Improving System Dependability by Enforcing Architectural Intent

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

ABSTRACT

Developing dependable software systems requires enforcing conformance between architecture and implementation during software development and evolution. We address this problem with a multi-pronged approach: (a) automated refinement of a component-and-connector (C&C) architectural view into an initial implementation, (b) enforcement of architectural structure at the programming language level, (c) automated abstraction of a C&C view from an implementation, and (d) semi-automated incremental synchronization between the architectural and the implementation C&C views.

We use an Architecture Description Language (ADL), Acme, to describe the architecture, and ArchJava, an implementation language which embeds a C&C architecture specification within Java implementation code. Although both Acme and ArchJava specify C&C views, a number of structural differences may arise. Our approach can detect structural differences which correspond directly to implementation-level violations of the well thought-out architectural intent. Furthermore, supplementing the C&C view extracted from the implementation with architectural types and styles can uncover additional violations.

Categories and Subject Descriptors

D.2.11 [**Software Architecture**]: Languages

General Terms

Documentation, Design, Languages, Verification.

1. Introduction

The software architecture of a system defines its high-level organization as a collection of interacting components, connectors, and constraints on interaction, along with additional properties defining the expected behavior. Over the past decade, various software architecture models and analyses, studying reliability [9, 20], performance [4] or graceful degradation [24], have been developed and applied to real-world systems. However, dependability analyses at the architectural level are accurate in their predictions of actual dependability in the realized system only if the system is implemented and maintained according to its architecture. The development of a dependable software system therefore calls for fault-prevention and fault-removal [3] of violations of the architectural intent. Koopman [11] notes that few system architectures are completely elaborated when the first implementation is built; sometimes, developers produce an implementation before even documenting the architecture. More often, developers work on the implementation without maintaining the architectural model, which quickly becomes outdated. In some cases, developers may introduce subtle structural differences that invalidate key architectural design intent. As a result, architects often deal in their analyses with incomplete and incorrect knowledge due to documentation or implementation defects.

We address detecting and correcting such differences, and ensuring conformance between architecture and implementation with a multi-pronged approach: (a) automated refinement of an architectural component-and-connector (C&C) view into an initial implementation, (b) enforcement of architectural structure at the programming language level, (c) automated abstraction of a C&C view from an existing implementation, and (d) semi-automated incremental synchronization between the architectural C&C view and the one extracted from the implementation.

We use Acme [8] as an example of a mature general purpose Architecture Description Language (ADL) to describe the architecture, taking advantage of Acme's support for extensible, domain-specific architectural styles as well as extensible properties and architectural analyses. We assume the implementation is represented in ArchJava [1], which embeds a C&C specification within Java code. Any changes made by the engineers are at least reflected in ArchJava's representation of architectural structure. However, ArchJava does not currently enforce other important architectural attributes such as architectural style. Using ArchJava greatly facilitates extracting a C&C view from an existing implementation. However, C&C views can also be extracted from implementation-constraining ADLs with code generation capabilities or implementation independent ADLs such as C2 [13] that provide an implementation framework for code generation. Incremental synchronization, the primary contribution of our approach, could in principle be applied in any of these settings.

Structural comparison of the architectural C&C view and the implementation C&C view only detects implementation-level violations of architectural structure. Setting applicable architectural types and styles on the implementation C&C view can uncover additional violations. The architect can further enrich the up-to-date architectural model with additional constraints, heuristics and properties. Having an up-to-date architectural model increases the accuracy of architectural analyses to predict various dependability attributes in the implemented system. For instance, some analyses can statically detect architectural mismatches during development; only handling mismatches at runtime requires designing more fault-tolerance into the system [6], resulting in additional complexity.

ICSE 2005 Workshop on Architecting Dependable Systems (WADS)
May 17, 2005, St. Louis, Missouri, USA.

Having an up-to-date C&C view of the implementation also enables the architect to incorporate new requirements and new insights into the architecture, and perform a change impact analysis by re-running the synchronization, viewing differences, and stopping short of making any changes to the implementation.

