Submitted for publication.

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. Furthermore, additional differences may be introduced during software development and evolution due to documentation or implementation defects. This makes ensuring conformance between an architectural design and code a difficult problem worth addressing.

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 tool can automatically detect corresponding elements in the presence of insertions, deletions, renames, and moves, and incrementally synchronize the two views. When we applied the approach on an architecture of over 20 components, we found several divergences between the conceptual and implementation level architectures.

1. Introduction

Ensuring that a system as built conforms to its architectural design during software development and evolution is important, as significant divergences between architecture and implementation can compromise architectural structure, style and properties that have been established with careful analysis at the architectural level.

Previous work has taken a number of approaches to ensuring conformance between architecture and implementation. Generating skeletal or glue code from architecture is one option [SDK+95], although it provides no guarantees that the code will remain conformant to the architecture as either evolves. Approaches based on architectural reconstruction include analysis-based architectural extraction [MNS01][EOG+98], run-time architecture monitoring [YGS+04], and using a type system to verify conformance to an architecture expressed within a programming language [e.g., ACN02].

A weakness of existing approaches is that they reconstruct architectures that are very concrete and implementation-oriented — in contrast, software architects often view architecture more abstractly, omitting components and connections that are not relevant to their particular concern. Also, automatically reconstructed architectures often lack the styles and properties on which an architect relies when designing a system. Finally, approaches assuming the primacy of one architectural view over the other, i.e., allowing changes to be made in only one direction, are overly limiting, as it is valid to make changes in either view. Thus existing work limits the ability of architects to work at an appropriate level of abstraction and simultaneously ensure that the design is a faithful abstraction of the implementation, by not having the ability to synchronize the two views.

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 to represent a conceptual-level architecture in the Acme ADL. Implementation-level C&C views can be also extracted from implementation-constraining ADLs with code generation capabilities or implementation independent

ADLs such as C2 [MT00] that provide an implementation framework for code generation. We have chosen ArchJava [ACN02] for our approach.

We begin by describing the differences between architectural information at different levels of abstraction that any synchronization approach must address. Section 3 describes ensuring conformance based on matching tree-structured architectural information. Section 4 describes the prototype tool we have built. Section 5 shows how the approach was used to detect several divergences between the conceptual and implementation level architectures of an architecture of over 20 components. Finally, we discuss related work and conclude with future work to address some of the limitations of our approach.

2. The Design-Implementation Gap

In this section, we describe the problems that must be addressed when attempting to synchronize an implementation-level architecture with a conceptual-level architecture. Typical Architecture Description Languages (ADLs) model architecture as a set of components, connections between them, and constraints on how these components interact [MT00]. We use Acme and ArchJava to illustrate the differences in expressiveness, but many problems are common to any pair of design-oriented and implementation-oriented architectures, and the corresponding Component-and-Connector (C&C) views.

2.1 Expressiveness Gaps

In many design languages, types are arbitrary logical predicates: an element is an instance of any type whose properties and rules it satisfies, and one type is a subtype of another if the predicate of the first type implies the predicate of the second type. Such a type system is highly desirable at design time, because it allows designers to combine type specifications in rich and flexible ways. Acme embodies this approach, but it is hardly unique; for example, PVS [ROW98] takes a similar approach. As an example of the benefits of a predicate-based type system, consider an architecture that is a hybrid of the pipe-and-filter and repository architectural styles [SG96]. In this example, a filter component type has at least one input and one output port, while a client component in the repository style has at least one port to communicate with the repository. A component in this architecture might inherit specification information from both the filter and the repository client specifications, yielding a component that has at least three ports: two for communicating with other filters and one for communicating with the repository.

Unfortunately, examples like this cannot be expressed in implementation-level type systems such as the ones provided by C2SADL [MOR+96] or ArchJava. A specification that a component has a port implies a requirement that the environment will match that port up with some other component, and therefore conventional type systems require a component type to list all of the ports it might possibly have (or at least all those ports that are expected to be connected at run time). There is no way to say that a filter component has "at least two ports"—instead, one must say that the filter has "at most" or "exactly" two ports. Therefore, in the implementation, one cannot combine the filter type with a repository component type (which defines a third port that is prohibited by the filter specification). 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. In our current tool prototype, the architect specifies this mapping in a view that shows the type hierarchies in both systems flattened and shown side-by-side (See Figure 1).

