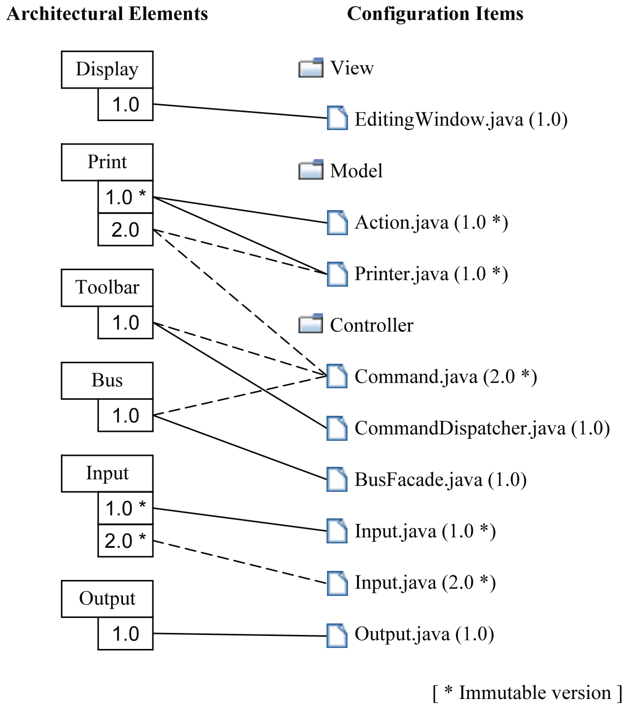


Fig. 5 Situation after Input and Input.java are updated jointly

3 Approach

The goal of ArchTrace is to support the *evolution* of traceability links; we are explicitly not concerned with establishing the links in the first place. While our framework does support the incorporation of data mining and other policies for the purposes of creating an initial set of links (we illustrate one such example in Section 4), we have concentrated on the evolution of traceability links since this is a difficult, unaddressed, yet important property in the face of architecture-based development (Medvidovic and Rosenblum 1997). Generally speaking, the problem that we address in this paper can be stated as follows: given an initial set of established traceability links, and given that both an architecture and its implementation can evolve independently, how can 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 configuration items, and vice versa? In essence, we want to find an automated way of evolving traceability links as an architecture and/or implementation change.

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;

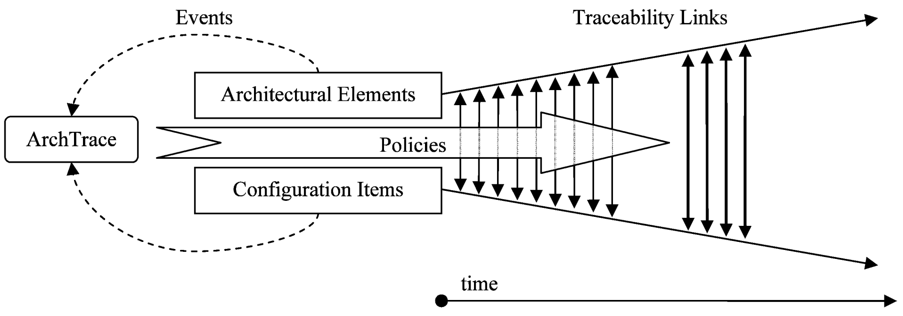


Fig. 6 Traceability links evolution via policy triggering

(4) policies that act as either rules, deciding upon actions to take, or constraints, limiting the kinds of actions that can be taken; and (5) a result that maintains a N-M bidirectional mapping, allowing both architectures and source code to evolve independently while maintaining full navigation from architecture to source code, and vice versa.

The philosophy behind our approach is that it is worth to evolve traceability links continuously, *during* the evolution of architectures and their implementation. Whereas existing approaches are forced to, in essence, rediscover all of the links after an architectural element or source code configuration item has changed, we stay in lockstep with the modifications and update the set of links as soon as a new (architectural or source code) element is checked into the respective repository, as shown in Figure 6.

In response to new "check in" events, multiple policies may be triggered. Policies are intentionally simple, each capturing one small aspect of traceability link evolution that matches potential actions that a user may take. For instance, a policy that deals with checking in a new architectural element, a policy that deals with removing a source code configuration item, a policy that suggests establishing traceability links to the most recent version of a source code configuration item, 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. Usually, then, multiple policies are involved when a developer checks in an architecture or a set of source code configuration items. Such updates can be implemented using three individual policies: a policy to verify whether an element is immutable, a policy to create new traceability links to the latest versions of the source code elements, and a policy to remove old traceability links.

Policies 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 (Conradi and Westfechtel 1998). Rather than attempting to build a single all-encompassing solution, we adopt a pluggable infrastructure that supports the addition of new policies through the programmatic interface of ArchTrace.

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 already existing traceability links. Should such inconsistencies arise, pre-trace policies can veto additions or removals, prohibiting actions from completing. An example of such a pre-trace policy is a policy that prohibits updating a traceability link from a version that resides in the main line of development (trunk) to a newly branched version; this kind of update would lead to inconsistent mixing of trunk and branch versions of source code in the same architecture.

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 code configuration item, 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. For instance, execution of a post-trace policy may lead to the addition of even more traceability links. If this occurs, all pre-trace policies will be triggered again to verify if the suggested traceability links are appropriate. Note that post-trace policies can only execute if all pre-trace policies have approved the suggested changes by other policies, and will only execute after a change has been made. Unlike pre-trace policies, post-trace policies cannot "rollback" the addition or removal of traceability links. Another example of a post-trace policy is a policy that removes an existing traceability link from a source code configuration item when a new traceability link is established to the directory that contains the source code: this existing traceability link would be redundant.

We observe that it is possible for pre-trace policies to not just act as constraints, but also to enact rules that establish or remove traceability links. Post-trace policies, on the other hand, cannot act as constraints, as the execution of a set of policies is stateless – a policy executed after some set of other policies is not aware of the links that were added or removed by these other policies and, therefore, cannot roll back to a previous state.

Policies may request assistance from users; they are not meant to operate automatically and 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 obtain the information it needs. While, in our experience, it is relatively rare that interaction with users is necessary, 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 guessing an alternative, it is better to request user assistance. Note that the reason that it is relatively rare for policies to need human input is because the users are involved in the selection of the policies that are activated in the first place: they already have selected a set of policies that describes how they operate and wish to be supported. An example of an interactive policy is a policy that detects the existence of a newer version of a source code file when a traceability link is being established to an older version. This may indicate that a user is working with older code on the main trunk, which generally is an undesirable situation, but at times may be necessary.

It is important to note that we designed our approach to be compatible with 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 and do not interfere or cause problems when multiple users are involved in modifying the architecture and source code base.

Another important aspect of our approach is that it is designed to be independent of specific tools that are used. Architects can have their own tool to evolve architectures, such as ArchStudio (Dashofy et al. 2002), and implementers can have their own tools to do their work, such as the Eclipse IDE (Eclipse Foundation 2007) and Subversion (Collins-Sussman et al. 2004). All our approach needs to operate are notifications about check in events, and access to the respective repositories to obtain, if needed, additional information with which policies can make their decision.

Clearly, underneath any approach like ours has to be an infrastructure for actually capturing and storing the links that trace architectural elements to their source code configuration items. This infrastructure must support fine-grained links in order to allow the mapping of individual architectural elements to (sets of) individual source code configuration items, and vice versa. Additionally, it must support versioning in order to distinguish different versions of architectural elements and different versions of source code configuration items, as each have their own sets of associated traceability links. This kind of infrastructure is readily available in the form of hypermedia and hypertext versioning systems (Whitehead 2000), and we describe in the next section how we built a straightforward incarnation of such an infrastructure ourselves.

