ArchEvol: Versioning Architectural-Implementation Relationships

Eugen C. Nistor, Justin R. Erenkrantz,
Scott A. Hendrickson, and André van der Hoek

University of California, Irvine
Donald Bren School of Information and Computer Sciences
Department of Informatics
Irvine, CA 92697-3425 USA
{enistor, jerenkra, shendric, andre}@ics.uci.edu

Abstract. Previous research efforts into creating links between software architecture and its implementations have not explicitly addressed versioning. These earlier efforts have either ignored versioning entirely, created overly constraining couplings between architecture and implementation, or disregarded the need for versioning upon deployment. This situation calls for an explicit approach to versioning the architecture-implementation relationship capable of being used throughout design, implementation, and deployment. We present ArchEvol, a set of xADL 2.0 extensions, ArchStudio and Eclipse plug-ins, and Subversion guidelines for managing the architectural-implementation relationship throughout the entire software life cycle.

1 Introduction

Software architecture provides a high-level, abstract view of a system where it is easier to identify and reason about a system's main computational parts (the components), the ways through which they interact (the connectors), and their configuration [1]. In the early stages of development an architecture is typically used as a communication tool to provide understanding to other developers but it is also often analyzed to determine or ensure specific properties of the resulting system [2].

The benefits of having an architecture in place can extend beyond the initial stages if a mapping between the architectural model and the implementation can be maintained throughout development. If an architecture describes properties that are not accurate, then there is no point in analyzing the architectural model in the first place, since the results of the analysis cannot be guaranteed. But even if the consistency is initially ensured, changes in the implementation can lead to changes in architectural properties, leading to a phenomenon known as architectural erosion [1]. Architectures are described using specialized architecture description languages (ADLs), while implementations are described using programming languages. During regular development, new requirements might determine the necessity of branching the architecture and the associated

SCM 2005, September 5-6, 2005, Lisbon, Portugal.

implementation of one or more components. In this case, we want to be able to accurately determine which versions of the component implementations belong to the initial version of the architecture and which belong to the branched version of the architecture. ArchEvol defines mappings between architectural descriptions and component implementations using a versioning infrastructure and addresses the evolution of the relationship between versions of the architecture and versions of the implementation.

Maintaining mappings between different versions of architecture and implementation is an issue that has not been addressed adequately to date. Some SCM systems try to blur the distinction between repository configuration and architectural configuration. Adele [3] and the Cedar System Modeller [4] provide consistency support between a description of a system in a module interconnection language and the underlying module implementations. However, their solution is enabled by having implementation, system description and configuration management support in the same system, and limits their development to the capabilities of the particular tools offered. Architectural description languages have evolved from module interconnection languages to include various new structural and behavioral properties, and thus need specialized tools that support their development and analysis beyond configuration descriptions. Popular configuration management approaches today can maintain versions of implementation and architecture as closed-semantics files and directories, just like they would keep any other type of software artifact, but lack the capability of explicitly maintaining a mapping between them [5][6]. Architecture-based development approaches have maintained mappings between single versions of the architecture and implementation [7][8]. Other approaches, such as MAE, have focused on versioning of the architecture, but do not integrally support the evolution of the implementation nor the mapping of architecture to implementation [9].

Our approach centers around promoting strong interconnections between the architecture, implementation, and the versioning sub-systems while still allowing their independent development. Not only are the architecture and implementation described using different languages and concepts, but there are different tools and environments that are used for creating and maintaining them. We do not propose one single tool to perform all tasks, but instead propose adopting the right tool available for the individual task and facilitating its interactions with the other tools. Furthermore, the way the tools are used and integrated commands an associated process. The links used in ArchEvol can be used to support decentralized development of architectures and component implementations by different parties. The architecture and the individual components can be developed in parallel and the common information between the two can be synchronized at certain points in time.