We are interested in both how the implementation-level C&C view differs from the architectural C&C view, and how the architectural C&C view differs from the implementation C&C view. Since Acme and ArchJava both encode C&C views, one might suspect that synchronization is trivial. However, with the similarities come a number of crucial differences that make synchronization non-trivial. Acme is an architecture description language, whereas ArchJava is primarily an implementation language. This requires a notion of conformance or correspondence, where not all ArchJava elements are carried over to the Acme model and vice versa, as well as being able to account for a large number of name differences between the two representations. Some of the technical challenges include:

- Acme views types as logical predicates over an architecture (many of which ArchJava cannot currently express), and ArchJava views types as a particular interface of provided and required functionality; this requires matching and synchronizing types in addition to instances;
- ArchJava does not have named connectors or roles, whereas Acme does; this requires matching them modulo renaming;
- Acme's type system can mandate ports with specific types and names on instances of a given component type, whereas ArchJava is more flexible with the naming of ports and does not declare port types; this requires forbidding some port renames and complicates assigning architectural types to ArchJava ports; the same problem arises with connector types and roles;
- Acme requires types for roles, whereas ArchJava does not even have first-class roles; this requires inferring types of roles whenever possible;
- Acme can leave out required and provided methods on ports, whereas ArchJava's type system mandates that each required method is bound to a provided method with the same name and signature; therefore conformance should not always require that information to avoid false positives;
- Acme's type system is very flexible; ArchJava's type system mandates that a component subtype cannot require more methods than its supertype to preserve component substitutability; this prohibits generating ArchJava component types corresponding directly to Acme component types defined in Acme's architectural families intended to be shared across systems;
- Acme views hierarchy as design-time composition, while nesting in ArchJava has implications for component lifetime and data sharing; this requires detecting moves across hierarchy levels.

The differences between Acme and ArchJava are typical of those one might find between any design and implementation language. And some of the challenges, such as mapping both types and instances, are typical of issues involved in representing architectures using multiple views or models such as UML [10].

2. Structural Conformance

Tool support for our approach uses AcmeStudio [22], a domain-neutral architecture modeling environment for Acme, and ArchJava's development environment, both implemented as plugins in the Eclipse tool integration platform [17]. Our code generation capability can generate ArchJava skeleton code from the architectural model to prevent early structural differences which will likely deepen as the system evolves. Merely generating skeleton implementation code is not enough: it would be ideal to be able to regenerate code without overwriting any manual changes to existing files. In that case, code generation becomes a special case of incremental synchronization. More generally, at any point during development or maintenance, we would like to incrementally synchronize the C&C views between architecture and implementation. We have completed initial tool support 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.

Incremental synchronization 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, and renames), 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, 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 want to avoid changing the implementation and performing regression tests, or may simply be interested in a change impact analysis.

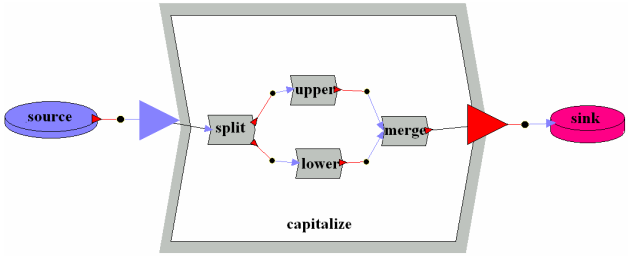

Figure 1: Acme model for the *CaPiTaLiZe* system.

```
public component class Capitalize {
    private final Upper upper = new Upper();
    private final Lower lower = new Lower();
    private final Split split = new Split();
    private final Merge merge = new Merge();

    public port portIn {
        requires char getChar();}
    public port portOut {
      provides char getChar();}

    connect lower.portOut, merge.portIn1;
    connect split.portOut2, lower.portIn;
    connect upper.portIn, split.portOut1;
    connect merge.portIn2, upper.portOut;

    glue portOut to merge.portOut;
    glue portIn to split.portIn;
}
```

Figure 2: ArchJava implementation of component *capitalize*.

We illustrate our approach using a simple but instructive pipe-and-filter system, presented in Figure 1, which consists of a data source component (*source*), a data sink component (*sink*), a filter component (*capitalize*), and two connectors (character pipes) connecting them. Component *capitalize* converts characters it receives from component *source* alternatively to upper or lower case before passing them on to component s*ink*. Component *capitalize* is further decomposed into a sub-architecture consisting of another pipe-and-filter system (shown as an Acme representation in Figure 1). Figure 2 shows how developers specify components, connectors, and port constructs in ArchJava, and relate object instances to these architectural constructs, while completing the implementation using the Java programming language.