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

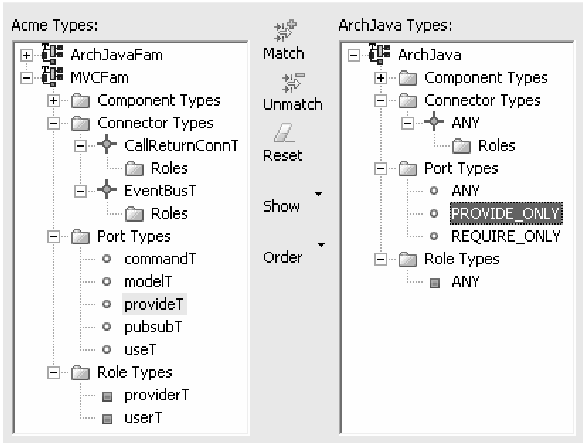

Figure 1: Matching Types Structures: the user assigns any ArchJava port with only provided methods the *provideT* Acme type defined in the *MVCFam*, a Model-View-Controller style.

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 both ArchJava and UML-RT [CG01] 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 connectors can connect an external port to ports on subcomponents.

2.2 Incidental Differences

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:

- **Missing Port Types:** ArchJava does not declare explicit types for ports. This means that Acme port types must be assigned for each ArchJava port instance, rather than assigning an Acme type to each ArchJava port type.
- **Missing Instance Names:** Unlike Acme, ArchJava does not name connectors or roles. This requires us to match connectors and roles based on structural criteria rather than names.
- **Missing Connector Roles:** ArchJava does not have first-class roles and role types, unlike Acme, one of the few ADLs with explicit support for roles. Our tool takes advantage of style constraints in order to automatically infer the types of the implicit roles when going from ArchJava to Acme.
- **Top Level Elements:** The top-level structure of architecture is represented as a *system* construct in Acme which differs from a component in that it cannot declare external ports. In ArchJava, top-level architectures are represented by components which may have ports, forcing us to model an ArchJava top-level component that declares ports with an extra level of hierarchy in Acme.

2.3 Structural Differences

In addition to the gaps in language expressiveness, the ways conceptual-level and implementation-level C&C views are developed and evolved differently creates challenges for architectural synchronization. Some of these differences are by design; others are due to implementation or documentation defects.

For instance, architects may choose to structure a system in different ways than system designers: an architect may choose to abstract away some of the components and connections in a system, because they are not relevant to her modeling task. An implementation-oriented view, on the other hand, is likely to be complete. This example suggests that any synchronization approach must be able to handle elements that are inserted and deleted between the design-level and the implementation level, 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. Similarly, the architect may update a name in one representation and forget to update it in the other representation. 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 [Par72], but an implementer may wish to flatten the hierarchy for efficiency reasons. This requires detecting moves across levels of hierarchy.

These are not the only structural differences that may arise: elements (e.g., components or ports) can be split or merged during restructuring of the architecture or the implementation. Splitting is common practice, but is difficult to formalize, since it affects connections in a context-dependent way [Erd98]. We leave splits and merges to future work.

3. Ensuring Conformance

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 – See Figure 2), 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). The final step is optional because the architect may consider the differences innocuous or may only be interested in a change impact analysis [KPS+99].

3.1 Tree Structured Data

The architectural structure information is represented as a cross-linked tree structure instead of a graph, to emphasize the notion of hierarchy inherently present in nested sub-architectures and to keep the algorithms tractable. For scalability reasons, and taking advantage of the recursive nature of the problem, our structural comparison is designed to start at a given Acme system (or a given Acme component representation) and a corresponding ArchJava top-level component. If there are multiple representations for a component in Acme, the user can use this feature to determine which representation was actually implemented by the developers, by finding the one that most closely matches the implementation.

The tree structure closely mirrors the hierarchical decomposition of the system and includes information to improve the accuracy of the structural comparison. For instance, the subtree of a node corresponding to a port or role includes all the port's or the role's involvements, i.e., all components (and their ports) or connectors (and their roles) reachable from that port or role through attachments or bindings. Cross-links refer back to the defining occurrence of each element and allow the user to navigate the architectural graph.

We decorate each element in the tree-structured data with various properties, some automatically retrieved from either representation, others corresponding to user-entered data (e.g., type assignments). These properties are not represented in the tree-structure and are not directly considered during tree-to-tree correction (i.e., they will not have edit actions associated with them). However, these properties provide additional semantic information that the matching algorithms can rely on. For instance, the type information, if provided, is used to build a matrix of incompatible elements that may not be matched. In addition, some of these properties provide a loose mapping between the C&C view elements and the corresponding elements in the module view; e.g., properties with ArchJava namespace and source code location information are automatically added to elements retrieved from ArchJava to help distinguish between similarly named types in the implementation.

Obtaining the architectural tree-structured data is simply a matter of converting the Acme architectural graph into the cross-linked tree structure. Most of the elements are already available as Acme model elements; required and provided methods are retrieved from properties on ports if they exist.

The tree-structured data is derived from the implementation by traversing the ArchJava compilation units, ignoring non-architecturally relevant Java classes or fields (i.e., not of type component or connector), representing the various elements and bridging expressiveness gaps as they are encountered, e.g., representing a private port by first creating an internal component instance and adding the port to it.