ArchEvol specifically builds upon three previously existing tools: ArchStudio [10], Eclipse [11], and Subversion [6]. Our solution could be applied to any combination of architectural development, source code development and configuration management systems, but the three that we chose have particular features that makes their integration easier. ArchStudio provides the facilities for managing ar-

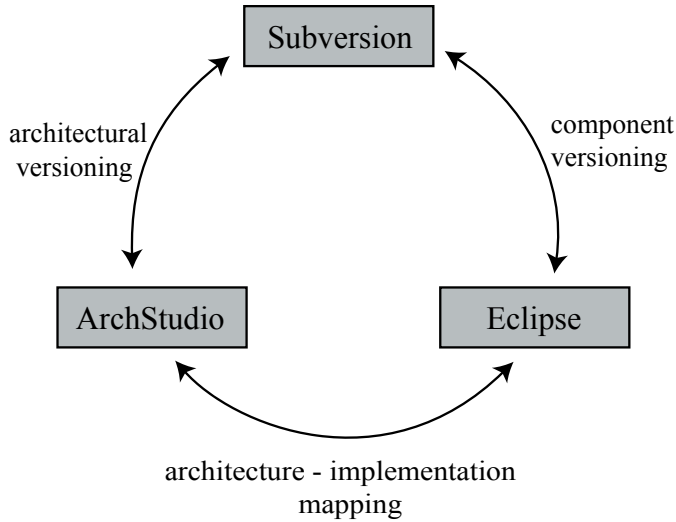

Fig. 1. ArchEvol Overview.

chitectures, through its XML-based xADL2.0 architecture description language [12]. Eclipse is an extensible development environment with comprehensive support for source code manipulation, used to manage implementations. Subversion is an SCM system that provides a WebDAV-compliant repository, and it is used to store evolving versions of the architecture and implementation. Our solution requires the creation and maintenance of a versioned set of hyperlinks between artifacts that did not previously exist, and its main contribution lies in maintaining the mapping and ensuring that the right versions of the architectural components map onto the right versions of the code (and vice versa).

2 Approach

ArchEvol provides an integrated approach for maintaining an accurate architectural model which is consistently mapped to implementation throughout the development process. This is achieved by integrating ArchStudio, an architectural environment, Eclipse, a development environment, and Subversion, a WebDAV-centric software configuration management system. While these three tools work well independently, we have created an additional layer of infrastructure and a process that allows these tools to work together to offer end-to-end automated

support for architecture-centric evolution that focuses on the identified problem of maintaining the mapping among an evolving architecture and its evolving code.

2.1 Architectural-implementation mapping

An architectural model describes properties of the components and connectors in an abstract manner. However, these architectural entities must eventually be implemented using a programming language in order to realize this abstraction. Therefore, the first step of the ArchEvol process is to integrate the ArchStudio and Eclipse environments by enabling design-time coordination between the design of the architectural model and the implementation of the components.

The integration of ArchStudio and Eclipse presented two challenges that we had to resolve in order to be successful. First, we had to create the necessary extensions to ArchStudio and Eclipse to support bi-directional communication. Secondly, and more importantly, we also had to construct a mapping between their different elements: components and connectors are the primary extensibility focus in ArchStudio, while Eclipse centers on projects, packages and classes.

Creating extensions. Both ArchStudio and Eclipse were intended to be easily extensible development environments. However, they have taken different philosophical approaches to extensibility: Eclipse allows extension through plug-ins that implement pre-defined interfaces, while ArchStudio uses loosely coupled tools that interact through exchanges of pre-defined event types. In order to have the two environments communicate, we implemented an Eclipse plug-in and an event-based ArchStudio component that are linked through an inter-process connector.

Consequently, when both environments are running, the two environments can exchange the required information about the architecture and implementation. It should be pointed out that it is not mandatory for both environments to be run together at all times. Modifications to the architecture and the implementation can be carried out independently, and when both environments are brought up together, the synchronization between the two can be completed.

