Understanding and Propagating Architectural Changes

Christopher Van der Westhuizen and André van der Hoek
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
vanderwe@uci.edu and andre@ics.uci.edu

Abstract: Like source code, architectures change. The use of product line architectures provides a particularly rich source of changes: new products are introduced, existing products are enhanced and modified, and old products are retired. Methods exist that record these kinds of changes by maintaining explicit representations of the evolution of a product line architecture. Despite the availability of such representations, it still is difficult to quickly gain an understanding of the exact changes that define the difference between two products. Furthermore, it is difficult to automatically propagate such changes to yet another, third product in the product line. This paper aims to fill this void and contributes a set of algorithms and an associated representation for understanding and propagating architectural changes within a product line architecture. The approach is based on xADL 2.0, an extensible representation for product line architectures, and adapts well-known differencing and merging algorithms from the field of configuration management to the domain of software architecture.

Key words: Architectural change, product line architecture, differencing and merging

1. INTRODUCTION

Change is an unavoidable and intrinsic part of any kind of effort involving software architecture. Not only do individual architectures evolve over time, but the advent of product line architectures has brought the problem of managing architectural change to a whole new level [2]. New

products are continuously introduced, existing products evolve, and old products are phased out. The set of architectural changes resulting from these actions must be carefully managed. This kind of management involves addressing two key problems: (1) *capturing* architectural changes, and (2) *understanding* the architectural changes that define the difference between two products (or two versions of the same product) and *propagating* these architectural changes to yet another, third (version of a) product. The first problem has already been addressed through the advent of architectural description languages that incorporate facilities for capturing different versions of a product line architecture [10,17]. The second problem, however, has not been addressed as of yet.

Consider a situation in which a number of architects maintain a product line architecture. The product line architecture is defined as a set of core components and connectors that are shared among all of the products, and a set of per-product components and connectors that define the unique aspects of each product. Furthermore, the evolution of the product line architecture is explicitly captured at the level of individual components and connectors, at the level of the products themselves, and at the level of the overall product line architecture. One architect, responsible for maintaining a part of the product line architecture description that consists of some specific products, quits the organization. Fortunately, another architect who was responsible for those products just six months ago is able to take over. A first task for this architect is to get up to speed with the current state of the evolved products, i.e., to understand what has changed between six months ago and now. After spending quite some time examining the product line architecture description of six months ago and now, the architect gains the desired level of understanding and, in the process, realizes that a number of other products in the product line can benefit from the advances made by the changes to the products. A second task, then, is for the architect to propagate those changes from the products in which they were originally incorporated to those that can benefit.

Although it is possible for the architect to carry out all of these tasks manually, it may be an intricate job that requires a significant amount of time and effort, especially if the product line is large and contains many different versions of many different products that each consist of many components and connectors. Clearly, it is desirable that the architect has automated support, not only in this scenario, but also in similar scenarios in which it is important to understand and propagate architectural changes.

This paper begins to address this problem and is based on the recognition that understanding and propagating architectural changes bears great resemblance to a similar, long-standing issue in the field of configuration management: understanding the exact nature of source code changes as they

have been made over time and propagating selected changes from one version of a software system to another [4]. To address this problem, differencing and merging algorithms have been developed [3]. However, direct application of these algorithms to architectures would not yield the desired result. Because existing algorithms typically only operate on textual artifacts and are line-based in their operations, they cannot be aware of any specific architectural semantics and therefore offer little help, particularly in the understanding of architectural changes. Nonetheless, these algorithms form a solid basis upon which our approach is based. Specifically, we have adapted them in making three contributions to the field of software architecture. First, we have enhanced an existing representation for product line architectures, xADL 2.0 [5], with a representation in which the exact difference between two products in a product line architecture can be captured. Second, we have created a differencing algorithm that uses the representation to create an understanding of the exact set of architectural changes that constitute a difference between two products. Third, we have created a merging algorithm through which it is possible to propagate such architectural changes to other products in the product line.

The remainder of this paper details our approach and is organized as follows. In Section 2 we provide some background material regarding differencing and merging algorithms as they exist in the field of configuration management. Section 3 introduces xADL 2.0, the representation for product line architectures upon which we have based our research. Subsequently, Section 4 introduces our approach to understanding architectural change, including the representation for capturing architectural change and the differencing algorithm. Section 5 highlights the merging algorithm used for propagating architectural changes. Section 6 briefly discusses the implementation of the algorithms and we conclude in Section 7 with an outlook at future work.

