Versioned Software Architecture

André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA
{andre,dennis,alw}@cs.colorado.edu

Abstract

In this position paper we introduce a novel use of software architecture. Rather than following the traditional focus on design, we propose to use the notion of *versioned software architecture* to support other activities in the software life cycle. In particular, we are investigating how the activities of configuration management and software deployment can benefit from the availability of an explicit architectural representation that is enhanced with versioning capabilities. Below, we present some of the initial results of this investigation. We motivate our research into versioned software architecture, present some usage scenarios in the context of configuration management and software deployment, and conclude with an outlook at the future work that remains to be done.

1 Introduction

The ability to accurately describe the high-level design of a software system is at the heart of the discipline of software architecture. Much research has gone into developing architecture description languages (ADLs) in which a design can be precisely captured.[1] Although a large number of ADLs have been created, the set of features commonly found in these languages is fairly stable, and it can be expected that the modeling capabilities of the various languages will converge sometime in the future.

Additional research has been concerned with the verification of particular properties of an architecture once it has been created. Methods exist that, for example, can verify whether an architecture is free of deadlock [1] or whether an architecture eventually reaches a certain desired state [2]. Other ADLs lend themselves to the detection of inconsistencies among components that have been put together in an architecture. Architectural mismatches, such as competing threads of control, have been uncovered this way [3].

Our research takes software architecture in a rather different direction. As opposed to focusing on design, we intend to leverage existing ADLs and their analysis methods to support other activities in the software life cycle. In particular, we are developing an abstraction, *versioned software architecture*, to support the activities of configuration management and software deployment. This abstraction is similar to software architecture in that it allows the modeling of components, interconnections, behaviors, and constraints. Additionally, though, it models the existence of multiple versions of these elements and, indeed, of architectures themselves.

Figure 1 illustrates the role that the abstraction plays in supporting the activities of configuration management and software deployment. Both activities still manage the artifacts that they create, but do so in terms of versioned software architecture. This raises the level of abstraction with which the configuration management and software deployment activities are carried out, reduces overall modeling effort throughout the life of a system, and reduces the context switching that typically occurs between these two activities.

In the remainder of this position paper we first motivate why we have chosen versioned software architecture as our organizing abstraction. We then present some example scenarios that we intend to support, and conclude with an outlook at the future work that remains to be done.

2 Motivation

Versioned software architecture combines the concepts of software architecture and versioning. We have chosen to use this combination as our abstraction for a number of reasons. Let us first consider the reasons for choosing software architecture as part of the abstraction.

- *Architecture adds structure to process.* Both the activities of configuration management and software deployment require knowledge about the structure of the system upon which they operate. Consider, for example, the ability to identify the components of a system that need to be redeployed when the system has been modified, or the ability to version components as opposed to individual source files. To be able to carry out these activities in terms of architectural structure matches the actual software process much closer than the typical low-level support that is embodied in most existing tools.

- *Software architecture description languages model other information about a system besides its structure.* This information can be exploited for configuration management and software deployment purposes.

[1] For a survey of existing ADLs, see [6].

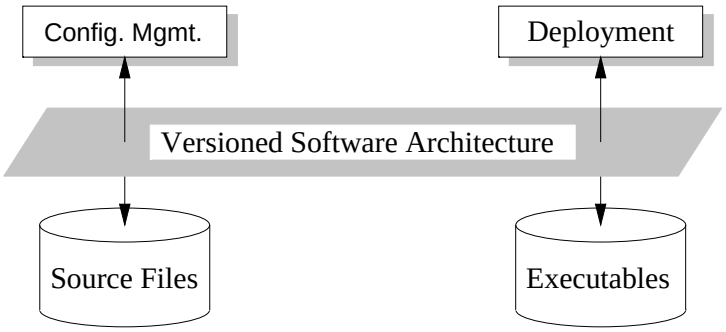

Figure 1: Versioned Software Architecture.

In particular, component behavior, component constraints, and architectural constraints can be leveraged to ensure the consistency of a particular system configuration. Especially when a system configuration is selected out of a set of versions of components, or when components need to be dynamically replaced in a continuously running system, the ability to verify architectural consistency of the system before the actual, potentially damaging action is taken, is important.

- *Architectural information is incorporated in new languages that are invented to support configuration management and software deployment.* Most notably, PCL [7], a language for system modeling that originated in the configuration management discipline, and the Software Dock [4], a generic framework that supports software deployment, have adopted constructs to model the structure of a system. These constructs are similar to the ones commonly found in ADLs. This mere fact suggests that a common, architectural abstraction can be used as a basis for supporting these activities.

These reasons illustrate that software architecture by itself would significantly enhance the activities of configuration management and software deployment. However, the addition of versioning to software architecture has the potential to advance these activities even further. Consider the following three arguments.

- *Architectures evolve.* As a system evolves, so does its architecture. Even though the architecture is meant to be relatively stable, this cannot always be guaranteed. In particular, if the architecture not only models the top-level design of a system, but also incorporates lower-level design choices, it can be expected to change. These changes need to be managed by tracking and managing multiple versions of an architecture.

- *A single architecture can exist in multiple variants.* Different hardware platforms, the availability of different subsystems, and the existence of optional components all may require different solutions for a single system. These solutions all have to be captured in the software architecture. Consequently, the variability of this architecture has to be managed.

- *The activities of configuration management and software deployment can benefit from an understanding of the difference among multiple versions of an architecture.* Consider a deployment system that only deploys and updates those components of a system that have changed, or a configuration management system that, based on the difference between two revisions of an architecture, only recompiles the minimum number of components. A versioned software architecture contains the information that is needed to support such activities.