Finally, we note that our approach does not necessarily prescribe any particular policies that it must or must not include; users are free to use whichever policies they desire. Nonetheless, certain policies are commonplace and including them as a standard part of the implementation of our approach – as described in Section 4 – clearly provides advantages in terms of reuse and examples of how policies are constructed and combined.

4 Implementation

ArchTrace is implemented in the Java language and is available at http://www.cos.ufrj.br/~murta/ArchTrace. The current implementation assumes the use of xADL 2.0 (Dashofy et al. 2001) to describe software architectures and Subversion to store source code configuration items.

4.1 Overall architecture

Figure 7 presents the ArchTrace architecture. It consists of six components, four of which standard (shown as solid grey boxes) and the other two custom (shown as

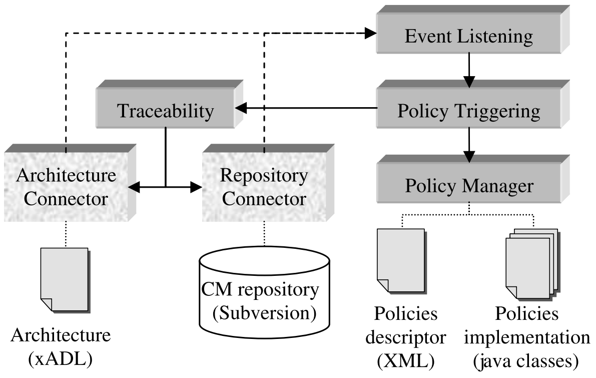


Fig. 7 ArchTrace architecture

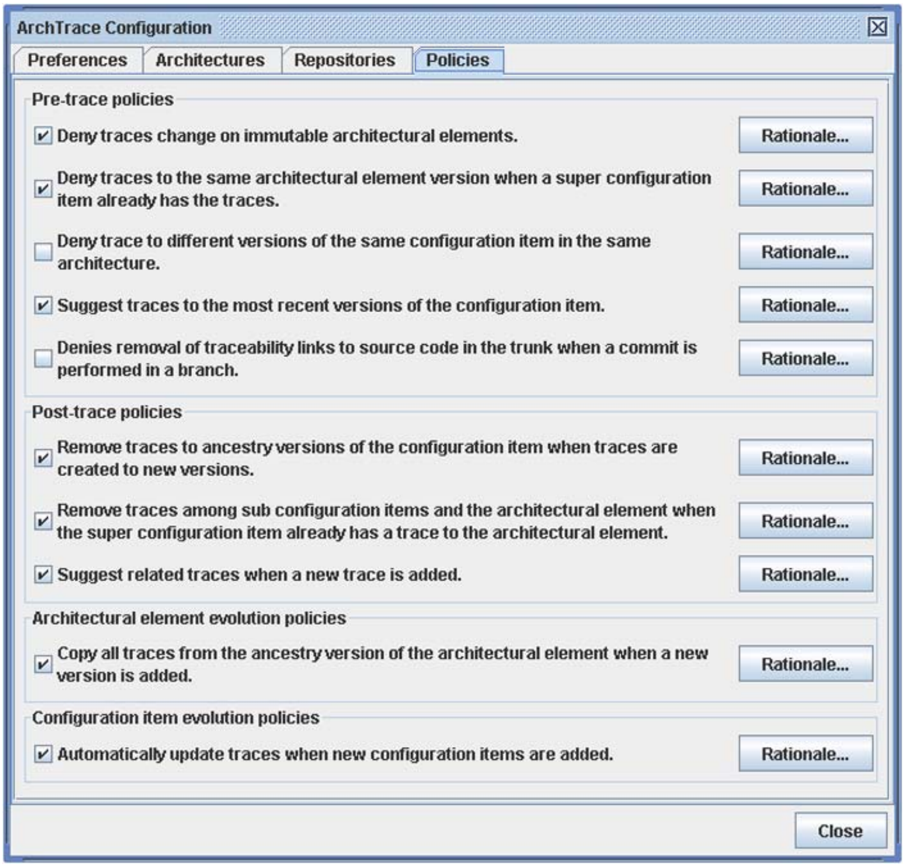
patterned boxes). The custom components depend on the particular architecture evolution environment and CM system used. As stated, we rely on xADL 2.0 and Subversion, but because the *Architecture Connector* and *Repository Connector* components are designed with abstract interfaces, the rest of ArchTrace 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 case of the one of the examples in Section 2, 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.

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, such as xADL 2.0 or UniCon (Shaw et al. 1995), 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, as shown in Figure 8. Here is where the pluggability of ArchTrace comes into play: when new policies are created, these new policies, once loaded by this


ArchTrace Configuration

Preferences | Architectures | Repositories | Policies

Pre-trace policies

- [✓] Deny traces change on immutable architectural elements. — Rationale...
- [✓] Deny traces to the same architectural element version when a super configuration item already has the traces. — Rationale...
- [ ] Deny trace to different versions of the same configuration item in the same architecture. — Rationale...
- [✓] Suggest traces to the most recent versions of the configuration item. — Rationale...
- [ ] Denies removal of traceability links to source code in the trunk when a commit is performed in a branch. — Rationale...

Post-trace policies

- [✓] Remove traces to ancestry versions of the configuration item when traces are created to new versions. — Rationale...
- [✓] Remove traces among sub configuration items and the architectural element when the super configuration item already has a trace to the architectural element. — Rationale...
- [✓] Suggest related traces when a new trace is added. — Rationale...

Architectural element evolution policies

- [✓] Copy all traces from the ancestry version of the architectural element when a new version is added. — Rationale...

Configuration item evolution policies

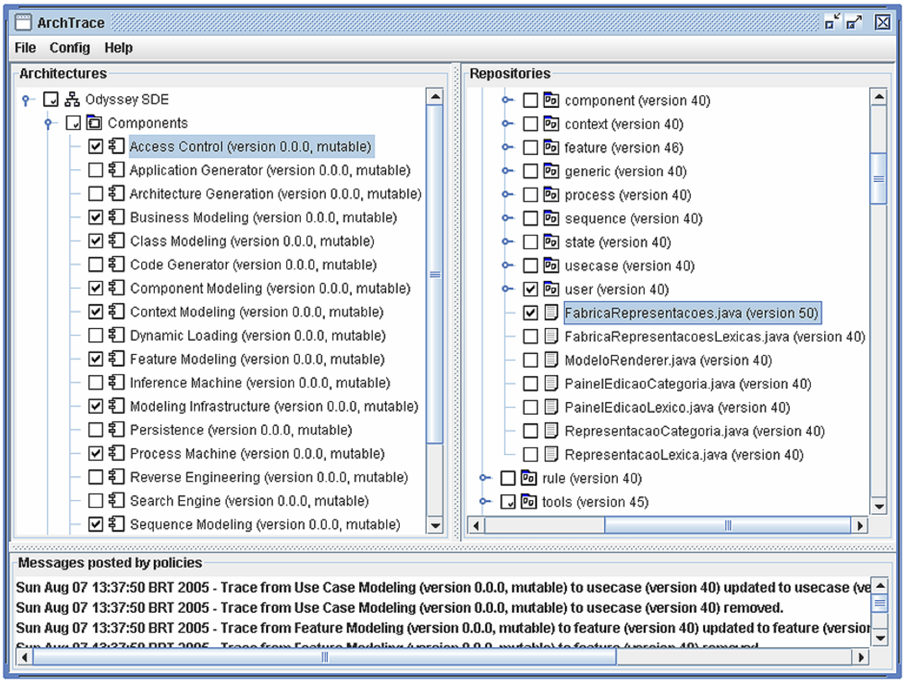
- [✓] Automatically update traces when new configuration items are added. — Rationale...