2. BACKGROUND

Differencing and merging algorithms as used in the field of configuration management rely on comparing text-files on a line-by-line basis [3]. In this process, lines are atomic. A line is considered either exactly the same or completely different. A traditional configuration management differencing tool, then, takes as input two text files and outputs a “diff” containing an ordered list of those lines that have been added, removed, or replaced. This diff output is normally in textual form, but can usually be visualized if necessary.

Configuration management merge tools follow the same process. Based upon an available text file and diff, a merged result is calculated based upon lines of text. Conflicts either result in failure of the merge altogether, or are highlighted in the text such that users can manually resolve them. As of late, visual tools have greatly reduced the effort involved in merging by graphically highlighting merge results along with the input files, thereby allowing users to guide and tailor the merge algorithm to suit their needs [1,9,12].

These traditional differencing and merging algorithms typically aim to be language independent and, thus, do not further analyze or use the contents of the documents upon which they operate. Herein lies the problem with their application to the domain of understanding and propagating architectural changes within product line architectures. Although they may be able to operate on text files containing architectural descriptions, the result would be distinctly non-architectural in nature. This represents a particular problem in understanding the replacement of architectural elements. Whereas our algorithm described in Section 4.3 discovers architectural replacement semantically, a text-based differencing algorithm only would find lines of text that may have been replaced with others. Although this may accidentally coincide with the semantically desired result, such a result is dependent on the order in which architectural elements are placed in an architectural description. Since, more often than not, such placement is random and architectural elements are randomly spread throughout the description, text-based differencing almost always leads to incorrect replacement detection at the architectural level.

Recently, more semantic algorithms have been developed in a number of domains. In using abstract syntax trees, differencing and merging tools have been created that operate on, for example, UML diagrams [18]. Similarly, algorithms are now being researched that attempt to understand and interpret the difference between HTML pages [8], and XML-based differencing and merging tools have been developed that operate in terms of XML elements rather than lines of text [11]. It should be noted that direct application of the XML algorithms, although leading to higher-level results than text-based algorithms, still does not provide us with the desired level of functionality. As with text-based merging, related changes that constitute a replacement are not detected unless the changes happen to be in a consecutive part of the XML file. Nonetheless, the algorithms described in this paper fall in the same class as these semantic algorithms and build upon the results to date.

3. xADL 2.0

xADL 2.0 [5,7] is an extensible representation for product line architectures that was born out of the observation that, while each new architecture description language usually contributes some kind of unique feature, most share a relatively large set of common modeling concepts [15]. To leverage this commonality while still allowing individual advances and contributions, xADL 2.0 is constructed as a set of extensible XML schemas. To create a new architecture description language with some particular set of exclusive modeling features, an initial set of schemas is chosen that provides the base set of features. If desired, features can be modified by extending some of the selected XML schemas with new definitions of existing modeling features. Additional features are then added by writing new XML schemas on top of the selected (and possibly modified) schemas.

xADL 2.0 already incorporates a number of schemas defining common architectural elements. The cornerstone of xADL 2.0 is formed by the *Structure and Types* schema, which defines the modeling features for capturing a particular architecture at design-time. Specifically, the schema allows the definition of the basic structure of one particular architecture in terms of a set of components, connectors, interfaces, and links among those elements. In addition, the schema provides a typing mechanism through which elements in the structure can be assigned specific types.

The *Options*, *Variants*, and *Versioning* schemas extend the *Structure and Types* schema with modeling features for product line architectures. The *Options* schema allows for the definition of elements that may or may not be present upon instantiation of a particular architecture as defined per the *Structure and Types* schema. The *Variant* schema allows the definition of alternatives: depending on a property selection mechanism, an architectural element in the structure is configured to be one of multiple types. This is critical in bringing variability into the picture: by introducing specific variation points, different products can be defined in a single product line architecture. Finally, the *Versions* schema allows the modeling of the evolution of a product line architecture, in terms of each of its individual types, each of its products, and the overall product line architecture.