It is for these reasons that we have chosen our abstraction to be the combination of software architecture and versioning. We are currently working on an implementation of the abstraction, which is based on Darwin [5] and its ability to specify constraints and behaviors as labeled transition systems [2].

3 Usage Scenarios

Although we have not yet constructed a configuration management or software deployment system that is based on the concept of versioned software architecture, we already have identified some of the desired and novel capabilities that are facilitated by the use of versioned software architecture in these domains. Below, we present a number of scenarios that highlight these capabilities.

Component-based workspace management. A typical use of a configuration management system is one in which the set of source files that constitute a system is broken down into smaller groups that, in essence, represent system components. Often, this is a manual process for which little support is provided. Consequently, the initial partition tends to become out of date as the architecture of a system changes. An explicit architecture-based configuration management system cannot only automatically create an initial set of views that represent each component in the system, but also update the views when the system architecture changes by repartitioning the source files.

Architecture-aware implementation verification. Not only does an architecture-based configuration management system allow the progression of changes from an architecture to the system that it manages, the reverse can also be achieved. In particular, when components are checked in after changes have been made, the configuration management system can verify whether the changes that have been made to the component represent architectural changes. Specifically, if new

connections are created, a developer can be notified that these changes are architectural changes that tend to have a rather significant impact. Even if the developer accepts the changes, the actual architecture of the system can be updated. Architectural erosion can be reduced in this way.

Minimal recompilation. Architectural connections can also be used to improve the build process. In particular, through analysis of component behaviors and connections it is possible to determine which source files need to be recompiled. Consider, for example, two components that each use some of the services provided by the other. Any change in the interface of one of the components usually leads to the recompilation of the source files of both. However, if behavioral analysis determines that the second component is not influenced by the change to the interface of the first one, the source files of the second component do not have to be recompiled and an optimization in recompilation effort is achieved.

Consistency verification of a selection. Another kind of optimization can be achieved even before the build process takes place. Typically, a selection of particular versions of components is chosen as a system configuration to be constructed. Using architectural constraints and behaviors, the consistency of the selection can be verified. This is important, because the build process does not necessarily fail if an inconsistent configuration is selected. Even careful testing might not reveal the existence of a problem, and an erroneous system could be delivered to a customer.

Consistency verification of component updates. The consistency of a system needs to be verified in another setting as well. When components of a system out in the field are updated with new versions, architectural behaviors and constraints can be used to verify whether the new component versions are compatible with the existing set of components. Once again, this ability is important because it prevents a number of faults that could result from the insertion of an incompatible component in a continuously executing critical system.

Minimal component update. Typically, the update of a system to a newer version requires a complete removal of the old version followed by a complete install of the new version. A deployment system that operates in terms of versioned software architecture is capable of calculating, in terms of components and connections, the difference between the old and new system versions. This difference can be used to only transfer and reinstall the components that are really needed. In essence, binary, component-based patches can be constructed and deployed.

Coordinated distributed deployment. A rather complicated problem is the coordinated deployment of a system to a distributed set of sites. Consider, for example, an insurance company that deploys new client software to its agents that has to be coordinated with an update of its server software. Typically, custom scripts have to be created to support such cases. A software architecture contains some of the information that a deployment system can leverage in the creation of standardized scripts to generically solve this problem. Of course, these scripts have to be tailored to specific cases, but at least a generic basis for a solution can be created.

4 Conclusions

In this position paper, we have laid out our plans to develop and use the abstraction of versioned software architecture. Although we believe the abstraction is useful in supporting other activities as well, our initial goal is to investigate how the activities of configuration management and software deployment can be enhanced. To this extent, we are planning on creating example configuration management and software deployment systems that demonstrate the scenarios we have presented.

Obviously, our solution heavily relies on the existence of a mapping, not only among the components in an architecture and the source files that implement the architecture, but also among the components and the executables that eventually will contain them. Besides the development of a representation for versioned software architecture, creating a solution to this mapping problem will be one of the first activities that our research will address.

Acknowledgements

This work was supported in part by the Air Force Material Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Numbers F30602-94-C-0253 and F30602-98-2-0163. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[2] S.C. Cheung, D Giannakopoulou, and J. Kramer. Verification of Liveness Properties Using Compositional Reachability Analysis. In *Proceedings of the Sixth European Software Engineering Conference*, number 1301 in Lecture Notes in Computer Science, pages 227–243, New York, New York, September 1997. Springer-Verlag.

[3] D. Compare, P. Inverardi, and A.L. Wolf. Uncovering Architectural Mismatch in Dynamic Behavior. *Science of Computer Programming*, 1999. To appear.

[4] R.S. Hall, D.M. Heimbigner, A. van der Hoek, and A.L. Wolf. An Architecture for Post-Development Configuration Management in a Wide-Area Network. In *Proceedings of the 1997 International Conference on Distributed Computing Systems*, pages 269–278. IEEE Computer Society, May 1997.

[5] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the Fifth European Software Engineering Conference*, number 989 in Lecture Notes in Computer Science, pages 137–153, New York, New York, September 1995. Springer-Verlag.

[6] N. Medvidovic and R.N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference*, number 1301 in Lecture Notes in Computer Science, pages 60–76, New York, New York, September 1997. Springer-Verlag.

[7] E. Tryggeseth, B. Gulla, and R. Conradi. Modelling Systems with Variability using the PROTEUS Configuration Language. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, number 1005 in Lecture Notes in Computer Science, pages 216–240, New York, New York, 1995. Springer-Verlag.