Close

Fig. 8 ArchTrace policy activation and deactivation

component, will integrate as any of the ten policies that we already built: they can be enabled, disabled, executed, and triggered by other policies.

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. Shown in Figure 9, 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. An example of impact analysis activity using ArchTrace user interface consists on selecting a source code (right hand side of Figure 9) and visualizing all architectural elements that are related to this source code. Moreover, it is possible to select some of these architectural elements (left hand side of Figure 9) and see all source code configuration items that implement them. An additional possibility is to use other tools of ArchStudio, such as Ménage (Garg et al. 2003), to perceive the relationships among components, connectors, and interfaces and use this knowledge to guide new queries in the ArchTrace user interface.



```
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
```

Fig. 9 ArchTrace screenshot

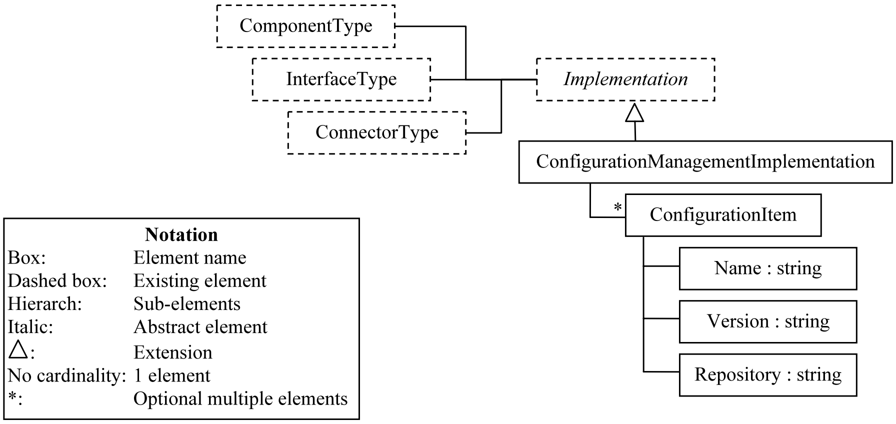


Fig. 10 ArchTrace schema

4.2 Traceability links schema

As mentioned before, ArchTrace uses xADL 2.0 to describe software architectures. Specifically, our work relies on the xADL 2.0 *Implementation* Schema, which defines an abstract element that is a placeholder for data that relates to the implementation of architectural elements. We have extended this abstract schema with a concrete schema that adds traceability to source code stored in configuration management repositories, as shown in Figure 10. Specifically, we support the tagging of architectural elements with a series of configuration items.

Our schema consists of an element named *ConfigurationManagementImplementation*, which is composed of a set of *ConfigurationItem* elements. Each *ConfigurationItem* is represented by the tuple (*name*, *version*, *repository*) where *name* is the name of the configuration item, *version* is the selected version of the configuration item, and *repository* is the configuration management repository address where the configuration item version is stored. For example, the traceability links of component *Print* version 2.0, as presented in the example of Figure 4, can be described via our schema using the information shown in these two tuples:

("Model/Printer.java", 1.0, svn://server/src)

("Controller/Command.java", 2.0, svn://server/src)

4.3 Architecture and repository connection

As mentioned before, ArchTrace abstracts its interaction with specific architecture development environments and configuration management systems through a generic layer. This generic layer has to be specialized for each concrete kind of architecture development environment or CM system. This specialization occurs via the creation of a new architecture or CM repository connector. However, ArchTrace imposes some restrictions to which these connectors should adhere.

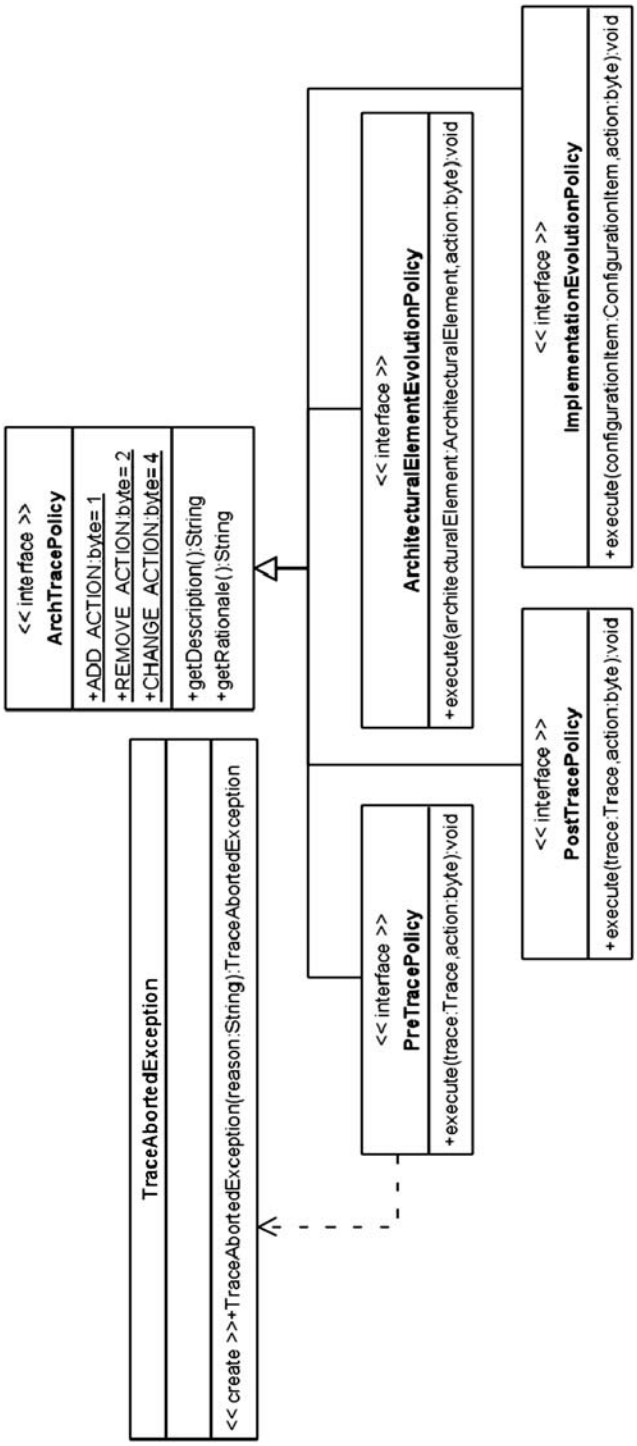
In the case of a new kind of architecture development environment, its architecture connector must implement the *ArchConnector* interface, which details three main functionalities: (1) translation between the general model of architecture used by ArchTrace, which consists of components, connectors, and interfaces, and the specific architectural model of the environment, which may similarly consist of components, connectors, and interfaces, but also can be as varied as packages and dependencies, workflow and services, CORBA components and IDL interfaces, and so on; (2) management (add, remove, and query) of traceability links for a given architectural element; and (3) emission of change events from the architecture development environment to trigger the policies of ArchTrace.

In the case of a new kind of CM system, the CM repository connector must implement the *CMConnector* interface, which details two main functionalities: (1) translation between the general model of a CM repository adopted by ArchTrace, which consists of configurations and configuration items, and the specific CM model of the CM system (e.g., file-oriented or object-oriented) and (2) emission of change events from the CM system to trigger the policies of ArchTrace.

