Differencing and Merging within an Evolving Product Line Architecture

P. Chen, M. Critchlow, A. Garg, C. Van der Westhuizen, A. van der Hoek

Department of Informatics
School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{pchen,critchlm,agarg,cvanderw,andre}@ics.uci.edu

Abstract. Propagating changes from one place in a product line architecture (PLA) to another is a difficult problem that occurs in a variety of settings. Currently, no automated tools exist that help an architect in doing so, and performing the task by hand in the face of a large PLA can be error-prone and difficult. To address this problem, we have built a set of tools for automating the process. Our approach breaks down into a two-step solution: (1) automatically determining the difference between two selected (versions of) a product architecture, and (2) automatically merging the difference back into a different location in the original PLA. In this paper, we detail each of these two steps and evaluate our solution on an example word processor PLA.

1 Introduction

Consider the following three scenarios that may occur in architecting a product line architecture (PLA):

1. A series of changes have been made to one particular variant of a subsystem. After the changes have been made, it turns out they are useful in some of the other variants as well. The architect now wishes to take the changes and apply them to those other variants [2].
2. In order to experiment with a new piece of functionality without interfering with the main line of development, the architect creates a new branch and implements the functionality on this branch first. It turns out that the functionality can be incorporated, and the architect now wishes to move the new functionality back into the main line of development [1].
3. To ensure accurate and responsive customer service, company policy requires a product architecture to be maintained independently from the main PLA after it has been deployed to a customer. During maintenance, however, the architect makes some changes to the individual product architecture that would benefit the overall PLA if they could be incorporated. The architect now wishes to move those changes back into the PLA [4].

A common concern throughout these scenarios is the need for the architect to be able to propagate a set of changes within the realm of a single PLA, e.g., from one place in the product PLA to another. Although it is possible to do this by hand, it may not always be feasible (or desirable) when considering a PLA consisting of many inter-related products that each comprise a large set of components. In such cases, manually propagating changes is highly error-prone and labor-intensive.

In this paper, we describe our solution to this problem. Our approach centers on the use of differencing and merging techniques that are specific to PLAs. In particular, we have designed and implemented two automated algorithms: (1) a differencing algorithm for determining the set of changes between two (versions of) a product architecture selected out of an overall PLA, and (2) a merging algorithm with which such changes are merged back into a different location in the PLA. The algorithms complement Ménage, our design environment for managing the evolution of PLAs [8], and enhance it with automated support for propagating changes throughout an evolving PLA.

The remainder of this paper is organized as follows. Section 2 discusses our overall approach. Section 3 defines our representation for capturing an architectural difference and presents our differencing algorithm in detail. Section 4 describes the merging algorithm, and illustrates how it can be used to propagate changes throughout a PLA. Section 5 discusses our experience in using the algorithms. We conclude by briefly discussing related work in Section 6 and discussing future work in Section 7.

2 Approach

Figure 1 illustrates our overall approach based on the third scenario discussed in the introduction. First, a PLA is precisely defined (for instance, by using Ménage [8]). This results in a PLA specification from which individual product architectures can then be selected for delivery to customers (PA version 1). Based on customer feedback, PA version 1 is changed and evolves into version 2, which is, once again delivered to the customer. The changes turn out to be useful to some parts of the original PLA as well, and now must be propagated into its specification. To do so, the architect first uses the differencing algorithm to precisely determine the changes that were made between PA versions 1 and 2. This results in an architectural “diff” file, which can then be merged back into the product line in a specified location, resulting in a new PLA that contains the new functionality.

We must make two observations regarding the process presented in Figure 1. First, similar processes can be defined for the other scenarios discussed in the introduction. Although the top line of process steps and specifications will be different, the same differencing and merging algorithms will be used in exactly the same way to propagate desired changes within the original PLA. The second observation regards the generality of the process: the algorithms are not limited to differencing and merging individual product architectures in which all variation points have been resolved. Instead, they fully support operation over specifications that still may contain “open” variation points.

