ArchTrace: A Tool for Keeping in Sync Architecture and its Implementation

Leonardo G. P. Murta[1], André van der Hoek[2], Cláudia M. L. Werner[1]

[1]COPPE – Federal University of Rio de Janeiro (UFRJ)
P.O. Box 68511 – 21945-970 – Rio de Janeiro – RJ – Brazil

[2]Department of Informatics – University of California, Irvine (UCI)
444 Computer Science Building – 92697-3440 – Irvine – CA – USA

{murta, werner}@cos.ufrj.br, andre@ics.uci.edu

Abstract. *During software evolution, the designed architecture usually loses sync to its source code. This problem creates a gap between the high-level specification of systems and their actual implementation. We present a tool, named ArchTrace, for the creation and maintenance of traceability among architectures and their implementations. Our tool allows architectural elements and source code to evolve separately, but provides an extensible infrastructure that employs nine pluggable policies to update the traceability links as a response to their evolution.*

1. Introduction

Software architectures are intended to represent a system at high abstraction levels, allowing complexity control and serving as the basis for further development. Moreover, software architectures are useful for quality attribute assessment, stakeholder communication and software product lines [3].

The architecture of software systems is typically described via an Architectural Description Language (ADL), which uses components, interfaces, and connectors as first-class elements. However, control over the evolution of software architectures is still a challenge. When architectures evolve, their implementations should evolve in the same way to avoid inconsistencies. This problem, named architectural erosion, may threaten the benefits of software architectures due to divergences among the high-level specification of systems and their actual implementation.

The goal of this paper is to present a tool, named ArchTrace, which manages traceability links among architectural elements and source-code. Furthermore, our tool provides a policy-based infrastructure to evolve the traceability links as a response to the evolution of architectural elements or source-code. This infrastructure is composed of nine built-in policies and may be extended through the construction of new policies, allowing complex behavior via the compound execution of policies. For example, the evolution of architectural elements or source-code triggers the execution of some pluggable policies. However, the execution of these policies may trigger the execution of other policies, recursively.

The rest of this paper is organized as follows. Section 2 presents some related work. Section 3 introduces the high level design of ArchTrace, followed by a usage example in Section 4. Section 5 details the ArchTrace implementation, and we conclude the paper in Section 6 with an outlook of our future work.

2. Related Work

Several different approaches already address the problem of maintaining traceability between an architectural description and its corresponding source code artifacts. 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 artifacts are perfectly traceable because one is embedded into the other. For instance, ArchJava [1] and XDoclet [11] embed the definition of architectural elements in the source code. While this kind of solution is effective in maintaining 100% accuracy, it is not realistic, since it is often the case that the software architecture is maintained in a structure that resides apart from source code, with different people using different tools and different notations maintaining the two.

Data mining [13], information retrieval [2], and syntactic analysis [4] 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.

3. High-level Design

ArchTrace falls in between equality by definition and after the fact reconstruction categories. We classify it as an *instant update* approach. It allows the creation and continuous evolution of traceability links among architectural elements and their implementation. The philosophy behind ArchTrace is that a group of people is responsible for developing and maintaining architectures, and another group of people implements this architecture using specific programming languages. Architects have their own tools to define and evolve architectures, such as the ArchStudio environment [7]. On the other hand, developers also have their own tools to do their work, such as the Eclipse IDE [8] and Subversion [5]. Moreover, ArchTrace provides a policy-based infrastructure that listens to notifications of changes in architectural elements and source code (actually, any kind of configuration items), as shown in Figure 1. These notifications trigger any applicable policy to update the traceability links, and these policies may trigger other policies, recursively. An important aspect of ArchTrace is that it is pluggable with respect to the set of traceability management policies that it uses. We have already implemented nine such policies, but other policies can easily be coded and used. Additional information regarding ArchTrace approach can be found at [9].

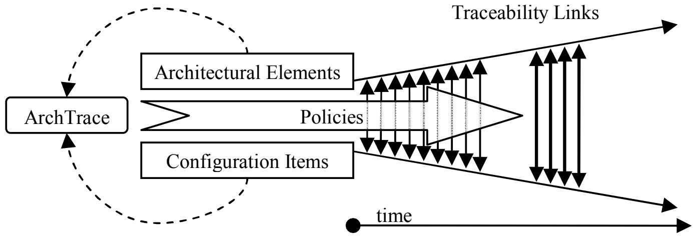
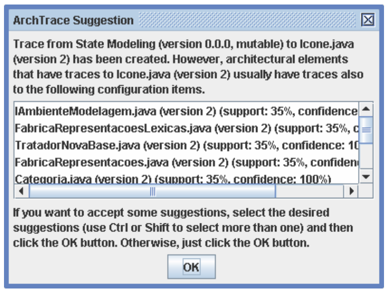