Defining Mappings. Since the two environments are intended for different purposes (architecture design versus implementation), we had to devise a mapping between their elements. At the architectural level in ArchStudio, components and connectors serve as the essential building blocks of the application. In a dynamic architecture, such as the ones ArchStudio supports, these components can be added, removed, and replaced on-the-fly. However, source code deals with elements at a different level of granularity. The functionality of a component can be implemented using one or more classes. The decision of how to split this functionality is a low-level design decision that can be dictated by language restrictions, clarity of design considerations or application of design patterns. Therefore, we considered that a component should accommodate an arbitrary number of classes.

Our solution supports the notion of a component in Eclipse as a *component project*, an extension to a regular Java project. The implementation for a com-

ponent or connector in the architecture will therefore consist of the contents of the component project, and the architectural description will have links to both a source code and a binary version of the implementation. These links will point to a corresponding repository location if the component or connector is not under development, or to a local Eclipse IDE workspace if the component or connector is under development and needs to be tested before being committed. A noteworthy observation is that, in what regards the mapping to source code, we found no difference between components and connectors. The difference between them is a semantic and more important at the architectural level, but we found implementing both as a *component project* being sufficient for time being.

Besides the regular packages and java class files, this project also contains a special file that contains descriptions of component's properties as metadata [13]. Although in this paper we present the means by which to maintain the mapping between architecture and implementation consistent, an interesting problem that we plan to address in the future is how to use this mapping to enforce the semantic consistency between the properties of the implementation, as derived from the source code, and the properties claimed for the component or connector in the architectural description. The role of the metadata is to mediate this problem, and its implementation is based on a simple observation: the architectural model already contains a number of descriptions related to component implementations. In order to enable parallel development of architecture and components, the metadata in the component project will describe the same types of properties that are relevant at the architectural level.

The problem of consistency between architecture and implementation is in this way split in two parts: first, the component metadata and its implementation need to be maintained consistent, then the component metadata and the architectural model need to be synchronized. An example of such a property would be the name of a main class that is needed from the implementation in order to instantiate the component. Within Eclipse, the component metadata editor can check if the name chosen for this property actually denotes an existing class in the implementation, and will help choose a valid one if not. Then, using our integration between Eclipse and ArchStudio, the metadata can be updated into architectural models that contain the component.

This link between Eclipse and ArchStudio is important because it allows implementation for components independently from the architectural design while at the same time makes sure that the properties implemented in the components are the same as the ones described in the architecture.

2.2 Versioning Structure

Since the architectural model evolves alongside its component implementation, the architecture description should itself be versioned. Therefore, a specific version of the architecture consists of an architectural configuration made up of specific component and connector versions. Besides recording the changes to the architectural model, versioning in such a manner allows us to determine for each

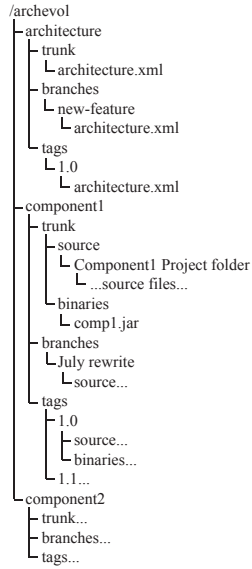

Fig. 2. ArchEvol Subversion Directory Structure.

version of the architecture the corresponding versions required for instantiation of all components and connectors.

ArchEvol intends to manage the architecture and implementation evolution throughout the design and deployment phases of the software life cycle, therefore a consistent versioning approach must be adopted throughout to ease the transition between one phase and another. This transition mandates that all necessary artifacts be consistently and uniformly available. One such technique to achieve this consistency and uniformity is to leverage a WebDAV repository to store the project artifacts.

In order to support both design-time and run-time evolution management for compiled languages (such as Java), both source files and compiled binaries need to be available at all times: our chosen method is making the resources available via a WebDAV interface. An important observation should be made here: relying upon URLs allows the source code for components and the architecture to be distributed throughout a potentially virtual organization by using multiple WebDAV servers. However, the simplistic approach would be to have one centralized WebDAV server that stores all of the necessary artifacts for a project.