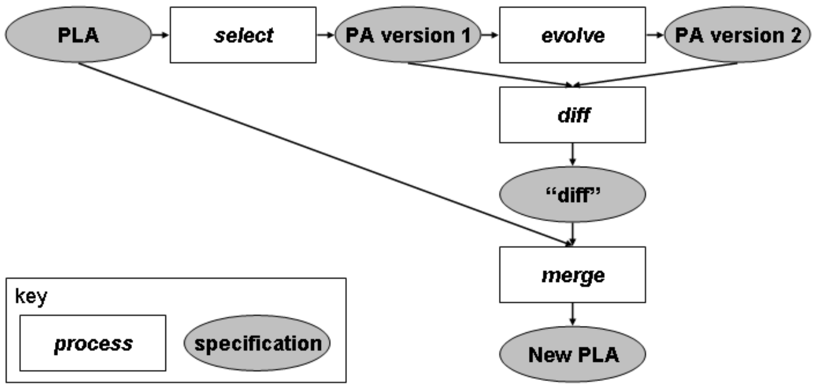


Figure 1. Overall Process for Scenario 3.

In our previous work, we developed differencing and merging algorithms that operate on single software architectures [11]. These algorithms lay the basis for the algorithms we discuss here, but they also exhibit a number of shortcomings that make them unsuited for the context of PLAs. In particular, they rely on identifiers to determine commonality; they do not handle type differences; they do not address hierarchical substructures; and they only operate at the level of components and connectors. In contrast, the algorithms we define in this paper adhere to the following objectives:

- *The algorithms do not rely on identifiers to establish commonality.* Different versions of a PLA will have parts that can be comprised of the same elements, but those elements must have different identifiers since they can evolve separately. As a result, our algorithms must base similarity on both the type and structure of elements.
- *The algorithms integrally address hierarchical substructure.* Virtually every PLA is composed in a hierarchical fashion; our algorithms should be able to handle differences at each level.
- *The algorithms are fine-grained.* In particular, the algorithms do not operate at the level of components as a whole, but are able to express subtle differences in component interfaces, differences in the way links connect components and connectors, and differences in mappings from high-level interfaces to interfaces in a substructure.

Overall, then, we take a highly semantic approach. Rather than applying semantically-neutral differencing and merging algorithms [10], we wish to be able to operate and view changes in terms of architectural elements. Only then can an architect easily understand the context of the changes and apply the algorithms without having to make complex mental translations from changed lines of text to the actual architectural elements that they describe.

3 Differencing Product Line Architectures

The first step in our approach is to determine exactly which changes must be propagated. For this purpose, we constructed a representation for capturing the changes and implemented an algorithm for helping architects in automatically calculating the changes as a difference between two particular selections out of a PLA. Below, we discuss each in detail.

3.1 Difference Representation

Our existing work relies on the xADL 2.0 framework and infrastructure for representing and accessing PLAs [5]. We have chosen to design our representation for differences as an extension of the framework. In particular, we have extended the set of XML schemas that define the xADL 2.0 language with a new schema for capturing architectural differences. Figure 2 presents the structure of this schema. The schema is defined as a recursive hierarchy of DIFFPART elements. Each DIFFPART corresponds to a level in the architecture and describes the set of changes to that level as a set of additions of new elements, a set of removals of existing elements, and (as a series of lower-level DIFFPART elements) a set of changed elements.

Most of the representation is fairly straightforward, but we make the following observations regarding the schema. First, we note that additions are either structural in nature or type-based. Structural changes represent changes to the structure of a component or connector type and determine the hierarchical composition of the type. For instance, the difference in the structure of a component type could state that it now includes an additional component and connector instance, as hooked up to the remainder of the internal elements of the original component type. Type-based differences concern the definition of a component or connector type, and include changes in its signatures (definitions of which functionality it provides), changes in its variation points (addition of new and removal of old variants), and changes in the mapping from its signatures to its internal component and connector instances. By separating structural changes from type-based changes, and capturing those changes in terms of detailed modifications, our representation achieves a fine-grained level of expressiveness that is semantically in sync with the remaining family of xADL 2.0 schemas to which it belongs.