Figure 1. Traceability links evolution via policy triggering

4. Usage Example

We applied ArchTrace to evolve the traceability links of the Odyssey environment [12]. First, we manually established an initial set of traceability links. During this step, one of the ArchTrace built-in policies, the data-mining policy, detected a pattern in the links creation and warned us, as shown in Figure 2. It is important to notice that this step could be performed with the support of any approach discussed in Section 2.



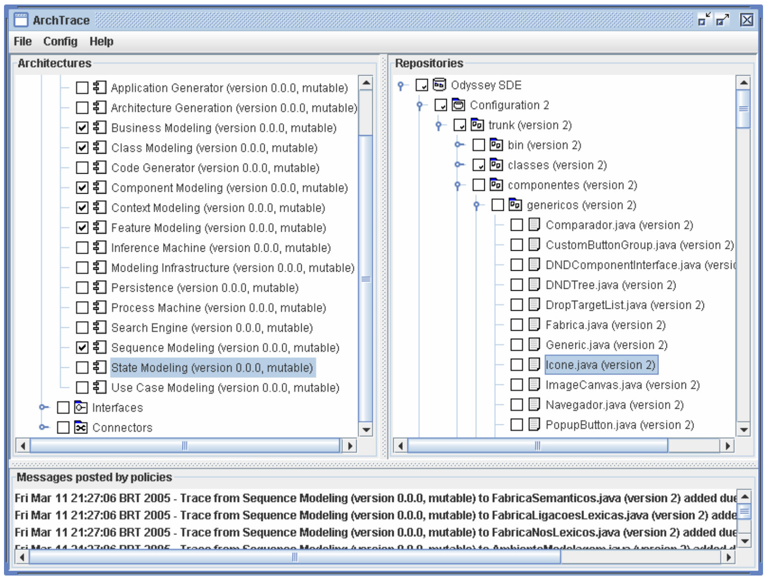
Trace from State Modeling (version 0.0.0, mutable) to Icone.java (version 2) has been created. However, architectural elements that have traces to Icone.java (version 2) usually have traces also to the following configuration items.

IAmbienteModelagem.java (version 2) (support: 35%, confidence
FabricaRepresentacoesLexicas.java (version 2) (support: 35%, c
TratadorNovaBase.java (version 2) (support: 35%, confidence: 10
FabricaRepresentacoes.java (version 2) (support: 35%, confiden
Categoria.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 OK button. Otherwise, just click the OK button.

Figure 2. Data-mining policy in action

After establishing the initial traceability links between the Odyssey architecture and its source code, we were able to query these links in a bidirectional fashion: it is possible to select an architectural element and visualize all source code artifacts that implement the element, and vice-versa. It is important to notice that usually only one architectural element "owns" a given source code artifact, but our tool does not impose this restriction. Figure 3 shows ArchTrace being used to query traceability links.



Architectures

- ☐ 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)
- ☑ 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)
- ☐ Search Engine (version 0.0.0, mutable)
- ☑ Sequence Modeling (version 0.0.0, mutable)
- ☐ State Modeling (version 0.0.0, mutable)
- ☐ Use Case Modeling (version 0.0.0, mutable)
- ☐ Interfaces
- ☐ Connectors

Repositories

- ☑ Odyssey SDE
  - ☑ Configuration 2
    - ☑ trunk (version 2)
      - ☐ bin (version 2)
      - ☑ classes (version 2)
      - ☐ componentes (version 2)
        - ☐ genericos (version 2)
          - ☐ Comparador.java (version 2)
          - ☐ CustomButtonGroup.java (version 2)
          - ☐ DNDComponentInterface.java (versi
          - ☐ DNDTree.java (version 2)
          - ☐ DropTargetList.java (version 2)
          - ☐ Fabrica.java (version 2)
          - ☐ Generic.java (version 2)
          - ☐ Icone.java (version 2)
          - ☐ ImageCanvas.java (version 2)
          - ☐ Navegador.java (version 2)
          - ☐ PopupButton.java (version 2)