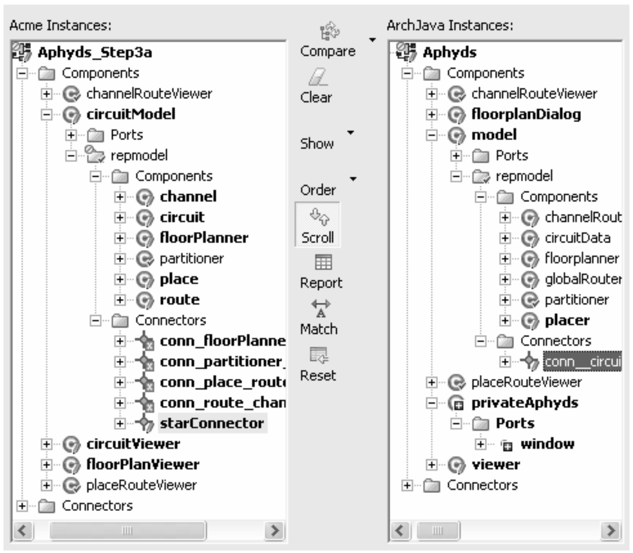

Figure 2: Structural comparison of architectural instances in a C&C view retrieved from Acme and a C&C view retrieved from ArchJava: component *privateAphyds* exists in ArchJava but not in Acme; similarly, connector *starConnector* matches a connector in ArchJava with an automatically-generated name (highlighted nodes).
Symbols: Match (✓), Insert (⊞), Delete (⊠), Rename (✎)

3.2 Tree-to-Tree Correction

Tree-to-tree correction is then used to compare the tree-structured data from the architecture and from the implementation views to find structural differences and produce an edit script. The comparison can be restricted to user-defined subsets of the two views: e.g., if the Acme model does not specify some information that exists in ArchJava (such as method signatures), this information can be excluded from the comparison to avoid gratuitous differences. Structural comparison finds matches and classifies differences as inserts, deletes, renames, and optionally moves.

Much of the research on tree-to-tree correction has focused on ordered labeled trees, since the problem for unordered trees is MAX SNP-hard [ZJ94]. We initially used an exact polynomial time tree-to-tree correction algorithm [SZ97], simply ordering nodes by name. We also added string-to-string correction to evaluate the intrinsic degree of similarity between the labels of two nodes, using the standard dynamic programming algorithm to find the longest common subsequence [WF74]. Given that the ordering we chose is artificial, however, it is perhaps unsurprising that we found this algorithm to perform poorly when renames change the ordering of sibling nodes in the tree. Other orderings are possible (such as ordering subtrees by weight) but these have similar drawbacks.

A software architecture has no inherent ordering among its elements, suggesting that an unordered tree-to-tree correction algorithm might perform better. Existing algorithms work around the NP-hardness result either through heuristic methods [WDC03] or through an exact solution under additional assumptions [THP05]. We chose the second approach, initially assuming that a node can be matched to another only if its parent is matched to some node. However, this assumption did not allow us to detect moves, such as when a top-level architectural component is moved into the representation of another component.

We generalized this assumption so that the algorithm can correctly identify architectural moves as long as they do not involve more than M levels of architectural hierarchy (for some fixed constant M). Our resulting algorithm is polynomial-time, yet generalizes the one in [THP05] to detect renames, inserts, deletes and moves, as well as support forcing and preventing matches between nodes. Unfortunately, space limitations prevent us from a more detailed description of our algorithm.

An upper bound on the running time of the algorithm is as follows: let X the set of nodes of both trees, x an element of X, p the maximum allowable size of a connected subgraph of the tree that can be deleted or inserted in the middle of the tree, $f(x,p)$ the number of nodes that lie within a distance of $(p+1)$ from x, and $F(a) = max\{f(x,p)\text{: } x \in X \text{ and } p=a\}$: our algorithm has a worst case running time of $O((2F(p))!\ N^2)$ and requires $O(d\ N^2)$ memory, where d is an upper bound on the maximum degree of a node and N is number of nodes. In our implementation, pruning the search tree, using additional semantic information (e.g., types) and being able to limit the running time by returning a suboptimal solution, make the average case much faster than the worst case. In comparison, [THP05] has a running time of $O(d^3\ N^2)$ and requires $O(N^2)$ memory.

Originally designed to detect moves, our algorithm also detects insertions better than [THP05]; e.g., the latter did not detect that component *privateAphyds* (In Figure 2) is an insertion. We avoided premature optimization in our implementation to allow for easier debugging. On an Intel Pentium4® CPU 3GHz with 1GB of RAM, comparing an Acme tree of around 800 nodes with an ArchJava tree of around 1,400 nodes (as in Figure 2) currently takes around 2 minutes, whereas our implementation of [THP05] takes around 30 seconds but produces less accurate results.