4. UNDERSTANDING DIFFERENCES

To understand and propagate architectural changes, it is necessary to represent them first. Therefore, we extended xADL 2.0 with an additional XML schema in which to capture architectural changes. Based on this schema, we defined two algorithms. The first performs architectural

differencing by taking two architectures—each representing one (version of a) product—and automatically calculating the difference in terms of additions and removals of elements (elements being the entities defined in the xADL 2.0 *Structure and Types* schema: components, component types, connectors, connector types, interface types, and opaque links among the interfaces on the components and connectors). The second algorithm complements the first by calculating which additions and removals constitute replacements, thereby enhancing the level of understanding an architect may gain from using our approach. Below, we first discuss the schema and then introduce each of the algorithms.

4.1 XML schema for representing architectural changes

Figure 1 presents the XML schema we developed for capturing architectural change. The schema as shown is somewhat condensed in that commentary and some XML namespace details are left out for brevity. The schema is straightforward in simply defining an architectural "diff" as a series of additions and removals of architectural elements. Nonetheless, three important observations are in place about the design of the schema. First, it should be noted that the schema is based on the *Structure and Types* schema of xADL 2.0. In particular, each addition of an element contains the full definition of the element as specified in the *Structure and Types* schema. This has two distinct advantages.

1. The original architectural specification does not have to be present if a merge is being performed with another architecture. All necessary data is contained in the diff.

2. The differencing and merging algorithms do not have to be reimplemented with each change in the XML schemas. The nature of our XML tool support [7] is such that any additional information as specified in extension schemas is automatically included in the diff. Thus, if a particular extension has augmented a component type with, for example, a mapping to source code, our implementations of the differencing and merging algorithms automatically incorporate that information in the diff (see also Section 6).

The second observation pertains to the fact that removals are specified in terms of identifiers. An important design consideration in xADL 2.0 is that each and every element has a unique identifier and that any change to an element will result in that element having a new, once again unique, identifier. This allows the removal of an element to simply be based on these

identifiers, since two elements with the same identifier are guaranteed to be the same element and any two elements that have a different identifier are guaranteed to be different elements. This holds true even if two elements are in different architectural specifications.

The final observation is that the schema does not directly incorporate replacements. Traditional difference formats are three-tiered and distinguish additions, removals, and replacements [3]. In our approach, we decided upon a two-tiered approach for simplicity reasons. Leaving out replacements keeps the schema, differencing algorithm, and merging algorithm straightforward and allows separate treatment of the more difficult replacement problem (see Section 4.3).

```
<xsd:schema
   xmlns="http://www.ics.uci.edu/pub/arch/xArch/diff.xsd">

    <xsd:element name="diff" type="Diff"/>

    <xsd:complexType name="Add">
        <xsd:choice>
            <xsd:element name="component"
               type="types:Component"/>
            <xsd:element name="connector"
               type="types:Connector"/>
            <xsd:element name="link"
               type="types:Link"/>
            <xsd:element name="componentType"
               type="types:ComponentType"/>
            <xsd:element name="connectorType"
               type="types:ConnectorType"/>
            <xsd:element name="interfaceType"
               type="types:InterfaceType"/>
        </xsd:choice>
    </xsd:complexType>

    <xsd:complexType name="Remove">
        <xsd:attribute name="removeId"
           type="archinstance:Identifier"/>
    </xsd:complexType>

    <xsd:complexType name="DiffPart">
        <xsd:choice>
            <xsd:element name="add" type="Add"/>
            <xsd:element name="remove" type="Remove"/>
        </xsd:choice>
    </xsd:complexType>

    <xsd:complexType name="Diff">
        <xsd:sequence>
            <xsd:element name="diffPart" type="DiffPart"
                         minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Figure 1. XML Schema for Representing Architectural Changes.

4.2 Architectural differencing

Based upon the XML schema defined in the previous section, we have defined a differencing algorithm that takes as its input two product architectures and creates as its output an XML diff file adhering to the XML schema. *Figure 2* illustrates this algorithm. Because a diff file only contains additions and removals, and because xADL 2.0 uses unique identifiers for all of its elements, the differencing algorithm is relatively straightforward. The basic algorithm iterates over each element (component, connector, link, component type, connector type, and interface type) in the first architecture, verifies via identifier comparison whether the element exists in the second architecture, and if not, adds the element to the diff. If the element does exist in the second architecture, the algorithm double checks whether the detailed contents match. If for any reason a mismatch is found (which would be a violation of the xADL 2.0 principles, but nonetheless could inadvertently occur if, for example, someone manually edited a xADL 2.0 document), the algorithm issues a warning and terminates.

After this first phase, it is still necessary to determine superfluous elements in the second architecture. To do so, the algorithm iterates over those elements, verifies whether the element exists in the first architecture, and in case of absence adds instructions in the diff to remove the element. If the element does exist in the first architecture, nothing needs to be done.

The result of applying the algorithm is a diff adhering to the schema defined in the previous section. This diff can be viewed with any standard XML viewer to gain an understanding of what changed between two architectures. Of note is that the algorithm operates solely in terms of the *Structure and Types* schema and does not worry about the selection of particular products out of a product line architecture (i.e., it does not interpret elements that are specified according to the *Options*, *Variants*, and *Versions* schemas). We consider this selection a separate problem, and are developing separate tools that take as input a set of selection properties and produce as output the specific product architecture as abstracted out of a product line architecture. This results in a two-phased approach, which is appropriate since both selection and differencing are algorithms that can be reused in many different places. To understand the differences between two products of a product line architecture, then, one first selects two (versions of) products by applying the selection algorithm twice. Only then, the differencing algorithm can be applied.

Note that the differencing algorithm is inherently architecture-based. Whereas a line-based or XML-based differencing algorithm would have identified small-grained elements that may have changed (e.g., individual lines in an architectural description or XML tags or elements), our algorithm

results in a diff that naturally operates in terms of architectural elements such as components and connectors.