Messages posted by policies

```
Fri Mar 11 21:27:06 BRT 2005 - Trace from Sequence Modeling (version 0.0.0, mutable) to FabricaSemanticos.java (version 2) added due
Fri Mar 11 21:27:06 BRT 2005 - Trace from Sequence Modeling (version 0.0.0, mutable) to FabricaLigacoesLexicas.java (version 2) adde
Fri Mar 11 21:27:06 BRT 2005 - Trace from Sequence Modeling (version 0.0.0, mutable) to FabricaNosLexicos.java (version 2) added due
```

Figure 3. ArchTrace screenshot

Finally, Archtrace updates traceability links when a check-in is performed on either ArchStudio or Subversion. The bottom of Figure 3 shown some policy messages presented after a source code check-in in Subversion. In this case, a policy was triggered just after the check-in. This policy updated the existing traceability links to the new versions of the source code. However, another policy denied this action for immutable architectural elements (immutable architectural elements represent configurations of the system in a fixed moment in time, and should not be changed). Finally, a third policy removed the old traceability links from previous versions of the source code.

5. Implementation Details

ArchTrace uses xADL 2.0 [6] to describe software architectures. One of the main reasons for choosing xADL 2.0 is its extensibility mechanism through XML Schemas. xADL 2.0 is composed of a set of schemas and each of them can be extended to include new features. 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 4. Specifically, we support the tagging of architectural elements with a series of configuration items.

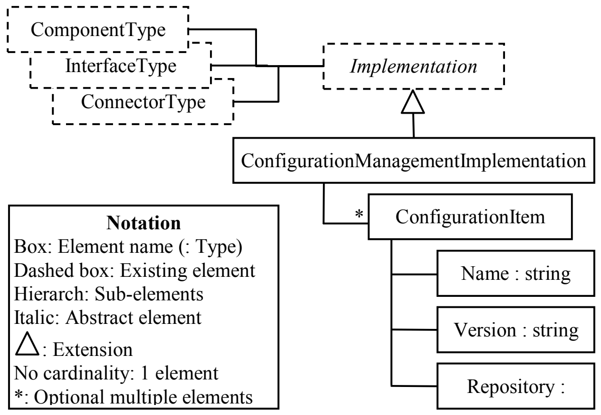
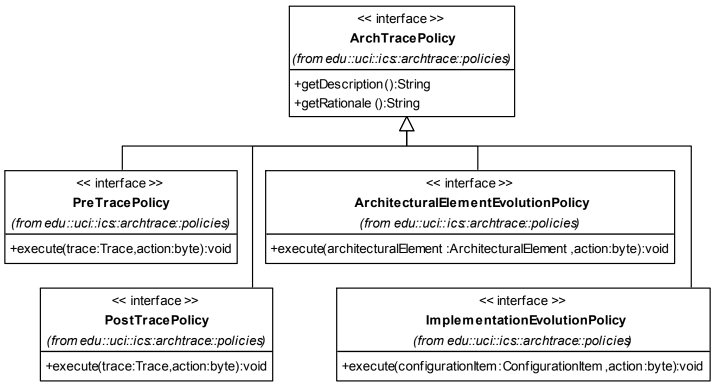


Figure 4. Proposed xADL Schema

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 a *Print* component, version 2.0, can be described via our schema using the information shown in the following two tuples: <"Model/Printer.java", 1.0, svn://server/src> and <"Controller/Command.java", 2.0, svn://server/src>.

Our tool distinguishes four types of policies: *architectural element evolution* policies, *implementation evolution* policies, *pre-trace* policies, and *post-trace* policies. The *architectural element evolution* policies are executed if some architectural element has been added, removed or changed. The most common change in an architectural element is related to its mutable state and sub-architecture. The *implementation evolution* policies are executed when new versions of source code are created in the repository. The main usage of policies of this type is to update existing traceability links when source code evolves. The *pre-trace* policies are executed just before a traceability link is added or removed. This type of policy allows detection of inconsistencies among the traceability link that is being created or removed and other traceability links that already exist. Finally, the *post-trace* policies are activated after the creation or removal of a traceability link. This allows the definition of policies that update other traceability links when some traceability links are effectively added or removed. The main difference between pre-trace and post-trace policies is that post-trace policies cannot rollback the action. Moreover, post-trace policies run if, and only if, the traceability link was "committed" by all pre-trace policies. Each ArchTrace policy is implemented as a Java class that follows a specific interface, shown in Figure 5.