Adoption of Subversion. Subversion provides a WebDAV-compliant repository used for storing project artifacts as they evolve in ArchEvol. This is a natural choice as Subversion provides a WebDAV interface as well as an Eclipse plug-in via the Subclipse client [14]. However, the adoption of Subversion does introduce one deficiency from other more traditional SCM systems in one re-

spect: a lack of per-file revision numbers. Instead, it provides a global revision number which includes directory versioning.

Therefore, for example, within Subversion, it is not possible to refer to just revision 20 of a file foo.c. Instead, you must refer to repository revision 20 which contains foo.c. To this end, it is considered standard practice by Subversion developers [15] to use a subdirectory called trunk to store the latest revisions of a project, and then copy the contents of trunk into a subdirectory within another directory called tags when an official version is to be released. Additionally, if development is to be split into multiple tracks, a corresponding copy of trunk can be copied to the branches directory where the branch can be conducted.

This directory structure means that the location within a Subversion repository explicitly indicates the version of the component, whether it is on the trunk, part of an official release tag, or part of a development branch off the mainline. As described in Figure 2, ArchEvol follows a particular versioning structure that adheres to the published Subversion best practices and satisfies the constraints of maintaining design-time and run-time evolution. The particular structure of the repository illustrates a scenario where the architecture and the implementation for the components are maintained in the same directory. However, the directory for the architecture and the directories for the components could be maintained in a distributed fashion over a number of Subversion repositories.

Versioning Components. Each component in an architecture is assigned its own directory within a Subversion repository. This directory contains two types of information: source code, containing the Eclipse project associated with the component, and binaries, where the result of compilation of the Eclipse project should be deposited.

An ArchEvol component can be opened up as a component project in Eclipse with the help of an Subversion client such as Subclipse. However, while the project is open in Eclipse, the developer has a local copy of the project that can be used to test the functionality in an open architecture in ArchStudio that uses this component. Once the changes are declared satisfactory, the project can be checked back in the trunk directory in Subversion, updating the same version, or a new version can be created and the project has to be copied in a new tag or branch directory.

In what concerns the binary packages, we now leave building them as the responsibility of the developer. The binaries have to be put in the specific directories that we prescribe, and the Subversion URL pointing to them will be synchronized between Eclipse and ArchStudio using our communication infrastructure. ArchStudio will need this information in order to instantiate the components.

One advantage of having the source code information separate from the binary packages for components is that we can provide a uniform way of instantiating the components for which the source code is not released and we only have the distribution package. Additionally, by having the release implementation for each component packaged as a jar file this offers the support for architectural deployment without a compiler. An architecture-based deployment tool will know