```
elementListOne ← All elements in architecture one
elementListTwo ← All elements in architecture two
diff ← empty

/*
Inspect all elements in the first structure and compare them
to elements in the second structure to check for additions.
*/
for each element1 in elementListOne
{
   if an element in elementListTwo has an ID matching
      element1.getID()
   {
      element2 ← element with matching ID in elementListTwo
      if (!element1.isEquivalent(element2))
      {
         /*
         element1 and element2 have same ID but not
         identical internally so are different.
         */
         ...output warning & terminate...
      }
   }
   else
   {
      diff.addAdd(element1)
   }
}

/*
Check second list for removals.
*/
for each element2 in elementListTwo
{
   if no element in elementListOne has an ID matching
      element2.getID()
   {
      /*
      element2 is an obsolete element.
      */
      diff.addRemove(element2)
   }
}
```

Figure 2. Architectural Differencing Algorithm.

4.3 Replacement

Simply presenting an architect with the set of elements that have been added and removed is not always sufficient to fully understand the difference between two architectures. Often, certain sets of changes are related in that some set of elements was removed and substituted, in place, by another set of elements. To an architect, it is important to gain insight in such

replacements because they represent a higher-level concept than a simple list of additions and removals.

We have defined a replacement detection algorithm that, given a target architecture (e.g., architecture 2 in *Figure 2*) and a diff, calculates replacement sets. Figure 3 illustrates this algorithm, which operates by searching for differing sets of elements that each are entirely surrounded by the same set of common elements. Based on the realization that, upon replacement of a component or connector, an old link must have been broken and replaced by a new link, the algorithm uses links from common elements to find potential starting points for replacements. Once such a starting point has been found, the algorithm grows the set of elements included in a replacement group by step-by-step examining whether a common element is found (indicating a boundary) or whether another replaced element is found. In doing so, the algorithm not only finds one-on-one replacements, but also replacements of the nature in which some number of elements is replaced by some other number of elements (e.g., 2 components are replaced by 3 components and a connector).

```
elementCollection ← All elements in new architecture
diffCollection ← All diff elements
commonElements ← All common components and connectors
closedList ← empty
openList ← empty
replacedGroups ← Groups of elements that are replaced
replacedWithGroups ← Group of elements that replace

for each element in commonElements
{
   if (element not in closedList)
   {
      closedList.add(element)
      for each curLink connecting to an interface of element
      {
         if (curLink not in closedList)
         {
            newLink ← link connecting to this interface
               in target architecture
            if (newLink present & different from curLink)
            {
               /* replacement beginnings found */
               closedList.add(curLink)
               closedList.add(newLink)
               rGroup ← new group representing a group
                  of replaced elements
               rwGroup ← new group representing a group
                  of replacing elements
               rGroup.add(curLink)
               rwGroup.add(newLink)
               oppositeElement ← element opposite curLink
                  in current architecture
               newOppositeElement ← element opposite
                  newLink in new architecture
               if (oppositeElement not in commonElements)
```