The second observation pertains to the fact that each DIFFPART includes a DIFFLOCATION to precisely identify the component or connector (type) to which the associated differences belong. This allows the representation to capture hierarchical changes, since each DIFFLOCATION identifies exactly which hierarchical part of a PLA it addresses. Note that a DIFFLOCATION is relative to the DIFFPART in which it is contained: this avoids having to use globally unique identifiers when differencing and merging PLAs.

Finally, we observe that our representation supports the specification of changes in variation points. For instance, the “optional” that can be added or removed represents an optional element (with associated guard) in the PLA. Similarly, a “variant” represents a changing alternative in the PLA (again, with associated guard) [8].

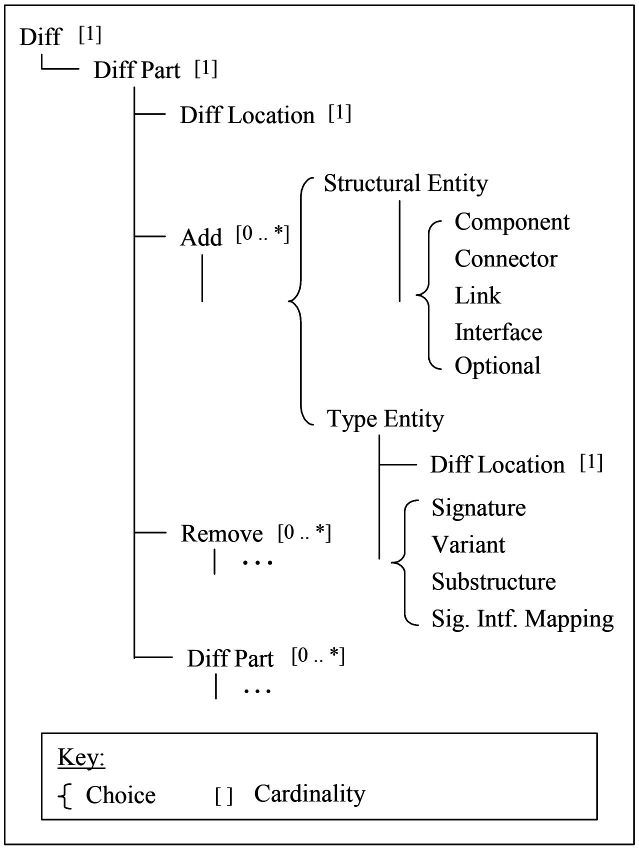
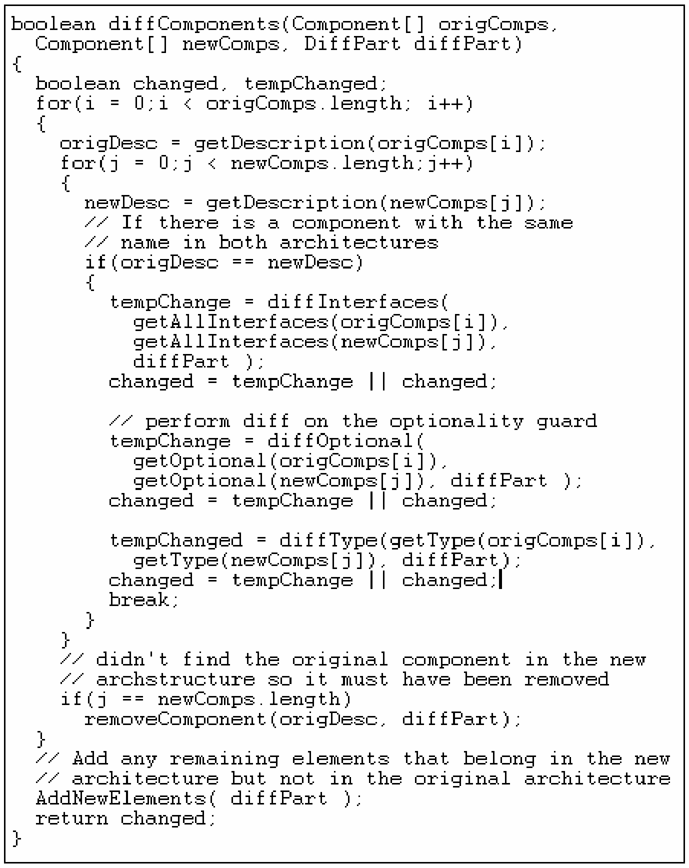