ArchTrace policies, thus, do not need to directly manipulate the architectural elements or their code base as stored in the CM repository. Instead, they can rely on the ArchTrace API, which is composed of the classes *Architecture* and *ArchitecturalElement* for interacting with architectures and the classes *Repository*, *Configuration*, and *ConfigurationItem* for interacting with CM repositories.

4.4 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 ratio-



<< interface >>
ArchTracePolicy
+ADD_ACTION:byte=1
+REMOVE_ACTION:byte=2
+CHANGE_ACTION:byte=4
+getDescription():String
+getRationale():String
TraceAbortedException
<< create >>+TraceAbortedException(reason:String):TraceAbortedException
<< interface >>
PreTracePolicy
+execute(trace:Trace,action:byte):void
<< interface >>
ArchitecturalElementEvolutionPolicy
+execute(architecturalElement:ArchitecturalElement,action:byte):void
<< interface >>
PostTracePolicy
+execute(trace:Trace,action:byte):void
<< interface >>
ImplementationEvolutionPolicy
+execute(configurationItem:ConfigurationItem,action:byte):void

Fig. 11 Policies API

nale 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 traceability link that is being added or removed, as well as the action that informs the policy as to whether the traceability link is being added or removed. An *architectural element evolution* policy receives a pointer to an architectural element and an indication as to what happened to that element (i.e., was it added, removed or changed?). Finally, an *implementation evolution* policy receives a pointer to a configuration item and, once again, an indication as to the specific action that took place (i.e., was it added, removed or changed?). Using this information, as well as the querying capabilities of the *Traceability* component listed in Figure 7, 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. Figure 11 summarizes the API provided by ArchTrace for policies construction.

4.5 Built-in policies

We have implemented an initial set of ten policies. Table 1 presents a list of the policies together with their motivation and related policies ("REL" column). We developed the policies based on informally observing ourselves and other developers in action. In addition, 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 these changes should have had on the traceability links among the elements. These scenarios included the creation of new versions of architectural elements, the creation of new versions of source code, the renaming and moving of source code, the structuring of source code in a composite way, and the initial establishment of traceability links using existing techniques. From these collective experiences, we devised the ten policies presented here, as these provide basic support for some of the most common scenarios.

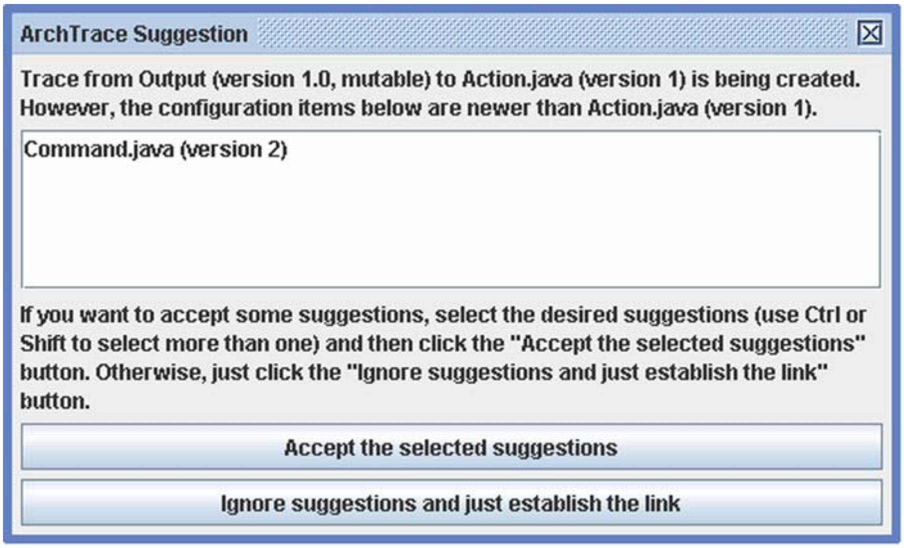
As a first observation, we noted that, when a new version of a source file is available, it is necessary to use this version for architectural elements that are under development. This led us to create three different atomic policies: addition of new traceability links when new versions of source files are available (policy 10), removal of old traceability links when new traceability links are created (policy 6), 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 9, 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 underlying 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

Table 1 ArchTrace built-in policies

ID	Type	Description	Reasoning	Rel
1	Pre-trace (interactive constraint)	Suggests the creation of traceability links to the most recent configuration item version if a traceability link is created to an older version.	Sometimes, especially when the configuration item versions have different names or paths, traceability links are mistakenly established to older versions of the configuration item because the user does not know that there are newer versions available.	
2	Pre-trace (automatic constraint)	Denies creation or removal of traceability links on immutable architectural elements.	Usually, it is not desirable to evolve traceability links of architectural elements that are marked as "immutable" because they are considered stable.	10
3	Pre-trace (automatic constraint)	Denies creation of traceability links 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 to be included in the same runtime environment.	6,10
4	Pre-trace (automatic constraint)	Denies creation of traceability link to sub configuration items if the composite configuration item is already traced.	If a composite configuration item (i.e., directory) is linked from a given architectural element, it is redundant to have traceability links to its parts (i.e., subdirectories and files).	7
5	Pre-trace (automatic constraint)	Denies removal of traceability links to source code in the trunk when a commit is performed in a branch.	Commits in branches should not interfere with the main line of development.	
6	Post-trace (automatic rule)	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.	3,10
7	Post-trace (automatic rule)	Removes traceability links from sub configuration items if a traceability link is created to the composite configuration item.	If a composite configuration item (i.e., directory) is linked from a given architectural element, it is redundant to have traceability links to its parts (i.e., subdirectories and files).	4
8	Post-trace (interactive rule)	Suggests related traceability links when a traceability link is created.	Usually, architectural elements that have traceability links to a given configuration item also have traceability links to other configuration items. Data-mining techniques can be used to detect these related traceability links, avoiding incomplete traces.	
9	Architectural Element Evolution (automatic rule)	Copies all existing traceability links to the new version of the architectural element when it is available.	Typically, new architectural element versions start out with the same traceability links as those of the previous version the version from which they were originated.	
10	Implementation Evolution (automatic rule)	Updates traceability links when a new version of a configuration item is available.	This represents natural evolution of the implementation of architectural elements.	2,3,6


ArchTrace Suggestion

Trace from Output (version 1.0, mutable) to Action.java (version 1) is being created. However, the configuration items below are newer than Action.java (version 1).

Command.java (version 2)

If you want to accept some suggestions, select the desired suggestions (use Ctrl or Shift to select more than one) and then click the "Accept the selected suggestions" button. Otherwise, just click the "Ignore suggestions and just establish the link" button.