2.1 Bridging the Gap

We synchronize architectural type and instance information (such as components, ports, methods, connectors, roles, attachments, and bindings). The 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 visualization manageable.

Obtaining the architectural tree-structured data is simply a matter of converting the Acme architectural graph into the cross-linked tree structure. We also add information to the view to improve the accuracy of the structural comparison. For instance, the sub-tree of a node corresponding to a port or role includes all the port's or the role's involvements, i.e., all components or connectors reachable from that port or role through attachments or bindings. This information is also useful for processing renames and deletes in the edit script, e.g., to delete any dangling attachments when a port is deleted.

The tree-structured data from the implementation is obtained by traversing the ArchJava compilation units, ignoring non-architecturally relevant Java classes or fields that are not of type component or connector. Not all information is readily available: (a) ArchJava does not have named connectors or connector roles; they are named after the components and ports they connect; (b) the ArchJava top-level component can have ports, whereas the top-level component in Acme, i.e., the system, cannot; one option is to create a top-level component in Acme to correspond to ArchJava's top-level component; (c) ArchJava ports can be private, whereas all Acme ports are public; one option is to represent ArchJava private ports as Acme ports on an internal component instance; (d) Acme does not have a first-class construct for required and provided methods; in keeping with Acme's model for extensible properties, we create properties on ports to represent methods, as well as other properties retrieved from the source code (e.g., visibility, dynamism, synchronization, …).

2.2 Tree-to-Tree Correction

We then use tree-to-tree correction between the tree-structured data from the architecture and from the implementation views to find structural differences and produce an edit script. We can restrict the comparison to user-defined subsets of the two views: 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 false positives. The structural comparison finds matches, and classifies differences as inserts, deletes and renames.

Detecting renames (i.e., not treating a rename as an insert-delete pair) is crucial, as there will always be name differences between Acme and ArchJava. For instance, a port can be named "in" in ArchJava, a reserved keyword in Acme; furthermore, names of Acme connectors and roles are lost when generating ArchJava code; in addition, port names in Acme are used to typecheck the model. Detecting renames is important for the general comprehension of an existing implementation. Names are often modified during software development and maintenance: a name may be turn out to be inappropriate or misleading due to either careless initial choice or name conflicts from separately developed modules [2]. Furthermore, developers tend to avoid using names that may be confused with names in use by an implementation framework or library, a minor detail as far as the architect is concerned.

The problem domain clearly calls for treating the tree-structured information as unordered labeled trees, since there is no ordering between subcomponents of a given component. We initially implemented an exact polynomial time tree-to-tree correction

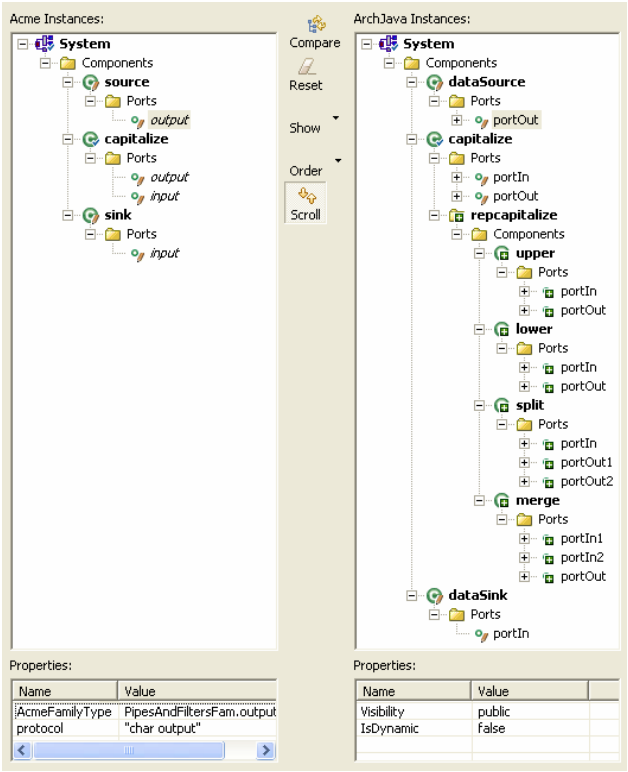

Figure 3: Comparing an Acme model without a representation for component *capitalize* to an ArchJava implementation with an implementation of the *capitalize* component. Ports shown in italics are inherited from the component type. Nodes shown in bold indicate differences in the subtrees. With selection tracking, the matching element of a renamed element is automatically selected and made visible in the other tree (highlighted nodes).
Symbols: Match (✓), Insert (+), Delete (×), Rename (✎)