Figure 2. Difference Representation.

3.2 Differencing Algorithm

A significantly simplified version of part of our differencing algorithm is presented in Figure 3. The algorithm takes as input the xADL 2.0 specifications of two product architectures that each may or may not have any variation points left (in essence, thus, the algorithm is able to difference complete PLAs, although that is typically not a very useful operation). In addition, it takes as input a starting point for each specification. The starting point is necessary since it should be possible to generate the difference between parts of a product architecture, since one does not always want to have the complete difference. For instance, the feature that has to be propagated back into the PLA may be at a lower level in the architecture and some unwanted changes may have been made at a higher level in that same architecture. The starting point allows an architect to hone in on just the feature they want to propagate and ignore any of the other changes.



```
boolean diffComponents(Component[] origComps,
  Component[] newComps, DiffPart diffPart)
{
  boolean changed, tempChanged;
  for(i = 0;i < origComps.length; i++)
  {
    origDesc = getDescription(origComps[i]);
    for(j = 0;j < newComps.length;j++)
    {
      newDesc = getDescription(newComps[j]);
      // If there is a component with the same
      // name in both architectures
      if(origDesc == newDesc)
      {
        tempChange = diffInterfaces(
          getAllInterfaces(origComps[i]),
          getAllInterfaces(newComps[j]),
          diffPart );
        changed = tempChange || changed;

        // perform diff on the optionality guard
        tempChange = diffOptional(
          getOptional(origComps[i]),
          getOptional(newComps[j]), diffPart );
        changed = tempChange || changed;

        tempChanged = diffType(getType(origComps[i]),
          getType(newComps[j]), diffPart);
        changed = tempChange || changed;
        break;
      }
    }
    // didn't find the original component in the new
    // archstructure so it must have been removed
    if(j == newComps.length)
      removeComponent(origDesc, diffPart);
  }
  // Add any remaining elements that belong in the new
  // architecture but not in the original architecture
  AddNewElements( diffPart );
  return changed;
}
```

Figure 3. Parts of Differencing Algorithm.

Note that a starting point is needed for both the original product architecture and the new product architecture. As part of the evolution of the product architecture, certain components may have been moved up or down to a different level. To be able to directly compare those elements a starting point is needed in each of the product architectures.

The difference algorithm operates by recursively iterating through all elements in the original architecture that are reachable from the starting point and comparing them with their corresponding elements in the new architecture. If an element does not exist in the new architecture, the element was removed and the diff will contain a remove instruction. Conversely, if an element exists in the new architecture that does not exist in the old architecture, an addition is entered into the diff. When an element exists in both the old and the new architecture, the algorithm compares the details of the elements to achieve a fine-grained diff. For example, when a component instance exists in both the old and new document, the difference algorithm verifies whether its interfaces are the same, whether or not its optionality is the same (and, if optional, whether it has the same Boolean guards governing its optionality), and whether it is of the same type. Only when all of these are exactly the same, the algorithm concludes that the component instance has not changed; otherwise, instructions are placed in the diff to account for the difference. The algorithm performs similar comparisons for all other elements.