Accept the selected suggestions
Ignore suggestions and just establish the link

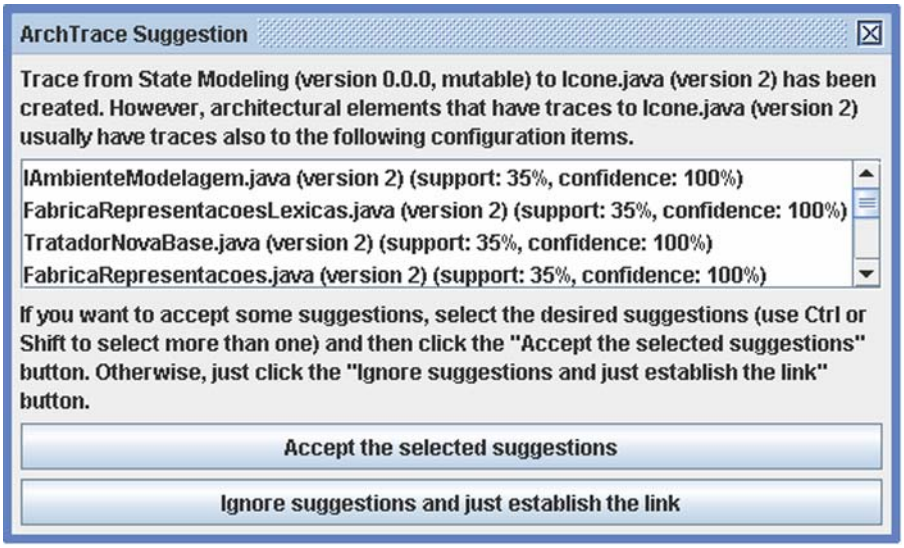
Fig. 12 ArchTrace suggestion based on history analysis

policy 3. Additionally, when a source code configuration item undergoes a name or path change, users that are not aware of the new name or path change may erroneously establish traceability links to older versions of the source code. In the example of Figure 4, *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, shown in Figure 12, 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 7. The policies simply link to the container, indicating that it and all of its contents belong to a particular architectural element.

An additional issue that we addressed is branching. While architectural branches are handled correctly just with policy 9, at the source code level some side effects take place when Policy 10 is used: inadvertent removal of traceability links to the main line of development (trunk) because generally policy 6 will also be in use. Therefore, we included in our standard set of policies a pre-trace policy (policy 5) that denies removal of traceability links to source code for which a new traceability link is added to a branch. It is worth to note that policy 5 is automatic at this moment, always denying the removal of a trunk traceability link in response to commits on branches.

Finally, we observe our discussion of Section 1 on data mining. We see data mining (Shirabad et al. 2001; Ying et al. 2004; Zimmermann et al. 2004) and other existing techniques for traceability detection (Antoniol et al. 2002; Briand et al. 2003; Huffman Hayes et al. 2003; Marcus and Maletic 2003; Settimi et al. 2004) as complementary to our approach. In order to demonstrate this, we implemented policy 8 to show the feasibility of integrating data mining into our technique. This policy uses association rules (Agrawal and Srikant 1994) to suggest new traceability links



Trace from State Modeling (version 0.0.0, mutable) to Icone.java (version 2) has been created. However, architectural elements that have traces to Icone.java (version 2) usually have traces also to the following configuration items.

IAmbienteModelagem.java (version 2) (support: 35%, confidence: 100%)
FabricaRepresentacoesLexicas.java (version 2) (support: 35%, confidence: 100%)
TratadorNovaBase.java (version 2) (support: 35%, confidence: 100%)
FabricaRepresentacoes.java (version 2) (support: 35%, confidence: 100%)

If you want to accept some suggestions, select the desired suggestions (use Ctrl or Shift to select more than one) and then click the "Accept the selected suggestions" button. Otherwise, just click the "Ignore suggestions and just establish the link" button.

Accept the selected suggestions
Ignore suggestions and just establish the link

Fig. 13 ArchTrace suggestion based on data mining

```
public void execute(Trace trace, byte action) throws TraceAbortedException {
  if ((action == ADD_ACTION) || (action == REMOVE_ACTION)) {
    if (trace.getArchitecturalElement().isImmutable()) {
      throw new TraceAbortedException("Immutable architectural elements can
                                          not have their traces changed.");
    }
  }
}
```

Fig. 14 Policy 2 algorithm

based on similarity to previously created sets of traceability links. Particularly, when a new architectural element is created that must be linked to existing source code configuration items, the developer has to create these traceability links one by one if the artifacts are scattered over different directories. If, however, the set of manually created links is similar to an existing set of links to some degree (i.e., above some threshold), then the policy will automatically suggest to include the rest of the traceability links of the existing set. Its user interface is shown in Figure 13. This policy is particularly useful when a new architectural element is created and initial traceability links need to be established.

4.5.1 Policy implementations

In this section, we detail some of our policies. The main purpose is twofold: (1) to illustrate how small in footprint policy implementations can be, and (2) to show how policies interact with the ArchTrace API to gather the necessary information for accomplishing their tasks.

Figure 14 presents the implementation of the *execute* method of policy 2. Each policy has 3 methods, as shown in Figure 11, but the other two of them are boilerplate

```java
public void execute(ArchitecturalElement ae, byte action) {
  if (action == ADD_ACTION) {
    for (ArchitecturalElement ancestry : ae.getAncestries()) {
      for (Trace trace : TraceManager.getInstance().getTraces(ancestry)) {
        ConfigurationItem ci = trace.getConfigurationItem();
        Trace newTrace = new Trace(ae, ci);
        if (ae.getArchitecture().addTrace(newTrace)) {
          GUIManager.getInstance().addPolicyMessage(newTrace + " added.");
        }
      }
    }
  }
}
```

Fig. 15 Policy 9 algorithm

```java
public void execute(Trace trace, byte action) throws TraceAbortedException {
  if (action == ArchTracePolicy.REMOVE_ACTION) {
    ConfigurationItem ci = trace.getConfigurationItem();
    if (ci.getLatestVersion().isBranch()) {
      throw new TraceAbortedException("Traces to configuration items in the
                                       trunk should persist after branching.");
    }
  }
}
```

Fig. 16 Policy 5 algorithm

(*getDescription* and *getRationale*). As this is a pre-trace policy, its main purpose is to prevent a certain situation from occurring, in this case the creation or removal[1] of traceability links on immutable architectural elements. Accordingly, the policy throws an exception if this is an immutable architectural element that is inputted via its parameters. This exception is handled by the *Policy Triggering* component, shown in Figure 7. The goal of this exception is to notify the *Policy Triggering* component to rollback the traceability creation or removal event. This is the way pre-trace policies indicate to ArchTrace that a proposed traceability link should not be established.

Figure 15 presents the implementation of the *execute* method of policy 9, which is responsible for copying all traceability links from a base version of an architectural element to its new version. This policy, thus, only deals with actions of the type *ADD_ACTION*, and then loops over all base versions of the architectural element (there may be more than one base version if the architectural element is the result of a merge operation), retrieves all traceability links of each such base version, creates corresponding traceability links for the new version, and reports its activities to the user. It is worth noticing that this policy does not throw an exception because is not a pre-trace policy (it cannot act as a constraint).

Finally, Figure 16 presents the algorithm of policy 5. This algorithm automatically denies the removal of traceability links from source code configuration items in the main line of development if a commit is performed to a branch. Again, this is a pre-trace policy, so it checks for the particular situation to occur and throws an exception if it does.

[1] Currently, ArchTrace supports no other actions beyond adding or removing traceability links, so the check is technically superfluous, but for future extensibility reasons and clarity we incorporated it in the code of the policy.

Overall, we note that the implementation of policies can be quite straightforward. None of our policies exceeds 150 total lines of code, and the essence of each policy is typically implemented in at most a few dozen lines of code. This is an artifact of the policy-based nature of ArchTrace, as well as the generic interfaces that abstract from the specifics of the architecture development environments and CM systems that are used.

4.6 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 2, to the final scenario after the changes, in Figure 4.

After the **first action** is performed by the developer, namely the creation of a new version of the *Print* component, ArchTrace receives an architectural evolution event. This event triggers policy 9, 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 9, 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 3 shows the scenario after the execution of policy 9.

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, an event is sent to ArchTrace, which triggers policy 10, creating a new traceability link from the *Toolbar* component (version 1.0) to *Command.java* (version 2.0). However, the execution of policy 10 triggers policy 6, which is responsible for removing the old traceability link from the *Toolbar* component (version 1.0) to *Action.java* (version 1.0).