algorithm for ordered labeled trees based on [23], simply ordering nodes by name. We evaluated it and confirmed that it fares poorly when renames change the ordering of sibling nodes. Most approaches for comparing tree-structured data have paid little attention to the unordered comparison since it has been shown to be MAX SNP-hard [28]. Our current implementation for unordered labeled trees, a polynomial-time approximation algorithm based on [25], produces accurate results even on non-trivial test cases exhibiting a large number of differences (see Figure 4), while offering interactive performance. For maximum generality, our approach is stateless and does not associate unique identifiers with architectural or implementation-level elements. In particular, it can be used even if the implementation was not based on skeleton code produced by automatic code generation from the architectural model.

2.3 Architectural Completeness

We next illustrate how our approach can uncover various architectural defects, such as those identified and classified by Roshandel et al in [21]. Many architectural defects fall into the class of usage or completeness. For instance, in Figure 3, we detect that ArchJava components *dataSource* and *dataSink* match Acme components *source* and *sink* respectively. Furthermore, if we remove the representation for the *capitalize* component in Acme and compare the resulting architecture to the same implementation, the comparison reveals the missing sub-architecture, namely components *lower*, *merge*, *split*, and *upper:* applying the edit script creates an Acme representation for component *capitalize* with the additional components, ports, connectors, roles, attachments, and bindings (parts of the edit script are shown in Figure 8).

2.4 Element-Level Conformity

Structural comparison can detect unintended structural differences in the implementation of specific architectural elements. For instance, when we compare one of our earliest ArchJava implementation of a *CaPiTaLiZe* system (which predates the Acme model) to the Acme model in Figure 1, the tool correctly detects a large number of differences. Structural comparison correctly identified the additional component (character buffer *b*) in ArchJava, and matched many of the renames, e.g., it matched ArchJava *m* to Acme *merge*. However, it did not initially match ArchJava components *s*, *u* and *l* to Acme components *split*, *upper*, and *lower* respectively. Upon further analysis, we discovered that ArchJava component *s* was implemented with one input port and one output port, whereas Acme *split* component has one input and two output ports. This made ArchJava component *s* structurally undistinguishable from ArchJava components *u* and *l*. Although it would have been acceptable to match *lower* to *u* and *upper* to *l* since components *u* and *l* are architecturally equivalent (i.e., have the same architectural type), clearly component *s* was incorrectly implemented. Since we do not yet provide the ability to manually force a match between Acme component *split* and an incorrectly implemented ArchJava component *s* (with one output port) without leaving the synchronization tool, the architect has to cancel the synchronization, correct the implementation of ArchJava component *s* using ArchJava's development environment, and resume the synchronization. With the above change, the structural comparison can now correctly match all the renames (See Figure 4). But there are cases where manual