There is one caveat to representing architectural graphs using trees: tree-structuring the data causes each shared node in the architectural graph to appear several times in several subtrees, with cross-links referring back to their defining occurrences. These redundant nodes greatly improve the accuracy of the tree-to-tree correction; however, they may be inconsistently matched with respect to their defining occurrences (either in what they refer to, or in the associated edit operations). We currently alert the user to inconsistent matches in the output, if they occur, and allow her to manually correct them; if provided, the corrections are taken into account to build the edit script.

Inconsistent matches becomes more pronounced if there are many cycles in the architectural graph. But, in practice, we found it is possible to address this problem by making two passes to synchronize the two representations: during the first pass, synchronize the strictly hierarchical information (e.g., components, connectors, ports, roles, and representations); during the second pass, synchronize the context-dependent attachments and bindings. We are also investigating techniques to express the dependencies between the mapping decisions and prevent inconsistent matches.

4. Tool Support

We intended for our approach to be lightweight enough that it can fit into a single dialog with a look-and-feel similar to the one provided by popular open-source Integrated Development Environments [e.g., Ecl03] instead of more specialized environments for architectural recovery such as [TMR02]. Tool support for our approach uses AcmeStudio [SG04], a domain-neutral architecture modeling environment for Acme, and ArchJava's development environment, both implemented as plugins in the Eclipse tool integration platform. At any time while using the AcmeStudio or the ArchJava development environments, the user can invoke the synchronization functionality. We have completed the functionality needed to 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. In both cases, the following five-step process applies:

1. Setup the synchronization
2. View and match types (optional)
3. View and match instances
4. View and modify the edit script (optional)
5. Confirm and apply the edit script

Because steps 1 and 5 are straightforward, we will only discuss steps 2-4 in more detail below.

4.1 Viewing and Matching Types

As we discussed earlier, matching type structures helps discover implementation-level violations of architectural styles and types that are not currently represented in ArchJava. This step has to be done mostly manually (See Figure 1), and for that reason, is currently kept as optional step in the tool. Matching type structures can take several forms:

- Match explicit types when possible: e.g., match an ArchJava component type with one or more Acme component types;
- Assign types to instances when no explicit type is available: e.g., assign types to individual ports on an ArchJava component type;
- Assign types to special wildcards: e.g., using the ArchJava connector type *ANY*, one can assign the Acme type *CallReturnT* to all ArchJava implicit connector instances; similarly, one can assign a specific Acme type to a port with only required and no provided methods (e.g., *useT*) or with only provided and no required methods (e.g., *provideT*);
- Finally, infer types when possible: e.g., infer the types of implicit ArchJava roles based on Acme connection patterns optionally defined for an architectural style: if the architect assigns types to components, ports and connectors, the role type (e.g., *providerT*) is inferred based on the source component type (e.g., *ANY*), source port type (e.g., *provideT*), and connector type (e.g., *ANY*)**.**

4.2 Viewing and Matching Instances

The differences found during structural matching are shown in each tree by overlaying icons on the affected elements (see Figure 2). If an element is renamed, the tool automatically selects and highlights the matching element in the other tree; for inserted or deleted elements, the tool automatically selects the insertion point by navigating up the tree until it reaches a matched ancestor. Various features give the user more control of the structural matching:

- **Direct manipulation:** the user can manually insert, delete or rename elements (e.g., add a port to a component) which will also generate the corresponding edit actions.
- **Elision:** the user can selectively hide (and unhide) elements, excluding them from comparison. Elision can be instance-based or type-based, where all elements of a given type are excluded at once (e.g., only match components and ports). Elision is temporary and does not generate any edit actions.
- **Forced matches:** the user can manually force a match between an Acme element and an ArchJava element without leaving the synchronization tool to change either representation. The user can use this feature to correct inconsistent matches in the output of the tree-to-tree correction as discussed earlier. The user can also use this feature to manually force a match between two elements that cannot be structurally matched : e.g., force a match between an Acme *spliFilter* component designed with one input and two output ports with an ArchJava *split* component implemented with one input port and one output port.
- **Manual overrides:** finally, the user can override any edit action produced by tree-to-tree correction, e.g., cancel a delete action.

4.3 Viewing and Modifying the Edit Script

We produce a common supertree to show the merged model as a tree-structured preview of the architectural model after the edit actions are applied. In this step, the user can assign types to elements to be created, change the types of existing elements, override automatically inferred types, or cancel any unwanted edit actions prior to the application of the edit script.

Setting types during synchronization may affect the processing of the edit script. For instance, when a component instance is assigned a type, it may inherit ports from its assigned type, so the edit script need not create additional ports on the component instance; it may rename a port to match the name declared in the architectural type. We generally allow the user to rename any architectural element in the edit script: for example, in our case study presented in Section 5, we gave the more meaningful name *windowBus* to a connector with an automatically generated name.