3.3 Differencing Example

Figure 4 introduces a simple example word processor PLA that contains an optional component PRINT, a variant component SPELL CHECKER, and an optional variant component GRAMMAR CHECKER. The component USER INTERFACE also has a substructure that includes the components TOOLBAR and DISPLAY. For simplicity and space reasons, the example ignores many details, such as, among others, connectors, interfaces, signatures, and various mappings of interfaces. It is noted, however, that the overall approach remains exactly the same.

In staying with scenario 3 as introduced in Section 1, imagine that the French product architecture is selected as shown in Figure 5. The product architecture includes the optional component PRINT, but does not include a GRAMMAR CHECKER. Based upon customer request, the product architecture is modified to include functionality for viewing pictures; this required the addition of two new components: PICTURE DECODER and PICTURE DISPLAY (Figure 5). It is determined that it is useful to include this functionality in the other variants of the product architecture, necessitating a need to propagate the changes to the PLA shown in Figure 4. As a first step in this process, the architect uses our implementation of the differencing algorithm, resulting in a diff file. The diff file contains a root DIFFPART that contains two parts: the addition of the component PICTURE DECODER and another DIFFPART, which contains the addition of the component PICTURE DISPLAY to the component USER INTERFACE.

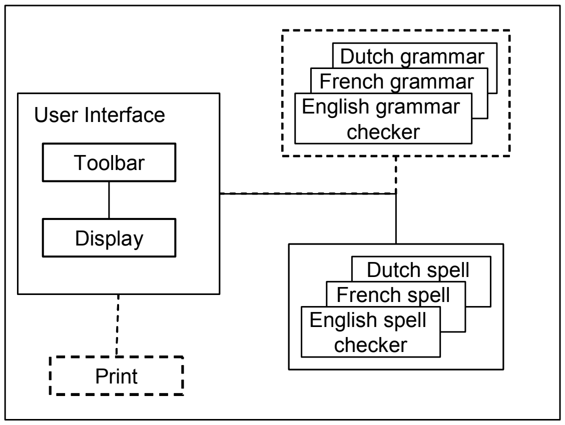


Figure 4. Example PLA.

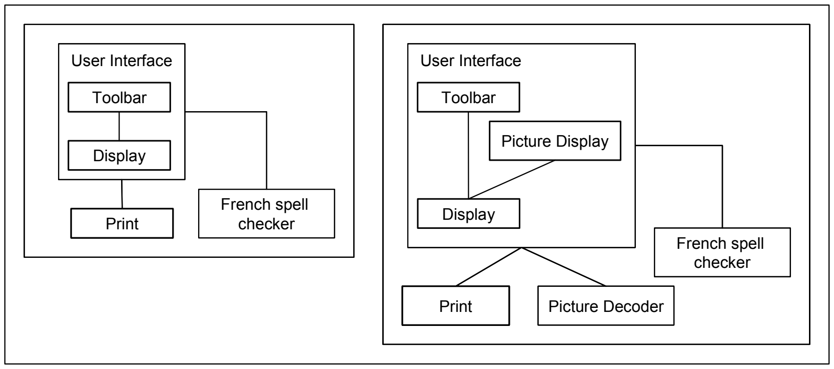


Figure 5. Original and Evolved French Word Processor.

4 Merging

Complementing the differencing algorithm is our merging algorithm. The merging algorithm propagates a set of changes as captured in a diff file into a particular part of a target PLA. As such, it takes the series of individual changes contained in the diff file and applies each instruction in turn. The result, assuming the merge is successful, is a new PLA in which the new functionality is now available.

Similar to the differencing algorithm, the merging algorithm takes as input a starting point that describes where in the target PLA the differences must be applied. Again, this allows selective merging into specific parts of the product line, as to not disturb other parts.