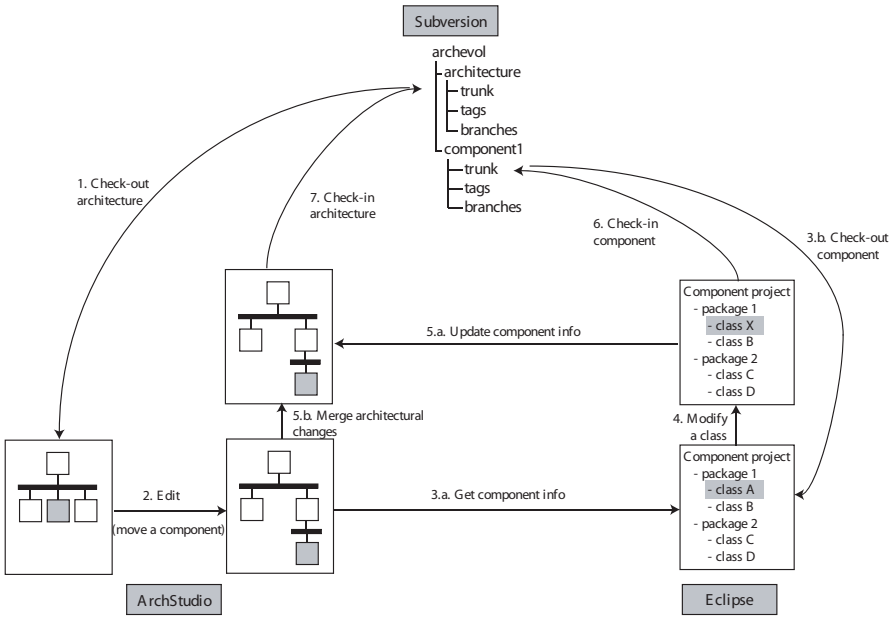

Fig. 3. ArchEvol Editing Process.

where the distribution packages are located, and it can rely upon the appropriate files from the repository.

Versioning Architectures. A version of the architectural model can be opened in ArchStudio for editing or for instantiation by providing a WebDAV URL to ArchStudio. An architectural model in ArchStudio consists of a single xADL2.0 file. The current version of the architecture will be kept in the trunk directory, while versions will be kept in tags and branches directories. It is important to point out that Figure 2 only shows the organization of directories for one architecture, but in reality the Subversion repository can contain any number of architectures for different systems.

An architectural model will contain links to the corresponding binary distribution in the WebDAV repository for each component and connector. These links can be used by ArchStudio's built-in Architecture Evolution Manager (AEM) to instantiate an architecture from the architectural description. When versioning an architecture, the links to the corresponding versions for the components are saved within the file.

3 Example

The scenario depicted in Figure 3 shows how our infrastructure helps us in fulfilling our vision for architectural-driven development and evolution. To date, the critical aspects of our process are automated, however, it is sometimes necessary to manually look up information presented in one of the three applications and then paste it into another. Our scenario notes instances where manual interaction is still necessary.

Figure 3 shows the steps involved in one cycle of development in an architecture based evolution setting. At a certain point in time, there is an architectural model and there are implementations for all the components in the architecture, and they are kept in a Subversion repository in a similar directory structure as we have described in 2.2. We want to follow the interaction of the three tools through a modification of the architecture that requires a modification of one component's implementation as well. In our scenario we move a component to a lower level in the architecture, which requires a change in the component's implementation because in the new position the component will handle additional types of messages produced by the components above it.

Step 1: In the first step, a local copy of the architecture is checked out using any Subversion client.

Step 2: The architect opens and modifies the architectural model in ArchStudio. The modification consists of adding a new connector and moving a component. Currently, ArchEvol does not determine if this type of change necessitates a change to the implementation. However, future analysis based on existing consistency mechanisms, such as incorporated in Rational Rose and other design editors to map UML to code, may be able to detect this automatically.

Step 3a: The component developer opens Eclipse which connects to ArchStudio through the ArchEvol-Eclipse plug-in. The user is presented with a list of components and connectors along with their respective source-code URLs.

Step 3b: The source-code URL is used to load the component's source code as a project into Eclipse. Currently the implementation must be manually checked out into a local folder using Subclipse. Once the implementation is checked out, a new component project is created using the contents of the checked out folder and a locally created copy of the component's architectural metadata contained in the architectural description opened in ArchStudio. It retrieves the component's architectural metadata from ArchStudio automatically. Connectors are handled in a similar way as components.

Step 4: The developer modifies the source code of the component. A consistency critic, provided by the ArchEvol plug-in, informs the developer if the class referred to in the component's architectural metadata is not present in the actual implementation.