The edit script is also checked for some common problems: e.g., the tool raises warnings for architectural elements without an assigned type, or errors, such as having an element name corresponding to a reserved Acme keyword. Currently, we do not check that the edit script will produce a valid architectural model before it is applied, e.g., check that it will not generate a dangling port. This is deferred for future work.

5. Case Study

We now illustrate our approach and tool support on an ArchJava implementation of a pedagogical circuit layout application, Academic Physical Design System (Aphyds) with over 20 components divided into several subsystems. In [ACN02], Aldrich discusses how Aphyds was re-engineered from 8,000 source lines of Java code (not counting the libraries used) to take advantage of the architectural features of ArchJava.

The architect evaluating our synchronization tool was familiar with ArchJava but was not previously involved with the development of the Aphyds system. He started out with the original Aphyds architect's drawing of the conceptual architecture shown in Figure 3. The architecture loosely follows the Model-View-Controller style, with the *views* consisting of user interface elements shown above the line in the middle

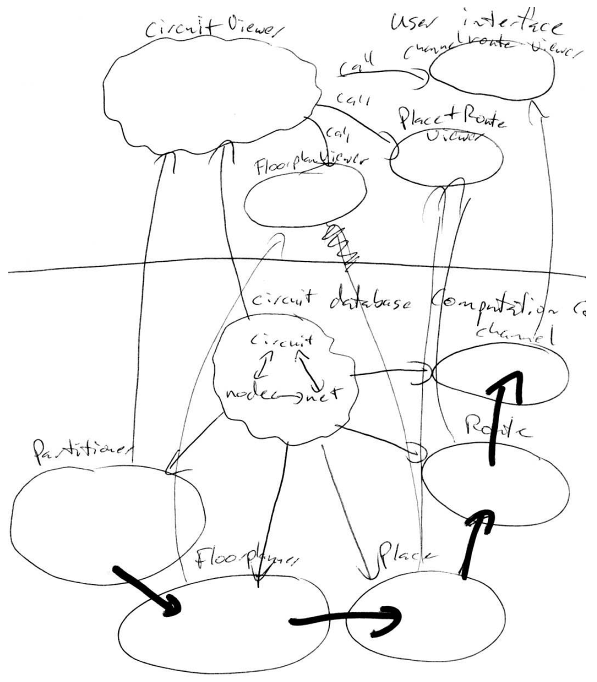

Figure 3: Original Java developer's model.

of the diagram and the *model* consisting of a circuit database and a set of computational components shown below the line; also the arrows labeled "call" correspond to control flow whereas the unlabeled arrows correspond to data flow. With this information, the Acme architect designed a conceptual C&C view for the system (See Figure 4): he created a single Acme component to represent the *circuitModel* and added all the components below the line to a representation of *circuitModel* (See Figure 5). The architect did not want to distinguish between data and control flow, so he added the data flow arrows as connectors. Finally, he used the following convention to carefully name the various elements: component instances start with lowercase to distinguish them from component types; a port or a role involved in a two-way connection is named after the component it is attached to through the connector; otherwise, a reasonably meaningful name is chosen. The Acme architect did not initially assign any architectural types to the model, since he was anxious to explore the ArchJava implementation, and see how well it matches the conceptual view. The architect ran the synchronization tool: carefully naming the elements was helpful since names are the main visual cue available when viewing the architectural C&C view in a tree. He noticed a few renames, e.g., ArchJava uses *model* instead of *circuitModel*, and in that representation, ArchJava uses *globalRouter* instead of *route* (See Figure 2).

The Acme architect was the least sure about how he represented the *circuitModel* component in Acme; facing a number of name differences certainly did not raise his confidence level. So, he decided to focus on the *circuitModel* Acme component instance which was matched to the *model* ArchJava component instance. He noticed that the ArchJava *model* component instance was being incorrectly interpreted by the synchronization tool as having only bindings, i.e., no connectors! This was an example of the expressiveness gap discussed earlier: ArchJava was using the *glue* primitive to connect inner and outer instances and ports, the equivalent of an illegal Acme construct of binding an outer port to a role on an inner connector:

```
glue circuit to circuitData.main,
partitioner.circuit, floorplanner.circuit,
placer.circuit, globalRouter.circuit,
channelRouter.circuit;
```

Since Acme cannot express this ArchJava construct directly, the developer chose to refactor the Archjava code to more closely reflect his Acme design. Our tool cannot yet propogate changes from Acme to ArchJava automatically, so the Acme architect looked at the properties of the ArchJava *model* component instance, copied the name of its source file