Policy 10 is triggered three additional times for the same event. The second triggering of policy 10 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. The third triggering of policy 10 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 6, which removes the old traceability link from the *Print* component (version 2.0) to *Action.java* (version 1.0). Finally, the fourth triggering of policy 10 creates a new traceability link from the *Bus* connector (version 1.0) to *Command.java* (version 2.0). However, the execution of policy 10 triggers policy 6 again, which is responsible for removing the old traceability link from the *Bus* connector (version 1.0) to *Action.java* (version 1.0). This behavior is allowed by the pre-trace policy 2, because version 1.0 of the *Bus* connector is not immutable.

The **third action** concerns the interrelated evolution of the *Input* interface and *Input.java* source code. We stipulated in Section 2 that ArchTrace must behave in the same way independent from the order of commit. Suppose that the architecture is committed first. In this case, ArchTrace receives an architectural element evolution event. This event triggers policy 9, which is responsible for copying all traceability links from *Input* interface version 1.0 to the new, second version of the interface. This results in a traceability link from *Input* interface version 2.0 to *Input.java* version 1.0. Then, the source code commit triggers policy 10, creating a new traceability link from *Input* interface version 2.0 to *Input.java* version 2.0. However, the execution of policy 10 triggers policy 6, which is responsible for removing the old traceability link from the *Input* interface version 2.0 to *Input.java* version 1.0. Policy 10 is additionally triggered to evolve traceability links of *Input* interface version 1.0, but this action is prohibited by policy 2 due to the immutability of version 1.0.

Now suppose that the source code is committed first. ArchTrace receives an implementation evolution event. This event triggers policy 10 first, which is responsible for evolving the existing traceability links of *Input* interface 1.0 to the new version of the source code. However, this action is prohibited by policy 2 due to immutability of *Input* interface version 1.0. Thus, at this stage no new links are created. When the architectural change is committed, it triggers policy 9, which is responsible for copying all traceability links from *Input* interface version 1.0 to the new, second version of the same interface. Still, this does not result in the creation of links, because the triggering of policy 9 triggers policy 1, which recognizes that a newer version of *Input.java* is available (version 2.0). It, thus, suggests the establishment of the traceability link to *Input.java* version 2.0. As one can see, through a difference sequence of policies, the same results are applied as when the architectural change was committed first.

5 Evaluation

To evaluate the effectiveness of ArchTrace and its current set of policies, we executed a retrospective study of an existing system. This kind of study, in which we replayed past data from a real development project to simulate an actual development effort involving "live" developers, allowed us to analyze how our tool would perform without having to actually put the research tool into prolonged use. In fact, we could simulate in two weeks a two-year effort. The system under study, named Odyssey, is a software development environment being developed at the Federal University of Rio de Janeiro 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.

The next sections detail our planning of the retrospective study, our preparation of the environment for the study, the mechanism we used to gather statistics, the execution of the study, and the qualitative and quantitative analysis of the results that we obtained.

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.

Below, we discuss each of these steps in more detail.

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, 3, 5 and 8. Policy 5 is designed to work with branches, but the case study involved only development on the main trunk. Policy 3 is not designed to operate together with policies 6 and 10, as the effect is either preventive (policy 3) or proactive (policies 6 and 10), and we chose a proactive approach (others may choose a more cautious route, in just using policy 3). Policy 1 and 8 are designed to operate in an interactive manner,

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

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 theoretically can be achieved.

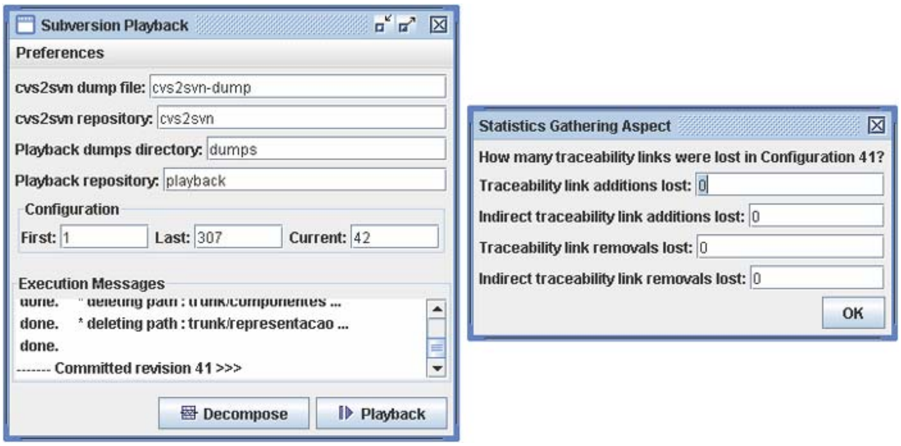
5.3 Statistics gathering

This retrospective study aims to analyze different statistics gathered from the ArchTrace execution. To allow this automatic gathering, we implemented a statistics gathering aspect (Kiczales et al. 1997) 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


Subversion Playback

Preferences

cvs2svn dump file: cvs2svn-dump

cvs2svn repository: cvs2svn

Playback dumps directory: dumps

Playback repository: playback

Configuration

First: 1 Last: 307 Current: 42

Execution Messages

```
done.   * deleting path : trunk/componentes ...
done.   * deleting path : trunk/representacao ...
done.
------- Committed revision 41 >>>
```

Statistics Gathering Aspect

How many traceability links were lost in Configuration 41?

Traceability link additions lost: 0

Indirect traceability link additions lost: 0

Traceability link removals lost: 0

Indirect traceability link removals lost: 0

Fig. 17 Incremental check-in playback

that we explicitly wrote to submit, check-in by check-in, the accumulated version history of Odyssey, shown in Figure 17. 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 releases: 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.

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 18. 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.