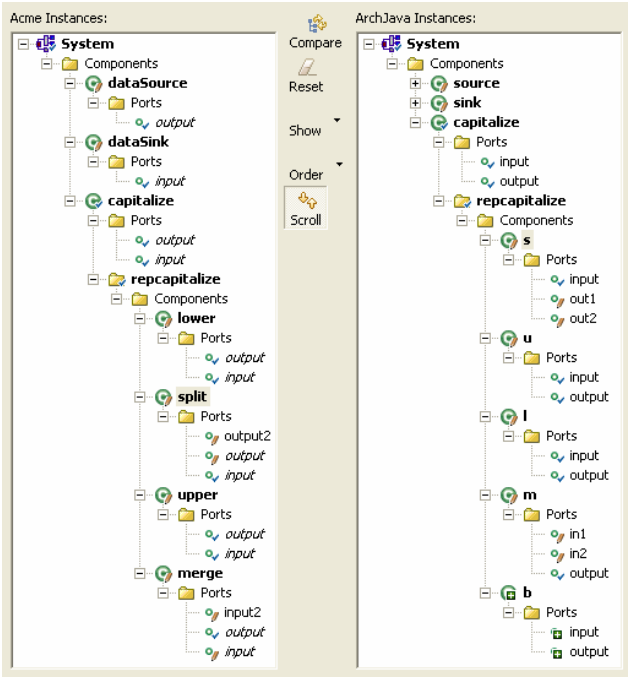

Figure 4: Comparing an early implementation of *CaPiTaLiZe* to the Acme model in Figure 1 after correcting ArchJava component *s*. Despite a large number of differences, unordered tree-to-tree correction correctly detects the inserted component *b*, and correctly matches renamed components *source, sink, s, u, l, m* to *dataSource, dataSink, split, upper, lower,* and *merge* respectively.

overrides will still be required, and we are planning on implementing that feature.

2.5 Behavioral Conformance

Our work to date addresses only a few of the behavioral defects listed in the taxonomy of architectural defects [21], namely interfaces and signatures. For instance, we retrieve signatures of required and provided methods on component ports from ArchJava and from architectural properties on component ports in Acme (if available), and include that information in the structural comparison. Currently, we do not perform additional static analyses of the ArchJava source code. However, there could be value in retrieving and representing as architectural properties, certain source code properties (e.g., thrown and handled exceptions), and design architectural analyses to use them. For instance, Ferreira et al. [7] argue that representing exception handling at the architectural level is important in the development of dependable systems.

2.6 Hierarchical Decomposition

The tree-to-tree correction algorithm currently treats a move as an insert and delete pair, and is being enhanced to recognize elements that have been moved across levels of the hierarchy. This situation can occur when a component or a connector is replaced with its representation. In some cases, the architect may want the implementation to respect the chosen hierarchical decomposition to insulate parts of the system from certain changes using the principle of information hiding [19]. If the outer component simply delegates to its inner components,

efficiency considerations may dictate replacing the component with its representation in the implementation. However, in the interest of managing complexity, the architect may still want a hierarchical architectural model to modularly reason about the properties of a given element.

3. Styles and Types Conformance

As discussed earlier, structural comparison between the C&C views retrieved from architecture and implementation can only find violations of architectural structure. Architectural types and styles are used to further validate the implementation against the architectural intent. Architectural types and styles have to be explicitly provided by the architect or inferred if possible.

3.1 Setting Types and Styles

Assigning architectural styles and types to implementation-level elements clarifies the architectural intent by enforcing any constraints (e.g., invariants and heuristics) associated with those types. For instance, a constraint on a component type may specify that all instances of that type must have exactly two ports, a constraint that cannot be directly enforced by ArchJava. Similarly, setting architectural styles on the overall system (and on each representation in that system, if applicable) enforces any constraints associated with the style. For instance, the pipe-and-filter style can prohibit cycles, a constraint that ArchJava, as a general purpose implementation language, does not directly enforce. Specifying architectural types and styles is optional; in that case, the synchronized elements will be untyped. Setting the types can still be done at a later point in AcmeStudio. Although setting types during synchronization seems like duplicating functionality already present in AcmeStudio, it is important since types affects 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. Finally, the visualization of architectural elements in AcmeStudio is heavily type- and style-dependent. Constraint violations do not become apparent until the architect is viewing the edited architectural model in AcmeStudio which provides user-assistance to resolve errors and warnings representing violations of constraints and heuristics [14].

Assigning architectural types to each implemental-level element during synchronization can be laborious. To reduce the burden on the architect, we provide an optional step for matching types. When the architect matches a specific ArchJava component type with one or more Acme component types; all ArchJava component instances of the ArchJava type are assigned the corresponding Acme types. Since ArchJava ports are not typed, ports on the component types are individually assigned Acme port types (Figure 5). We also support a limited form of wildcards: assigning Acme types (e.g., *Pipe)* to the wildcard ArchJava connector type *ANY* assigns those types to all ArchJava connector instances. In order to further reduce the burden on the architect, we infer architectural types whenever possible: if the architect assigns types to components, ports and connectors, we use AcmeStudio's connection patterns defined for each architectural style to infer the type of the connector roles (e.g., *sourceT*) based on the source component type (e.g., *Filter*), source port type (e.g., *inputT*), and connector type (e.g., *Pipe*).