(*AphydsModel.archj),* switched to the ArchJava development environment within the same running instance of the Eclipse environment, browsed that source file and manually changed the ArchJava code to the mostly equivalent ArchJava construct:

```
glue circuit to circuitData.main;
connectcircuitData.main,partitioner.circuit,
floorplanner.circuit, placer.circuit,
globalRouter.circuit,channelRouter.circuit;
```

The Acme architect then switched back to AcmeStudio, and restarted the synchronization wizard and confirmed that the ArchJava code modification had the intended effect. However, the structural comparison then showed that the Acme representation for *circuitModel* had more connectors than the ArchJava implementation, i.e., the tool only matched *starConnector* in the middle of the diagram, modulo renaming (See Figure 2). The architect investigated this further and confirmed that the dataflow arrows in the informal Aphyds boxes-and-lines diagram are not actually in the implementation, so he accepted the edit actions to delete the extra connectors from the Acme model (See Figure 5).

Having synchronized the *circuitModel* component, the Acme architect next turned his attention to the top-level system. The synchronization tool had alerted the Acme architect to the presence of additional representations for components *channelRouteViewer*, *placeRouteViewer*, and *circuitViewer*. The architect decided against adding those sub-systems to the Acme model, so he cancelled the corresponding edit actions. The architect next turned his attention to the additional top level component, shown as *privateAphyds* in Figure 2): he discovered that specific component was added to represent a private port in ArchJava and the corresponding glue, a limitation of Acme discussed earlier. By looking at the required and provided methods and the control flow, the architect decided to have that subsystem follow the publish-subscribe style, so he renamed component *privateAphyds* as *window* and renamed the added connector to *windowBus.* The architect also decided to use the same component names as the ArchJava implementation to future avoid confusion, so he let the tool apply the edit script.

The architect then decided to assign styles and types to the model. He reran the synchronization wizard using it to assign types. As he was interested in the control flow in the system, he assigned the *provideT, useT, provreqT* Acme types to ArchJava ports which only provide, only require, or have both methods, respectively; he assigned the generic *TierNodeT* Acme type to all components, the *CallReturnT* Acme type to all connectors, except the previously created *windowBus* connector, which was assigned the *EventBusT* connector type from the Publish-Subscribe

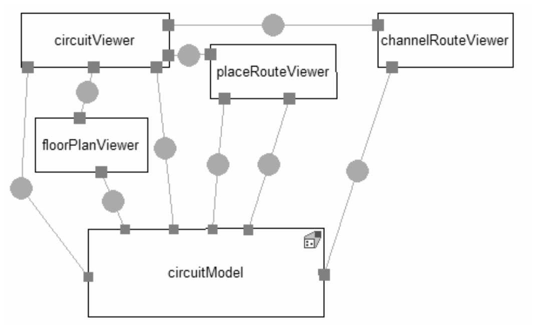

Figure 4: Original developer's model in Acme.

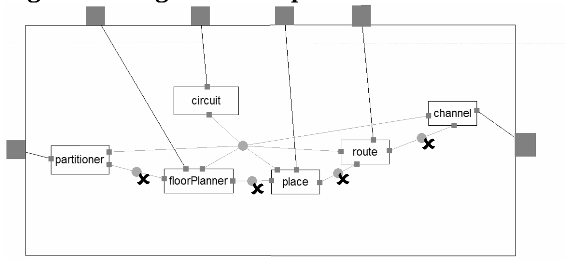

Figure 5: Acme representation for the *circuitModel* component. Extra connectors are marked with ✖.

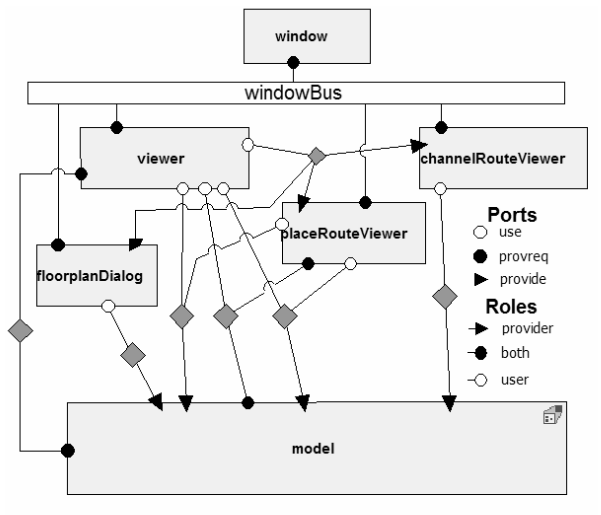

Figure 6: Acme model with styles and types.

style. Figure 6 shows the C&C view after it has been manually laid out in AcmeStudio. Unlike the original architect's model, Figure 6 shows bi-directional communication taking place between components *placeRouteViewer* and *model*; upon further investigation, the architect traced that to a callback. Since Aphyds is a multi-threaded application with long running operations moved onto worker threads, the architect makes note of the fact that developers should not carelessly add callbacks from a worker thread onto the user interface thread. Finally, the architect plans on using the up-to-date C&C view with types and styles for evolving the system.