<< interface >>
ArchTracePolicy
(from edu::uci::ics::archtrace::policies)
+getDescription():String
+getRationale ():String
<< interface >>
PreTracePolicy
(from edu::uci::ics::archtrace::policies)
+execute(trace:Trace,action:byte):void
<< interface >>
ArchitecturalElementEvolutionPolicy
(from edu::uci::ics::archtrace::policies)
+execute(architecturalElement :ArchitecturalElement ,action:byte):void
<< interface >>
PostTracePolicy
(from edu::uci::ics::archtrace::policies)
+execute(trace:Trace,action:byte):void
<< interface >>
ImplementationEvolutionPolicy
(from edu::uci::ics::archtrace::policies)
+execute(configurationItem:ConfigurationItem ,action:byte):void

Figure 5. Policies API

The set of nine build-in policies of ArchTrace is composed of one architecture evolution policy, one implementation evolution police, four pre-trace policies, and three post-trace polices, as shown in Table 1.

Table 1. Built-in policies

TYPE	POLICY
Arch evol	Copies all existing traceability links to the new version of the architectural element.
Impl evol	Automatically updates traceability links when a new version of a configuration item is available.
Pre-trace	Suggests traceability links to more recent configuration item version if a traceability link is created to older version.
	Denies traceability links creation or removal on immutable architectural elements.
	Denies traceability links creation to more than one version of the same configuration item.
	Denies traceability link creation to sub configuration items if the composite configuration item is already traced.
Post-trace	Removes traceability links from old configuration item versions when a traceability link is created to a newer version.
	Removes traceability links from sub configuration items if a traceability link is created to a composite conf. item.
	Suggests related traceability links when a pattern of traceability links creation is detected via data mining.

6. Conclusion

This paper has presented a tool for the creation and evolution of traceability links among architectural elements and source code. Some positive aspects of this approach are the graphical support for traceability links establishment, querying and evolution, and the policy-based infrastructure, together with nine built-in policies.

Some of our future works include the use of ArchTrace to leverage configuration management commands to the architectural level, the adoption of new functionalities to allow build, release, and deploy driven by architectural elements, and the addition of other configuration management systems, such as Odyssey-VCS [10], that are based on high level meta-models. This last feature would allow, for example, traceability links among components and the UML artifacts that internally model the components.

7. Acknowledgments

This work is sponsored in part by NSF grants CCR-0093489 and IIS-0205724. We also would like to thank CAPES for the financial support under grant BEX0323/04-7.

8. References

1. Aldrich, J., Chambers, C. and Notkin, D., ArchJava: Connecting Software Architecture to Implementation. in *International Conference on Software Engineering*, (Orlando, USA, 2002), 187-197.
2. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A. and Merlo, E. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering*, *28* (10). 970-983.
3. Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison Wesley, 2000.
4. Briand, L.C., Labiche, Y. and O'Sullivan, L., Impact Analysis and Change Management of UML Models. in *International Conference on Software Maintenance*, (Amsterdam, Netherlands, 2003), 256-265.
5. Collins-Sussman, B., Fitzpatrick, B.W. and Pilato, C.M. *Version Control with Subversion*. O'Reilly, 2004.
6. Dashofy, E., Hoek, A. and Taylor, R.N., A Highly-Extensible, XML-Based Architecture Description Language. in *Working IEEE/IFIP Conference on Software Architectures*, (Amsterdam, Netherlands, 2001), 103-112.
7. Dashofy, E., Hoek, A. and Taylor, R.N., An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages. in *International Conference on Software Engineering*, (Orlando, Fl, 2002), 266-276.
8. Eclipse Foundation. Eclipse 3.1, 2006.
9. Murta, L.G.P., van der Hoek, A. and Werner, C.M.L., ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links. in *International Conference on Automated Software Engineering (ASE)*, (Tokyo, Japan, 2006).
10. Oliveira, H.L.R., Murta, L.G.P. and Werner, C.M.L., Odyssey-VCS: a Flexible Version Control System for UML Model Elements. in *International Workshop on Software Configuration Management*, (Lisbon, Portugal, 2005), 1-16.
11. Walls, C. and Richards, N. *XDoclet in Action*. Manning Publications, 2003.
12. Werner, C.M.L., Mangan, M.A.S., Murta, L.G.P., Souza, R.P., Mattoso, M., Braga, R.M.M. and Borges, M.R.S., OdysseyShare: an Environment for Collaborative Component-Based Development. in *IEEE Conference on Information Reuse and Integration*, (Las Vegas, USA, 2003), 61-68.
13. Zimmermann, T., Weisgerber, P., Diehl, S. and Zeller, A., Mining version histories to guide software changes. in *International Conference on Software Engineering*, (Edinburgh, Scotland, 2004), 563-572.