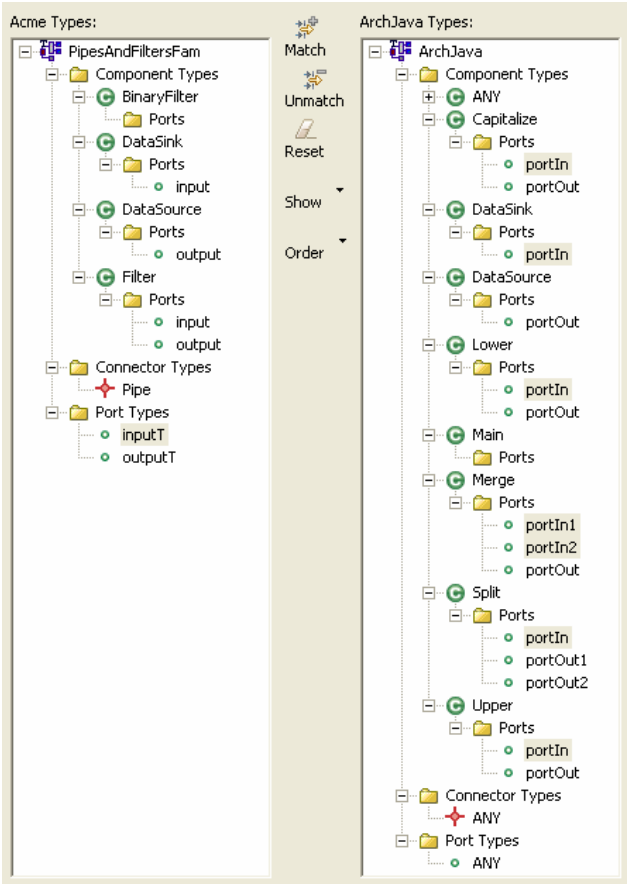

Figure 5: Matching ArchJava types with Acme types: multiple selection is used to match individual ports on ArchJava component types to Acme port types.

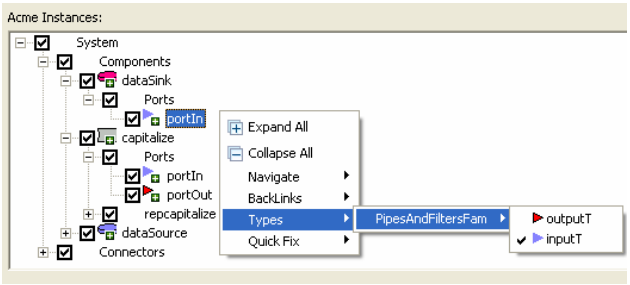

Figure 6: Changing type assignments: elements are now using the same visualization as AcmeStudio. Unchecking an element cancels the corresponding edit actions.

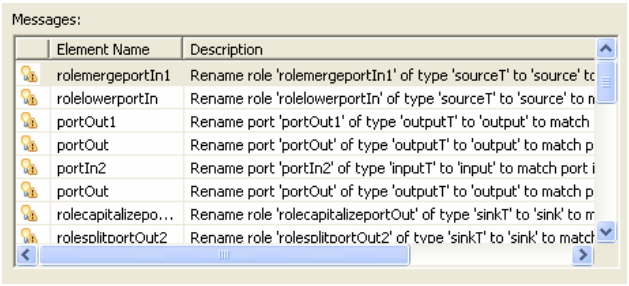

Figure 7: Checking the edit script for errors and warnings.

Changing the types of existing elements, and overriding automatically inferred types are also supported (See Figure 6).

3.2 Validation of the Edit Script

The edit script is validated to ensure it will produce a valid architectural model. For instance, the tool raises warnings, such as having an architectural element without an assigned type, or errors, such as having an element name corresponding to a reserved Acme keyword. In many cases, the tool suggests ways to fix the problem; it can rename a port to match the name of the port declared in the component type (See Figure 7). If the architect accepts the corrective actions, the tool finds and updates all the cross-references in the edit script. The architect can also cancel unwanted edit actions, e.g., cancel the deletion of recent additions to the Acme model that are not yet reflected in the ArchJava implementation.