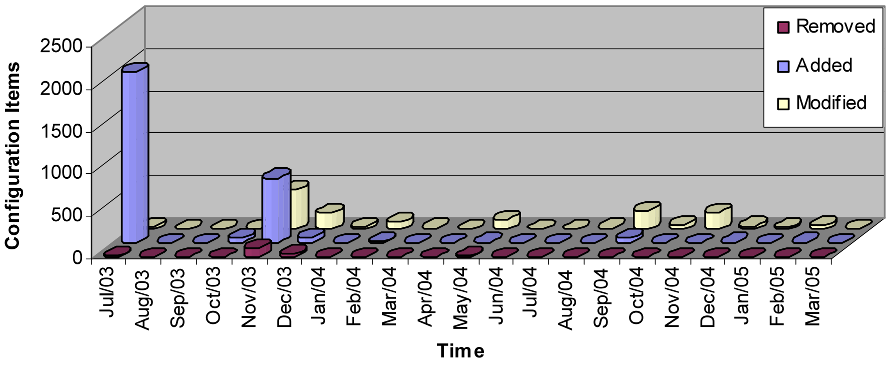


Fig. 18 Configuration items evolution

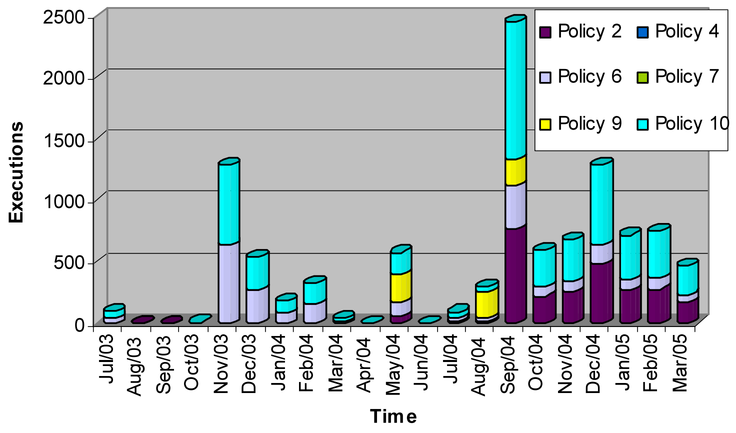


Fig. 19 Execution of different policies

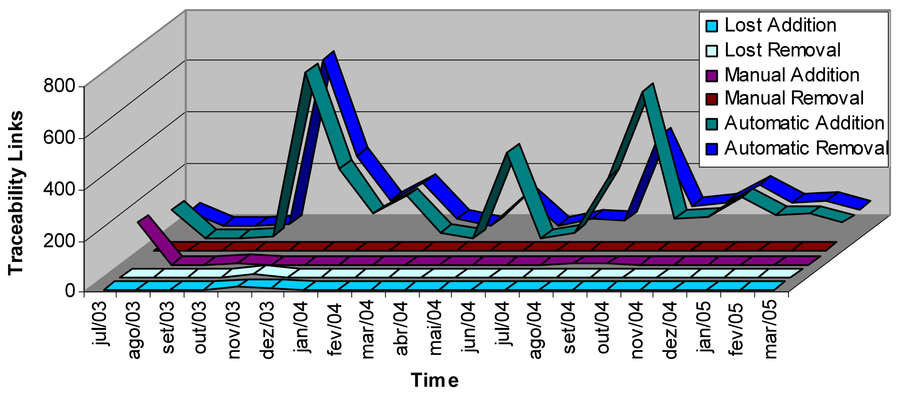


Fig. 20 Traceability links evolution

The results of which policies were active during the study are shown in Figure 19. As expected, policies 2, 6, and 10 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). But several interesting events took place that led to the involvement of other policies as well. First, during the initial detection of the proper traceability links between the Odyssey architecture and its source code on July, 2003, policy 8 was explicitly enabled to help with the otherwise manual effort of identifying an initial set of traceability links. While, as stated before, this problem is outside the scope of this paper, the use of policy 8 illustrates that techniques such as data mining can be effectively incorporated in ArchTrace and can add value. After this initial phase, though, we disabled policy 8.

The second interesting even took place in November 2003, as indicated by the spike in the number of times policies were executed. 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 6 and 10 dealt with this situation by updating traceability links to reflect the new organization of the source code. Figure 19 shows that only policies 6 and 10 were needed to support the reorganization, and Figure 20 shows that those two policies automatically added and removed many traceability links while only losing a few.

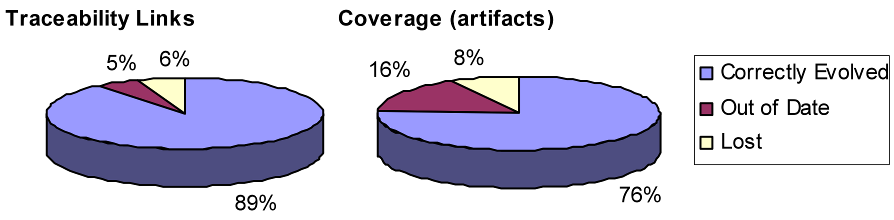
Policy 9, which is responsible for copying existing traceability links to new versions of architectural elements, was triggered on May, August, and September, 2004, meaning the three updates to the Odyssey architecture. Policy 2 was frequently triggered to deny the evolution of traceability links related 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 21 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 ($|T_i - T_e|$), 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 (Baeza-Yates and Ribeiro-Neto 1999): 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		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%

Fig. 21 Quantitative analysis summary

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 were 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).

5.7 Threats to validity

Despite our best efforts to create an experiment free of bias, there are some aspects of the study that may have affect the general applicability of our results. Here, we discuss the main threads to validity.

First, we observe that the study was performed over a relatively stable system and only included data covering activities that took place after Odyssey development had been in progress for several years. It is unclear how the current set of policies would perform on a new project in which rapid growth in the number of artifacts and constant reorganization may take place, especially if the project is agile in nature. Additional study of our policies in such settings, and potential development of additional policies, is necessary to broaden our results to such projects.

Second, the study is retroactive in nature. On the one hand, this is a strength, as it analyzes an actual sample of a development project without interference of the researchers. On the other hand, the developers were not exposed to the benefits and potential drawbacks of ArchTrace, and hence their behavior in evolving the artifacts may not accurately reflect what they would have done had ArchTrace been available. A second study with developers actively using ArchTrace is necessary to understand whether or not our results hold, particularly as to whether developers are capable of selecting the “right” policies.

Third, two of the researchers were involved in the development of Odyssey and were responsible for establishing the initial and ideal sets of traceability links. We therefore turned off all interaction of the policies with the researchers, so not to provide better input than other developers would have provided. By the same token, wrong guidance by actual developers could have seriously hampered effective operation of ArchTrace. Again, further study with interactive policies and developer responses is needed.

Finally, a threat exists in that our policies thus far do not exercise all scenarios of software evolution. Specifically, we do not cover branching. We believe this is not a serious threat, as an informal examination of this problem gives us confidence we can develop additional policies that will appropriately address branching. Thus, not covering branching is a limitation of the policies thus far, and not of the approach.

5.8 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 currently cannot deal. 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. Another option would have been to create a new implementation evolution policy that detects when a user tries to remedy an erroneous deletion and then automatically reinserts the previously existing links.

At other times, some traceability links were lost when new artifacts were introduced completely out of context of the existing artifacts. In this situation, data mining policies are also not useful because brand new artifacts do not have historical information to be analyzed. A possible solution to address this problem is the construction of a policy that employs information retrieval techniques (De Lucia et al. 2004) or syntactical analysis (Briand et al. 2003) 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.

It is important to note the interplay between pre-trace and post-trace policies. It would be possible to implement the same kind of functionality using only pre-trace or only post-trace policies, but doing so would lead to much duplication and context checking across policies. Particularly, each of our pre-trace policies would need to be replicated in every existing post-trace policy, or each of the post-trace policies actions would need to be selectively included in some of the pre-trace policies, neither of which is a desired solution. The approach of separate pre-trace and post-trace policies, which dynamically collaborate as needed, is a more elegant solution that integrally promotes reuse and separation of concerns.

6 Related work

Some approaches integrally combine the architecture definition with the source code, avoiding the need for traceability links. For instance, ArchJava (Aldrich et al. 2002) enhances the Java programming language with special keywords to integrate an architecture description inside the source code. Similarly, XDoclet (Walls and Richards 2003) uses source code annotations to define EJB components. Clearly, these kinds of approaches have their value. However, many situations require architectural representations separate from the source code (Ommering et al. 2000; Kruchten 2001). 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. (2004) 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 (2002), Nentwich et al. (2003), and

Abi-Antoun et al. (2005) 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, such as Chimera (Anderson et al. 1994) and Molhado (Nguyen et al. 2004). 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 Conclusions

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 its policies, this is exactly what ArchTrace does – and it achieves high-quality results in both precision and recall.

While promising as a new kind of technique for managing the evolution of traceability links, our work to date also highlights that additional work remains to be done. First and foremost, we recognize that achieving 100% precision and 100% recall is the ultimate result to be achieved by ArchTrace. This, however, may or may not be unrealistic. On the one hand, no set of policies can anticipate every single potential change made in every single situation by every single developer. On the other hand, however, this is perhaps also not needed: it is merely necessary to be able to match as closely as possible the working style and conventions of a group of developers. This is a much smaller problem, and becomes one of having a sufficiently broad set of policies available and providing developers strong guidance in selecting the policies appropriate for them. Through the building of a policy portfolio and further empirical studies, both retroactive and active (i.e., in a live development setting), we plan to build an understanding of whether ArchTrace can be made an effective and reliable solution for traceability management in the face of evolution and indeed provide the necessary guarantee that its resulting sets of traceability links do not contain false positives or false negatives.