```
                    rGroup.add(oppositeElement)
                if (newOppositeElement not in commonElements)
                    rwGroup.add(newOppositeElement)
                addLinksToOpenList(oppositeElement)
                addLinksToOpenList(newOppositeElement)

                /* Grow replacement group */
                while (openList not empty)
                {
                    link ← link popped off the openList
                    closedList.add(link)
                    if (link in new architecture)
                        rGroup.add(link)
                    else
                        rwGroup.add(link)

                    oppElement ← element on the opposite side
                        of this link
                    if (oppElement not part of the
                        commonElements collection)
                    {
                        addLinksToOpenList(oppElement)
                        if (oppElement in new architecture)
                            rGroup.add(oppElement)
                        else
                            rwGroup.add(oppElement)
                    }
                }
                replacedGroup.add(rGroup)
                replacedWithGroup.add(rwGroup)
            }
        }
    }
  }
}
```

Figure 3. Replacement Detection Algorithm.

5. PROPAGATION

To complement the differencing algorithm used for understanding architectural changes, we have defined a merging algorithm that is capable of propagating, to a third architecture in the product line architecture, the changes captured in a diff. Given a diff and a target architecture, this algorithm constructs a new architecture that is the result of merging the diff with the target architecture. The algorithm is shown in *Figure 4* and iterates over each of the elements in the diff and adds those elements that must be added and removes those elements that must be removed. Error handling (not shown) simply abandons the algorithm in case a conflict occurs (e.g., a non-existing element must be removed, an element must be added that already exists, a link must be added to a non-existing element).

Of note is the simplicity of the merging algorithm. Because replacement was relegated to a separate algorithm rather than incorporated in the diff

representation, an architectural diff is simply stated in terms of additions and removals and the merging algorithm only has to follow those instructions step-by-step.

Critical in the functioning of the merging algorithm is the presence of some common elements among the original two product architectures that were used to construct the diff, and the third architecture upon which the diff is applied. Without at least a few common elements, the merge algorithm would clearly not be able to function. Fortunately, the very nature of product line architectures is such that most, if not all, of its products share a set of common architectural elements that form the core of the product line. While we, thus, believe our algorithms to be effective within the domain of product line architectures, their application to random, non-related architectures may not give as good results.

```
elementCollection ← All elements in target architecture
diffCollection ← All diff elements