Step 5: Once changes to the implementation are complete and consistent with the local copy of the component's architectural metadata, the local copy of the component's architectural metadata is used to replace that in architectural description. We hope to provide the ability to merge this data in the future automatically. It is not always necessary to perform this step. This is only necessary

if a change in the implementation necessitates a change in the architectural description. For example, this might occur when the name of the main class used to instantiate the component is changed. At this point, with both ArchStudio and Eclipse open at the same time, the developer can instantiate the system in order to determine if the changes to the component's source code are effective. This is possible because the URL for the binary version of the component that is being edited is temporarily replaced to point to the location of the locally compiled Java class files created by Eclipse when compiling the component's source code. Other components in the architecture that are not checked out locally can be instantiated using the binary URL provided in the architectural description. This allows a developer to instantiate an entire system without needing to check out every component within it. Steps 4, and 5 when necessary, can be repeated until the desired results are achieved.

Step 6: If the changes are satisfying, the folder for the component project in Eclipse can be committed manually using the Subclipse plug-in.

Step 7: The architectural description can be saved from ArchStudio, and the local copy can be committed using the same Subversion client that was used at step 1. The scenario described here is just one step in the development cycle. The benefits of having the versioning structure in place can also be seen during the release process. When a component's implementation is stable enough for an official release, the content of the trunk folder in the component's repository can be copied to the tags folder. The architecture can be modified so that each component's URL points only to official component releases. Likewise, it can be copied into its own tags folder. The official release architecture can then be instantiated without the need to check-out and compile any source code and without being effected by continuing development of the architecture or components.

4 Related Work

Maintaining the relationships between architectural descriptions and component implementation has been the focus of earlier research. One approach is to enforce consistency by ensuring that an implementation conforms to its architecture and that the architecture correctly represents its implementation. ArchJava [8], proposes specific Java language extensions that incorporate architectural descriptions into the source code. It enforces application communication integrity, meaning that components only communicate along architecturally declared communication channels. However, ArchJava uses implicit mappings, since architecture and implementation are described in the same files. Furthermore, ArchJava does not address any versioning constructs as the mapping between an architecture and its implementation is static.

Projects, such as Software Concordance [16], have treated implementation artifacts as hypertext with links to other project artifacts, such as specifications. The Molhado configuration management system [17], integrated with the Software Concordance environment, aims at providing a generic infrastructure

for maintaining versions of all kinds of software artifacts. In particular, MolhadoArch is very similar to ArchEvol in that it offers versioning capabilities for both the architecture and the individual components in an architecture. Their architecture-implementation mapping shares many of the same characteristics, but is specific to the Molhado environment. The architectural information can be imported from an xADL file, but by doing so it is transformed into Molhado's internal data structures. Other architectural tools such as ArchStudio cannot be used any longer on the architecture. ArchEvol, instead, builds upon the generally available architectural environment such as ArchStudio and adds integration with familiar systems of Eclipse and Subversion. Therefore, the developer experience does not change or require use of different tools.

Focus [18] takes a reverse engineering approach to mapping architecture to implementation. The source code for an application is reverse engineered to an UML class diagram and then related classes are grouped together as components. This approach maintains these mappings through evolution by applying the recovery process incrementally, to the changed portions of the source. The mapping between architecture and implementation proposed by ArchEvol is similar to the one in Focus in that a component is defined as a group of related classes. However, Focus seems more suitable for large applications that have all the source code at the same location, and where changes are first done at the implementation level. ArchEvol instead tries to promote a decoupled development model where architecture and implementation can be developed and evolved in parallel.

Traditional component-based development uses components as binary packages, but reusing them usually requires changes and customizations that can only be performed at source code level [19]. The Source Tree Composition project [20] proposes a component model that uses source code components rather than binary ones, and offers a solution to merging the source code in different components in order to build new systems with various configurations. ArchEvol uses the same definition of components as being composed out of source code elements, however Source Tree Composition goes further into solving different dependencies and building mechanisms between components. ArchEvol is focused more on the development of such components, while deployment and solving dependencies are problems that can be solved in a future extension.