4. Related Work

Many researchers have studied ensuring conformance between architecture and implementation. Murphy et al. [15] also follow an incremental approach to check the actual architecture against the idealized one. The work on Reflexion Models and Hierarchical Reflexion Models [12] appears to be mostly concerned with module views, and not with C&C views. Unlike Reflexion Models, we do not currently maintain a mapping between the implementation C&C view elements and the corresponding elements in the source code (module view); we may add that mapping to support applying the edit script to an implementation. There are additional similarities with the Extended Reflexion Models that support typing and tagging [16]. 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 our case, implementation-level violations of architectural intent. In Reflexion Models, a minimal representation of types is used, i.e., names, whereas Acme types have additional semantics and constraints associated with them. Just as Reflexion Models permit inconsistencies to remain, we allow the user to cancel any unwanted edit actions. Reflexion Models let the user elide information from view; we can also restrict the structural comparison to a subset of the underlying tree-structured data.

Work on finding differences between inheritance trees [26] and class diagrams [18] inspired the use of tree-to-tree correction algorithms. However, most approaches use variants of tree-to-tree correction for ordered labeled trees. We discussed earlier how mapping between architecture and implementation has to potentially take into account a large number of name differences, even if the implementation is in complete structural conformance with the architecture. Reliably detecting renames requires using unordered tree-to-tree correction.

ArchDiff (Dashofy et al. [5]), an extension schema and tool for xADL 2.0, only compares two architectural models in xADL. Furthermore, ArchDiff detects inserts and deletes, but not renames nor moves, and seems to be using a simpler comparison algorithm. Our more general implementation can be readily adapted to compare and synchronize two architectural models.

```
Rename AcmeComponent : Current Name= 'source'
        -> New Name = 'dataSource'.
...
Create Representation 'repcapitalize' on
        Component 'capitalize'.
...
Create Port 'input' on Component 'split'.
...
Create Connector 'conn_split_ouput__lower_input'.
Create Role 'sink' on Connector
        'conn_split_ouput__lower_input'.
...
Create Binding from Component 'split'
        at Port 'input' to Component 'capitalize'
        at Port 'input'.
...
Create Attachment from Component 'split'
        at Port 'output' to
        Connector 'conn_split_ouput__lower_input'
        at Role 'sink'.
...
```

Figure 8: Parts of the edit script corresponding to Figure 3.

5. Limitations

As mentioned earlier, we do not yet support applying the edit script to an existing ArchJava implementation. Retrieving a C&C view from ArchJava without abstracting any information enables us to make incremental changes to the ArchJava implementation. However, 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 [27], or in some cases of architectural dynamism.

We only addressed synchronizing C&C views without any architectural dynamism; in particular, Acme has limited expressiveness for architectural dynamism and cannot currently express the dynamic constructs ArchJava can. We also did not address behavioral conformance between architecture and implementation, a crucial element for system dependability.

The tree-to-tree correction algorithm we chose supports only low-level operations, such as inserts, deletes, renames, and potentially moves. For instance, we do not support detecting components or ports that have been split or merged during restructuring of the architecture or the implementation.

6. Future Work

Tool support for the proposed approach is still under development. We are adding various features to allow for more user control and minimize the amount of data entry. First, we want to let the user manually guide the comparison by forcing matches between elements without leaving the synchronization tool, and have the tool persist the user's overrides. Second, we want to let the user specify a hierarchical mapping from elements in the architectural C&C view to elements in the implementation C&C view (including using regular expressions) to handle a large number of naming differences more elegantly. Finally, we would like to also support a stateful mode in which persistent unique identifiers are assigned to Acme and ArchJava elements, so the comparison algorithm can quickly match elements that do not change between invocations, or simply ignore known differences in a large architectural model to speed up the comparison. The

identifiers will not be persisted in the Acme model or in the ArchJava source code, to keep the synchronization as unobtrusive as possible.

We also plan on applying the lessons learned from this research to propose changes to both Acme and ArchJava to streamline synchronizing the two representations: for instance, we consider Acme's type system overly restrictive to require a port on a component instance to have a specific name and specific types; we are also considering adding explicit types to ArchJava ports to simplify matching them with Acme ports.

The main purpose of incremental synchronization, when used during software development and evolution, is to facilitate and encourage the continuous use of architectural views and analyses throughout the software life cycle, in order to identify implementation-level violations of architectural structure that may adversely impact system dependability.

7. Acknowledgments

This work is supported by a 2004 IBM Eclipse Innovation Grant and NSF grant CCR-0204047, and was performed as part of a joint research project in Strategic Partnership between Carnegie Mellon University and Jet Propulsion Laboratory.