6. Related Work

Many researchers have studied ensuring conformance between architecture and implementation, often within the context of architectural recovery. Some of the challenges we identified, such as mapping both types and instances, are typical of issues involved in representing architectures using multiple views or models such as UML [AM99][ICG+04][HK03].

Murphy et al. [MNS01] also follow an incremental, lightweight, approximate approach to check the actual architecture against the idealized one. The work on Reflexion Models and Hierarchical Reflexion Models [KS03] appears to be mostly concerned with module views and not with C&C views. In Reflexion Models, the source model and the high-level models can be typed, partially typed or un-typed; similarly, assigning types is an optional step in our approach. Having the user match Acme and ArchJava types or specify additional types on the edit script during synchronization supports the same "goal of a lightweight technique by reducing the burden on the engineer to define a type for each high-level model interaction" with a "focus on those parts of the model where typing will provide the most benefit". In Reflexion Models, a minimal representation of types is used, i.e., names, whereas Acme types have additional semantics, constraints and heuristics associated with them [Mon98]. Just as Reflexion Models let the user elide information from view and permit inconsistencies to remain, we allow the user to cancel any unwanted edit actions and can restrict the structural comparison to a subset of the tree-structured data.

Medvidovic et al [MJ04] also attempt to check the conformance of an implementation with respect to an architectural style. However, in their approach, the idealized architecture is not represented using C&C views: they mainly employ architectural recovery techniques and manually relate the two views.

For maximum generality, we match elements based on their structure and do not assume that architectural or implementation-level elements have unique identifiers associated with them. In some approaches, the names are immutable—every time an element is changed, it gets a new unique identifier [OWK03], so checking for renames is not needed anymore. Assuming unique identifiers may be possible when comparing two versions of the same model, but is not appropriate when dealing with different levels of abstractions. As an optimization to our system, persistent unique identifiers could be assigned to Acme and ArchJava elements so that they could be quickly matched up between invocations, or to automatically ignore previously flagged known differences in a large architectural model. The identifiers could be persisted outside the Acme model or the ArchJava source code, to keep the synchronization unobtrusive.

Tracking changes to an architectural representation in an ADL using features similar to those in source control management systems (e.g., Mae [RHM+04]) may provide the ability to infer coarse grained operations, such as merges or splits, in addition to the fine-grained operations (inserts, deletes, …). For maximum generality, we assume a disconnected operation, i.e., no monitoring of structural changes is taking place while the user is modifying the Acme model or the ArchJava implementation. The connected operation seems less appropriate when dealing with different levels of abstraction. Furthermore, even having accurate, fast and reliable structural comparison algorithms will not completely eliminate some of the manual steps involved, such as matching the type structures between the two representations.

Work on finding differences between inheritance trees [XS03] inspired the use of tree-to-tree correction algorithms. However, most approaches use variants of tree-to-tree correction for ordered labeled trees mentioned earlier. We discussed earlier how the problem calls for potentially taking into account a large number of name differences: even if the implementation-level architecture structurally conforms to the conceptual-level architecture, such as right after code generation, name differences will be found. Reliably detecting renames requires using unordered tree-to-tree correction. ArchDiff [WH02] detects inserts and deletes, but not renames nor moves, and seems to be using a simpler comparison algorithm. ArchDiff only compares two architectural models in xADL and does not have to bridge any expressiveness gaps. Our implementation could be readily adapted to compare and synchronize two architectural models.

7. Limitations and Future Work

In this paper we have described a lightweight, scalable, semi-automated, incremental approach for synchronizing an implementation-oriented C&C architectural view with a design-level architectural C&C view described in an ADL. We have presented a tool that implements this approach to provide synchronization between Acme and ArchJava.

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.

In general, it may not be feasible to make incremental changes to an implementation in a programming language that does not encode architectural structure, or if the C&C view is obtained by instrumenting a running system [YGS+04], or when dealing with C&C views with structural dynamism. The latter case has not been addressed mainly since Acme currently cannot express the dynamic constructs ArchJava can. Finally, we plan to address in future work some of the incidental differences encountered during this research to further streamline synchronizing the Acme and ArchJava architectural representations.

8. References

[AM99] Abi-Antoun, M. and Medvidovic, N. Enabling the Refinement of a Software Architecture into a Design. In Proc.of «UML» 99, 1999.

[AP03] Alanen, M. and Porres, I. Difference and Union of Models. In Proc. of «UML» 2003, 2003.

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

[CG01] Cheng, S.-W. and Garlan, D. Mapping Architectural Concepts to UML-RT. In Proc. of PDPTA, 2001.