5 Conclusion

Since the architecture descriptions and implementations are kept separate by ArchEvol, it is possible to have parallel development of architecture and components. We explicitly focused on integrating existing best-of-breed off-the-shelf applications to support a coherent decentralized development process. An architect can focus on working with architectural environments and tools while developers can work on implementation with familiar tools.

In the ArchEvol model, components become now not only the basic units for deployment, but also the basic units for decentralized development. Our current progress has demonstrated that ArchEvol can indeed achieve the integration

necessary to provide versioning support for maintaining relationships between architecture and implementation. While the basic functionality is present, there still remains work to increase the automation of the entire process. At present, all functionality described in this paper is possible - however, some aspects still require manual intervention where we believe automation would ultimately be possible.

We also seek to examine the long-term effects of ArchEvol on the overall development cycle and analyze whether our selected mappings between architectural entities and component projects are sufficient to describe real-world projects. Defining consistency rules between architectural description and implementation is still an open question, and involves determining which exact parts from the source code have an influence at the architectural level. With the proper feedback and refinements, we believe that ArchEvol will be able to provide developers with the support necessary to maintain meaningful relationships between architecture and implementation throughout the software life cycle.

6 Acknowledgements

Effort funded by the National Science Foundation under grant numbers CCR-0093489 and IIS-0205724.

References

1. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes **17** (1992) 40–52
2. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26** (2000) 70–93
3. Estublier, J., Casallas, R.: The Adele configuration manager. In Tichy, W., ed.: Configuration Management. John Wiley and Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England (1994) 99–133
4. Lampson, B.W., Schmidt, E.E.: Organizing software in a distributed environment. In: SIGPLAN '83: Proceedings of the 1983 ACM SIGPLAN symposium on Programming language issues in software systems, New York, NY, USA, ACM Press (1983) 1–13
5. GNU: CVS. http://www.gnu.org/software/cvs/ (2005)
6. Collabnet: Subversion. http://subversion.tigris.org (2005)
7. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: An infrastructure for the rapid development of xml-based architecture description languages. In: 24th International Conference on Software Engineering (ICSE 2002), ACM (2002)
8. Aldrich, J., Chambers, C., Notkin, D.: Archjava: Connecting software architecture to implementation. In: Proceedings of the 24th International Conference on Software Engineering. (2002)
9. Roshandel, R., van der Hoek, A., Mikic-Rakic, M., Medvidovic, N.: Mae—a system model and environment for managing architectural evolution. ACM Trans. Softw. Eng. Methodol. **13** (2004) 240–276

10. Institute for Software Research: ArchStudio 3. http://www.isr.uci.edu/projects/archstudio/ (2005)
11. Eclipse Foundation: Eclipse. http://www.eclipse.org (2005)
12. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: A highly-extensible, xml-based architecture description language. In: Working IEEE/IFIP Conference on Software Architecture (WICSA 2001). (2001)
13. Orso, A., Harrold, M.J., Rosenblum, D.S.: Component metadata for software engineering tasks. In: Second International Workshop on Engineering Distributed Objects (EDO 2000), Springer Verlag: Berlin (2000) 126–140
14. Collabnet: Subclipse. http://subclipse.tigris.org (2005)
15. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, M.: Version Control with Subversion. http://svnbook.red-bean.com/ (2004)
16. Nguyen, T.N., Munson, E.V. In: The software concordance: a new software document management environment. ACM Press (2003) 198–205
17. Nguyen, T.N., Munson, E.V., Boyland, J.T., Thao, C. In: Molhado: Object-Oriented Architectural Software Configuration Management. IEEE Computer Society (2004) 510
18. Medvidovic, N., Jakobac, V.: Using software evolution to focus architectural recovery. Journal of Automated Software Engineering (2005)
19. Garlan, D., Allen, R., Ockerbloom, J. In: Architectural mismatch or why it's hard to build systems out of existing parts. ACM Press (1995) 179–185
20. de Jonge, M.: Source tree composition. In: Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools, Springer-Verlag (2002) 17–32