Another focus of our future work concerns conflicting policies. Currently, the user is responsible for ensuring they choose a set that does not conflict. If they happen to choose a conflicting set of policies, ArchTrace cannot guarantee its results and may even exhibit race conditions or infinite loops. To address this issue, we plan to research the use of meta-policies, which act as arbitrators, and build analyses that can determine, at the moment of their activation, whether policies conflict.

An additional issue that we would like to address is branching. Policy 5 was designed to avoid interference of versioning actions on branches with respect to the main line of development. This, however, represents merely a first step. While we can support keeping branches isolated, we still need to support the creation of branches and the explicit merging of branches into the main line of development.

Finally, we mention that the long-term goal of our work is not just to maintain traceability links, but to put these traceability links to good use. While an accurate trace is of help to humans in understanding the system at hand, it is also a first step towards automation of various processes at the architectural level of abstraction. Our specific efforts will focus on architecture-based build and release mechanisms, allowing a developer to drive the build and release process from the architectural specification. In today's world of component-based software development, these two processes are particularly critical and suited for a new architectural slant.

Acknowledgements This work was 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

Abi-Antoun, M., Aldrich, J., Garlan, D., Schmerl, B., Nahas, N.: Semi-automated incremental synchronization between conceptual and implementation level architectures. In: Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 265–268, Pittsburgh, PA, USA, November 2005

Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: International Conference on Very Large Data Bases (VLDB), pp. 487–499, Santiago de Chile, Chile, September 1994

Aldrich, J., Chambers, C., Notkin, D.: ArchJava: connecting software architecture to implementation. In: International Conference on Software Engineering (ICSE), pp. 187–197, Orlando, USA, May 2002

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

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng. **28**(10), 970–983 (2002)

Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. ACM, New York (1999)

Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley, Reading (2000)

Briand, L.C., Labiche, Y., O'Sullivan, L.: Impact analysis and change management of UML models. In: International Conference on Software Maintenance (ICSM), pp. 256–265, Amsterdam, Netherlands, September 2003

Chen, P., Critchlow, M., Garg, A., Westhuizen, C., Van der Hoek, A.: Differencing and merging within an evolving product line architecture. In: International Workshop on Product Family Engineering, pp. 269–281, Siena, Italy, November 2003

Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M., O'Reilly, J.: Version Control with Subversion (2004)

Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Comput. Surv. **30**(2), 232–282 (1998)

Dashofy, E., Van der Hoek, A., Taylor, R.N.: A highly-extensible, XML-based architecture description language. In: Working IEEE/IFIP Conference on Software Architectures (WICSA), pp. 103–112, Amsterdam, Netherlands, August 2001

Dashofy, E., Van der Hoek, A., Taylor, R.N.: An infrastructure for the rapid development of XML-based architecture description languages. In: International Conference on Software Engineering (ICSE), pp. 266–276, Orlando, FL, USA, May 2002

De Lucia, A., Fasano, F., Oliveto, R., Tortora, G.: Enhancing an artefact management system with traceability recovery features. In: International Conference on Software Maintenance (ICSM), pp. 306–315, Chicago, IL, USA, September 2004

Eclipse Foundation: Eclipse IDE. http://www.eclipse.org (accessed 29 September 2007)

Garg, A., Critchlow, M., Chen, P., Van der Westhuizen, C., Van der Hoek, A.: An environment for managing evolving product line architectures. In: International Conference on Software Maintenance (ICSM), pp. 358–367, Amsterdam, Netherlands, September 2003

Huffman Hayes, J., Dekhtyar, A., Osborne, J.: Improving requirements tracing via information retrieval. In: International Conference on Requirements Engineering (RE), pp. 138–147, Monterey, USA, September 2003

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: European Conference on Object-Oriented Programming (ECOOP), pp. 220–242, Jyväskylä, Finland, June 1997

Kruchten, P.: The Rational Unified Process: an Introduction. Addison-Wesley, Reading (2001)

Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: International Conference on Software Engineering (ICSE), pp. 125–135, Portland, OR, USA, May 2003

Medvidovic, N., Rosenblum, D.S.: Domains of concern in software architectures and architecture description languages. In: Conference on Domain-Specific Languages, pp. 199–212, Santa Barbara, USA, October 1997

Muccini, H., Van der Hoek, A.: Towards testing product line architectures. In: International Workshop on Testing and Analysis of Component Based Systems, pp. 111–121, Warsaw, Poland, April 2003

Nentwich, C., Emmerich, W., Finkelstein, A., Ellmer, E.: Flexible consistency checking. ACM Trans. Softw. Eng. Methodol. **12**(1), 28–63 (2003)

Nguyen, T.N., Munson, E.V., Boyland, J.T.: The molhado hypertext versioning system. In: Conference on Hypertext and Hypermedia, pp. 185–194, Santa Cruz, USA, August 2004

Ommering, R.V., Linden, F.V.D., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. IEEE Comput. **33**(6), 78–85 (2000)

Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: International Conference on Software Engineering (ICSE), pp. 177–186, Kyoto, Japan, April 1998

Reiss, S.P.: Constraining software evolution. In: International Conference on Software Maintenance (ICSM), pp. 162–171, Montreal, Canada, October 2002

Richardson, D.J., Wolf, A.L.: Software testing at the architectural level. In: International Software Architecture Workshop (ISAW), pp. 68–71, San Francisco, USA, October 1996

Settimi, R., Cleland-Huang, J., Khadra, O.B., Mody, J., Lukasik, W., Depalma, C.: Supporting software evolution through dynamically retrieving traces to UML artifacts. In: International Workshop on Principles of Software Evolution (IWPSE), pp. 49–54, Kyoto, Japan, September 2004

Shaw, M., Deline, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G.: Abstractions for software architecture and tools to support them. IEEE Trans. Softw. Eng. **21**(4), 314–335 (1995)

Shirabad, J.S., Lethbridge, T., Matwin, S.: Supporting software maintenance by mining software update records. In: International Conference on Software Maintenance (ICSM), pp. 22–31, Florence, Italy, November 2001

Van der Hoek, A.: Design-time product line architectures for any-time variability. Sci. Comput. Program. **53**(3), 285–304 (2004)

Walls, C., Richards, N.: XDoclet in Action. Manning Publications (2003)

Werner, C.M.L., Mangan, M.A.S., Murta, L.G.P., Souza, R.P., Mattoso, M., Braga, R.M.M., Borges, M.R.S.: OdysseyShare: an environment for collaborative component-based development. In: IEEE Conference on Information Reuse and Integration (IRI), pp. 61–68, Las Vegas, USA, October 2003

Whitehead, E.J.: An analysis of the hypertext versioning domain. Ph.D. thesis, University of California, Irvine, USA (2000)

Ying, A.T.T., Murphy, G.C., Ng, R., Chu-Carroll, M.C.: Predicting source code changes by mining change history. IEEE Trans. Softw. Eng. **30**(9), 574–586 (2004)

Zhao, J., Yang, H., Xiang, L., Xu, B.: Change impact analysis to support architectural evolution. J. Softw. Maintenance: Res. Pract. **14**(5), 317–333 (2002)

Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: International Conference on Software Engineering (ICSE), pp. 563–572, Edinburgh, Scotland, May 2004