[Erd98] Erdogmus, H. Representing Architectural Evolution. In Proc. CASCON' 98, 1998.

[EOG+98] Eixelsberger W., Ogris M., Gall H., and Bellay B. 1998. Software architecture recovery of a program family. In Proc. ICSE, 1998.

[GKC01] Garlan, D., Kompanek, A. J., and Cheng, S.-W. Reconciling the Needs of Architectural Description with Object-Modeling Notations. In Science of Computer Programming, Volume 44, Elsevier Press, 2001.

[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.

[HK03] Hausmann, J. H., Kent, S. Visualizing Model Mappings in UML. In Proc SOFTVIS 2003, 2003.

[ICG+04] Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B. and Silva, J.O. Documenting Component and Connector Views with UML 2.0. CMU/SEI-2004-TR-008, Software Engineering Institute, 2004.

[KS03] Koschke, R., and Simon, D. Hierarchical Reflexion Models. In Working Conf. on Reverse Eng., 2003.

[KPS+99] Krikhaar, R., Postma, A., Sellink, A., Stroucken, M., Verhoef, C. A Two-Phase Process for Software Architecture Improvement. In Proc. ICSM, 1999.

[MJ04] Medvidovic, N., and Jakobac, V. Using Software Evolution to Focus Architectural Recovery. In Journal of Automated Software Engineering, 2004.

[MOR+96] Medvidovic, N., Oreizy, P., Robbins, J. E. and Taylor, R. N. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In Proc. FSE 1996.

[MT00] Medvidovic, N., and Taylor, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. In IEEE Transactions on Software Engineering, vol. 26, no. 1, pp.70–93, 2000.

[Mon98] Monroe, R.T. Capturing software architecture design expertise with Armani. Technical Report No. CMU-CS-98-163, Carnegie Mellon University, 1998.

[MNS01] Murphy, G. C., Notkin D., and Sullivan K. Software Reflexion Models: Bridging the Gap Between Design and Implementation. In IEEE Transactions on Software Engineering, vol. 27, no. 4, pp. 364–380, 2001.

[Ecl03] Object Technology International, Inc. Eclipse Platform Technical Overview, 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf

[OWK03] Ohst, D., Welle, M., and Kelter, U. Differences between Versions of UML Diagrams. In ESEC/SIGSOFT FSE, 2003.

[Par72] Parnas, D. On the Criteria for Decomposing Systems into Modules. In Communications ACM, 15 (12), 1972.

[RHM+04] Roshandel, R., van der Hoek, A., Mikic-Rakic, M. and Medvidovic, N. Mae - A System Model and Environment for Managing Architectural Evolution. In ACM Transactions on Software Engineering and Methodology, 13(2), pages 240-276, 2004.

[ROW98] Rushby, J., Owre, S., and Shankar, N. Subtypes for Specifications: Predicate Subtyping in PVS. In IEEE Trans. Software Engineering 24(9), 1998.

[SG96] Shaw, M. and Garlan, D. Software Architectures: Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[SDK+95] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. Abstractions for Software Architecture and Tools to Support Them. In IEEE Trans. Software Engineering, 21(4), April 1995.

[SG04] Schmerl, B. and Garlan, D. AcmeStudio: Supporting Style-Centered Architecture Development. In Proc. Int'l Conference on Software Engineering, 2004.

[SZ97] Shasha, D., Zhang, K. Approximate Tree Pattern Matching, in Pattern Matching Algorithms, Apostolico, A. and Galil, Z., Eds., Oxford University Press, 1997.

[TMR02] Telea, A., Maccari, A. and Riva, C. An open visualization toolkit for reverse architecting. In Proc. 10th Int'l Work. on Program Comprehension, 2002.

[THP05] Torsello, A., Hidovic-Rowe, D. and Pelillo, M. Polynomial-Time Metrics for Attributed Trees. To appear in IEEE Transaction on Pattern Analysis and Machine Intelligence, 27 (7), 2005.

[WDC03] Wang, Y., Dewitt, D.J. and Cai, J.-Y. X-Diff: An Effective Change Detection Algorithm for XML Documents. In Proc. 19th Int'l Conf. Data Eng., 2003.

[WF74] Wagner, R.A. and Fischer, M.J. The string to string correction problem. Journal of the ACM, 21:168--173, 1974.

[XS03] Xing, Z., and Stroulia, E. Understanding Object-Oriented Architecture Evolution via Change Detection. Technical Report TR03-20, University of Alberta, 2003.

[YGS+04] Yan, H., Garlan, D., Schmerl, B., Aldrich, J. and Kazman, R. DiscoTect: A System for Discovering Architectures from Running Systems. In ICSE, 2004.

[ZJ94] Zhang, K., and Jiang, T. Some MAX SNP-hard results concerning unordered labeled trees. In Information Processing Letters, 49, pp. 249–254, 1994.

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