8. References

[1] Aldrich, J., Chambers, C., and Notkin, D. ArchJava: Connecting Software Architecture to Implementation. *Proc. International Conf. on Software Engineering*, 2002.

[2] Ammann, M. M., and Cameron, R.D. Inter-Module Renaming and Reorganizing: Examples of Program Manipulation-in-the-Large. In *Proc. International Conference on Software Maintenance*, 1994.

[3] Avizienis, A., Laprie, J.-C., and Randell, B. Fundamental Concepts of Dependability, Research Report N01145, LAAS-CNRS, 2001.

[4] Balsamo, S., Di Marco, A., Inverardi, P., and Marzolla, M. Experimenting different Software Architectures Performance Techniques: A Case Study. In *Proc. Workshop on Software and Performance*, 2004.

[5] Dashofy, E. M., van der Hoek, A., and Taylor, R. N. Towards Architecture-Based Self-Healing Systems. In *Proc. Workshop on Self-Healing Systems*, 2002.

[6] de Lemos, R., Gacek, C., and Romanovsky, A. Architectural Mismatch Tolerance. In *Architecting Dependable Systems*, de Lemos, R., Gacek, C., and Romanovsky, A., Eds., Springer-Verlag, 2003.

[7] Ferreira, G. R. M., Rubira, C.M. F., and de Lemos, R. Explicit Representation of Exception Handling in the Development of Dependable Component-Based Systems. In *Proc. Sixth IEEE International Symposium on High Assurance Systems Engineering*, 2001.

[8] Garlan, D., Monroe, R., and Wile, D. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*, Leavens, G.T., and Sitaraman, M., Eds., Cambridge University Press, 2000.

[9] Goseva-Popstojanova K., Mathur A.P., Trivedi K.S., Comparison of Architecture-Based Software Reliability Models. In *Proc. 12th IEEE International Symposium on Software Reliability Engineering*, 2001.

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

[11] Koopman, P. Elements of the self-healing system problem space. In *Proc. Workshop on Architecting Dependable Systems*, 2003.

[12] Koschke, R., and Simon, D. Hierarchical Reflexion Models. In *Working Conf. on Reverse Engineering*, 2003.

[13] 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.

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

[15] 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.

[16] Murphy, G. C., Notkin D., and Sullivan K. Extending and Managing Software Reflexion Models. Technical Report, TR-97-15, University of British Columbia, 1997.

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

[18] Ohst, D., Welle, M., and Kelter, U. Differences between Versions of UML Diagrams. In *ESEC/SIGSOFT Symp. Foundations of Software Engineering*, pp. 227–236, 2003.

[19] Parnas, D. On the Criteria for Decomposing Systems into Modules. In *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[20] Roshandel, R., and Medvidovic, N. Toward Architecture-Based Reliability Estimation. In *Proc. Twin Workshops on Architecting Dependable Systems*, 2004.

[21] Roshandel, R., Schmerl, B., Medvidovic, N., Garlan, D., and Zhang, D. Understanding Tradeoffs among Different Architectural Modeling Approaches. In *Proc. 4th Working IEEE/IFIP Conference on Software Architecture*, 2004.

[22] Schmerl, B. and Garlan, D. AcmeStudio: Supporting Style-Centered Architecture Development. In *Proc. International Conference on Software Engineering*, 2004.

[23] Shasha, D., Zhang, K. Approximate Tree Pattern Matching, in *Pattern Matching Algorithms*, Apostolico, A. and Galil, Z., Eds., chapter 14. Oxford University Press, 1997.

[24] Shelton, C. and Koopman, P. Using Architectural Properties to Model and Measure Graceful Degradation. In *Architecting Dependable Systems*, de Lemos, R., Gacek, C., and Romanovsky, A., Eds., Springer-Verlag, 2003.

[25] Torsello, A., Hidovic, D., and Pelillo, M. *Polynomial-Time Metrics for Attributed Trees*. Technical Report No. CS-2003-19, Università Ca' Foscari di Venezia, 2003.

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

[27] Yan, H., Garlan, D., Schmerl, B., Aldrich, J. and Kazman, R. DiscoTect: A System for Discovering Architectures from Running Systems. In *Proc. 26th International Conference on Software Engineering*, 2004.

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