Semi-Automated Incremental Synchronization between Conceptual and Implementation Level Architectures

Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley Schmerl and Nagi Nahas
Institute for Software Research International, Carnegie Mellon University, Pittsburgh, PA 15213
{mabianto+, aldrich+, garlan+, schmerl+}@cs.cmu.edu nnahas@acm.org

Abstract

In practice, there are many differences between an implementation-level architecture (such as one derived using architectural recovery techniques) and a more conceptual architecture used at design time.

We present a lightweight, scalable, semi-automated, incremental approach for synchronizing a Component-and-Connector (C&C) view retrieved from an implementation with a conceptual C&C view described in an Architectural Description Language. Our approach can automatically detect corresponding elements in the presence of insertions, deletions, renames, and moves, and incrementally synchronize the two views.

1. Introduction

This paper describes a lightweight and scalable approach to synchronize an implementation-level architectural view, such as one reconstructed using architectural recovery techniques, with a conceptual-level architectural view expressed in an Architecture Description Language (ADL). Our approach handles expressiveness gaps between conceptual and implementation-level architectural views, allowing architects to keep the two architectures up-to-date without losing architectural style, type and property information needed for architectural-level analyses.

Synchronization matches elements in the presence of insertions, deletions, renames and moves, and proposes a set of edits to make one representation more consistent with the other. While our approach is potentially applicable to a wide range of ADLs, in order to evaluate it, we have chosen the Acme ADL to represent the conceptual-level architecture in, and ArchJava [ACN02] for the implementation-level architecture.

2. The Design-Implementation Gap

There are several problems that must be addressed when attempting to synchronize any pair of design-oriented and implementation-oriented architectures.

In Acme, as in many other design languages, types are arbitrary logical predicates. Such a type system is highly desirable at design time, because it allows designers to combine type specifications in rich and flexible ways. Unfortunately, examples like this cannot be expressed in implementation-level type systems such as the ones provided by ArchJava. Since a design-level predicate-based type system is fundamentally incompatible with a programming-language-style type system, any system synchronizing between design- and implementation-level views has to allow the user to specify arbitrary matches between the two type hierarchies in the two systems.

Design languages such as Acme tend to treat hierarchy as design-time composition, where a component at one level in the hierarchy is just a transparent view of a more detailed decomposition specified by the representation of that component. Multiple representations for a given component or connector could correspond to alternative decompositions into sub-systems. On the other hand, implementation-level C&C views such as ArchJava tend to view hierarchy as the integration of existing components, along with glue code, into a higher level component; due to the glue, a higher-level component is semantically more than the sum of its parts. These differing meanings of hierarchy create additional challenges for synchronizing the two views. For example, if multiple representations are present at the design level, there must be a way to specify which of these representations was actually implemented.

As another example, components in ArchJava can have internal ports that are used for communication between a component and its subcomponents. These ports cannot be directly represented in Acme, forcing us to model a private port as a port on an internal component instance with properties specifying its visibility. As a final example, Acme views an external port of a composite component as just an alias for ports in its subcomponents: it only allows binding an outer port to one or more inner ports; in contrast, ArchJava does not distinguish between bindings and attachments and con-

nectors can connect an external port to ports on subcomponents.

There are additional differences between Acme and ArchJava that are more incidental in nature, but nonetheless make the problem of relating the two representations more challenging. While these differences will vary according to languages, they are suggestive of the challenges likely to be encountered by anyone trying to synchronize two C&C views. To mention only a few of these differences: unlike Acme, ArchJava does not declare explicit types for ports and does not have first-class named and typed connector roles.

3. Structural Differences

Any synchronization approach must be able to handle elements that are inserted and deleted between to views, as supported by the ArchDiff tool [WH02]. Synchronization between design-level and implementation-level architectures, however, requires going beyond insertions and deletions to support renames and moves. Name differences between the two representations can arise for a variety of reasons. ArchJava does not even name certain elements (e.g., connectors, roles and attachments): any names they may have in Acme are lost during code generation. Identifying an element as being deleted and then inserted when it fact it is renamed, would result in losing crucial style and property information about the element at the design level.

Furthermore, it is not unusual for architects and implementers to differ in their use of hierarchy, so that components expressed at the top level in one architecture are nested within another component in some other architecture (i.e., in Acme, this would correspond to replacing an architectural element with its representation). For example, the architect may want to use hierarchy to analyze the architecture at a higher level or hide certain decision decisions from some parts of the system, but an implementer may wish to flatten the hierarchy for efficiency reasons. This requires detecting moves across levels of hierarchy.

4. C&C View Synchronization

Our approach to enforcing structural conformance between an architectural C&C view and an implementation-level C&C view proceeds as follows: 1) convert the architectural C&C view into tree-structured data; 2) retrieve a C&C view from the ArchJava implementation and convert it to tree-structured data; 3) use a tree-to-tree correction algorithm for unordered labeled trees to identify matches and structural differences (classified as inserts, deletes, renames and moves), and obtain an edit script to make one view more consistent with the other; 4) supplement the edit script with information that cannot be derived from the architecture or the implementation (for example, styles and types, in one direction, or namespaces and source code locations in the other); and 5) optionally apply the edit script to the underlying representation (e.g., the Acme model or the ArchJava implementation).

A C&C view has no inherent ordering among its elements, and calls for unordered tree-to-tree correction, an NP-Complete problem. However, a recently published algorithm [THP05] provides an exact polynomial-time solution under additional assumptions. We generalized this algorithm into one that can also detect moves and force and prevent matches between view elements, to improve speed and accuracy.

5. Validation

To validate our approach, we have implemented a tool based on the algorithm that can make an Acme model incrementally consistent with an ArchJava implementation. We still need to change the ArchJava infrastructure to support making incremental changes to an existing ArchJava implementation. We have conducted several successful case studies with the tool.

In one case study, we use an ArchJava implementation of a pedagogical circuit layout application [ACN02] with over 20 components divided into several subsystems. The tool found several interesting divergences between the C&C view obtained from the implementation and a C&C view postulated by the architect, while offering interactive performance.

6. Future Work

There are various limitations of the approach and tool that we will address as future work. We would like to explore other comparison algorithms to determine which gives the best performance and the best results, as well as support additional differences such as merges and splits. We still need to change the ArchJava infrastructure to support making incremental changes.

7. References

[ACN02] Aldrich, J., Chambers, C. and Notkin, D. ArchJava: Connecting Software Architecture to Implementation. In Proc. ICSE, 2002.

[GMW00] Garlan, D., Monroe, R., and Wile, D. Acme: Architectural Description of Component-Based Systems. In Foundations of Component-Based Systems, Cambridge University Press, 2000.

[THP05] Torsello, A., Hidovic-Rowe, D. and Pelillo, M. Polynomial-Time Metrics for Attributed Trees. IEEE Trans. on Pat. Analysis and Machine Intel., 27 (7), 2005.

[WH02] van der Westhuizen, C. and van der Hoek, A. Understanding and Propagating Architectural Changes. In Proc. WICSA 3, 2002.