for each diffElement in diffCollection
{
   if (diffElement is an Add)
   {
      addElement ← element in diffElement
      if (!elementCollection.hasElement(addElement))
      {
         architecture.addElement(addElement)
      }
   }
   else if (diffElement is a Remove)
   {
      removeElement ← element in diffElement
      if (elementCollection.hasElement(removeElement))
      {
         architecture.removeElement(removeElement)
      }
   }
}
```

Figure 4. Architectural Merging Algorithm.

6. IMPLEMENTATION

We have created the XML diff schema as part of xADL 2.0 and implemented the algorithms in ArchDiff, a new component in the ArchStudio environment for architecture-based software development [14]. Thus far, we have used ArchDiff extensively on several small examples and briefly on one larger demonstration project. This project involved a real-life architecture description consisting of hundreds of components and

connectors. Our algorithms and tools scaled to support an architecture description of this size, and returned results generally within several seconds and at most within minutes. The nature of this delay does not lay in the algorithms, which are of order $O(n^2)$, but in the implementation which needs to perform a large amount of XML processing and is not further optimized in any which way. We expect a future version of the tool to be optimized and to reduce the differencing and merging time significantly.

Currently, ArchDiff is entirely text-based with a simple command-line interface. We plan for future versions of the tool to incorporate graphical views that illustrate the process taking place and allow users to manually resolve conflicts as they occur.

ArchDiff is build upon the xADL 2.0 data binding library [7] that is automatically generated by a tool called ApiGen [6]. The data binding library preserves any information attached by extension schemas (such as source code mappings or versioning metadata) to lower-level xADL 2.0 elements (such as components and component types). ArchDiff, though implemented to operate at the level of components and component types, therefore supports extensions and does not have to be reimplemented when new modeling concepts are added to xADL 2.0.

7. CONCLUSION

This paper makes a small but important contribution to the field of product line architecture in developing algorithms that can be used for understanding architectural changes and propagating those changes among individual architectures in the product line. The strength of the algorithms lies in their use of a simple, XML-based representation for capturing architectural changes. Different algorithms build upon this representation to determine not only those architectural elements that have been added or removed, but also those sets of elements that represent replacements within the architecture.

ArchDiff represents only the beginnings of our work in this area. The creation of a graphical user interface is clearly at the forefront of our further development efforts. However, we also intend to address some more fundamental research questions as part of our future work. Most notably, we intend to investigate how the differencing and merging algorithms can be adapted to support dynamic, run-time updates [13,16]. Given that xADL 2.0 supports attaching implementation information (such as Java class files) to architectural elements, we intend to leverage the above results in developing a tool that “merges” architectural changes into a running system by removing, instantiating, and linking elements dynamically. Additionally, we

intend to investigate the use and applicability of more fine-grained, semantic-based differencing and merging algorithms to further support the management of architectural change.

ACKNOWLEDGMENTS

The authors would like to thank Eric Dashofy for his valuable contributions to the described research.

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. Effort also partially funded by the National Science Foundation under grant number CCR-0093489. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Laboratory, or the U.S. Government.

REFERENCES

[1] Allen, L., Fernandez, G., Kane, K., Leblang, D., Minard, D., and Posner, J. ClearCase MultiSite: Supporting Geographically-Distributed Software Development. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers.* p. 194-214, Springer-Verlag, 1995.

[2] Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. ed. Wesley, A. 2000.

[3] Buffenbarger, J. Syntactic Software Merging. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*. p. 153-172, Springer-Verlag, 1995.

[4] Conradi, R. and Westfechtel, B. Version Models for Software Configuration Management. *ACM Computing Surveys.* 30(2), p. 232-282, 1998.

[5] Dashofy, E., van der Hoek, A., and Taylor, R.N. A Highly-Extensible, XML-Based Architecture Description Language. In *Proceedings of the The Working IEEE/IFIP Conference on Software Architecture (WICSA 2001).* Amsterdam, The Netherlands, August 28-31, 2001.

[6] Dashofy, E.M. Issues in Generating Data Bindings for an XML Schema-Based Language. In *Proceedings of the Workshop on XML Technologies in Software Engineering.* 2001.

[7] Dashofy, E.M., van der Hoek, A., and Taylor, R.N. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Proceedings of the 24th International Conference on Software Engineering.* p. (to-appear), 2002.

[8] Douglis, F., Ball, T., Chen, Y.-F., and Koutsofios, E. The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web. *World Wide Web.* 1(1), p. 27-44, January, 1998.

[9] Flamsholt, R. *Viff - A Tool for Visual Diffing and Merging.* <http://www.richard.flamsholt.dk/src/viff/viff.html>, Web site accessed on 03/06/2001.

[10] van der Hoek, A., Mikic-Rakic, M., Roshandel, R., and Medvidovic, N. Taming Architectural Evolution. In *Proceedings of the Sixth European Software Engineering Conference (ESEC) and the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9).* p. 1-10, Vienna, Austria, September 10-14, 2001.

[11] IBM. *XML Diff and Merge Tool*. <http://www.alphaworks.ibm.com/-tech/xmldiffmerge>, Web site accessed on 01/17/2002.

[12] INTERSOLV. *Using PVCS for Enterprise Distributed Development.* 1998.

[13] Kramer, J. and Magee, J. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering.* 16(11), p. 1293-1306, 1990.

[14] Medvidovic, N., Oreizy, P., Taylor, R.N., Khare, R., and Guntersdorfer, M. *An Architecture-Centered Approach to Software Environment Integration.* <ftp://www.ics.uci.edu/pub/arch/papers/TR-UCI-ICS-00-11.pdf>, Web site accessed on March.

[15] Medvidovic, N.M. and Taylor, R.N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering.*, p. 70-93, 2000.

[16] Oreizy, P. and Taylor, R.N. On the Role of Software Architectures in Runtime System Reconfiguration. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems.* p. 61-70, IEEE Computer Society Press, 1998.

[17] van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. The Koala Component Model for Consumer Electronics Software. *Computer.* 33(3), p. 78-85, 2000.

[18] Zündorf, A., Wadsack, J.P., and Rockel, I. Merging Graph-Like Object Structures. In *Proceedings of the Tenth International Workshop on Software Configuration Management.* 2001.