Parts of the merging algorithm are shown in Figure 6. It operates by iterating through the hierarchal instructions of a diff document and, at each step, adding all elements that need to be added and removing all elements that need to be removed. The algorithm does so by alternating over structural changes and type-level changes. This is inherent to the underlying xADL 2.0 representation for PLAs. Since each structural element is typed, the algorithm ensures that it updates all necessary types when changes to a structural layer have been completed.

Clearly, due to the fine-grained nature of our algorithm, it is possible for conflicts to occur during the merging of changes. Among many others, it is possible that the algorithm attempts to remove a non-existing element, add an element that already exists, and add to a substructure that no longer exists. In such cases, the algorithm issues a warning message but continues the merging process. Our decision was to make a best-effort attempt rather than simply abandoning merging altogether; often, an architect will be able to correct merge problems relatively easily after the merging algorithm has completed its actions (but see our future work plans in Section 7).

```
void mergeStructure(Structure structure, DiffPart diffPart)
{
  for(int i = 0; i < diffPart.getNumRemoves(); i++)
  {
    Remove remove = diffPart.getRemove(i);
    /* Check for all structural entities to remove such as
    Components, Connectors, Links, Interfaces, Signatures,
    SubArchitectures, Variants and Signature Interface
    Mappings */
    removeElements(structure, remove);
  }
  for(int i = 0; i < diffPart.getNumAdds(); i++)
  {
    Add add = diffPart.getAdd(i);
    /* Check for all structural entities to add such as
    Components, Connectors, Links, Interfaces, Signatures,
    SubArchitectures, Variants and
    SignatureInterfaceMappings */
    addElements(structure, add);
  }
  // For each DiffPart in diffPart merge into the correct
  // substructure or variant
  for(int i = 0; i < diffPart.getNumDiffParts(); i++)
  {
    Object ref =
    getTypeToMergeInto(diffPart.getDiffPartAt(i));
    Structure subArch = ref.getSubStructure();
    if(subArch == null)
      mergeType(ref.getType(), diffPart.getDiffPartAt(i));
    else
      mergeStructure(subArch, diffPart.getDiffPartAt(i));
  }
}

// this also deals with merging into variants
void mergeType(Type type, DiffPart diffPart)
{
  for(int i = 0; i < diffPart.getNumRemoves(); i++)
  {
  Remove remove = diffPart.getRemove(i);
  /* Check for all type entities to remove, such as:
  Signatures, SubArchitectures, Variants and
  SignatureInterfaceMappings */
  removeTypeElements(type, remove);
  }
  for(int i = 0; i < diffPart.getNumAdds(); i++)
  {
  Add add = diffPart.getAdd(i);
  /* Check for all type entities to add, such as:Signatures,
  SubArchitectures, Variants and SignatureInterfaceMappings
  */
  addTypeElements(type, add);
  }
  // For each DiffPart in diffPart merge into the correct
  // substructure or variant
  for(int i = 0; i < diffPart.getNumDiffParts(); i++)
  {
    Object ref =
    getTypeToMergeInto(diffPart.getDiffPartAt(i));
    Structure subArch = ref.getSubStructure();
    if(subArch == null)
      mergeType(ref.getType(), diffPart.getDiffPartAt(i));
    else
      mergeStructure(subArch, diffPart.getDiffPartAt(i));
  }
}
```

Figure 6. Parts of Merge Algorithm.

To exemplify the operation of our merge algorithm, we apply it to the example of Figures 4 and 5. After determining the set of necessary changes using our differencing algorithm and merging those changes back into the original PLA, Figure 7 shows the resulting PLA. We note that the two new components (shown in bold) are available in each of the language variants of the PLA; any product architecture that is now selected will include the components PICTURE DISPLAY and PICTURE DECODER.

Again, we note that the example omits many of the fine-grained details concerning the operation of the algorithm. Space constraints prohibit us to show and discuss all of them. We refer the interested reader to the algorithms themselves, which are publicly available.

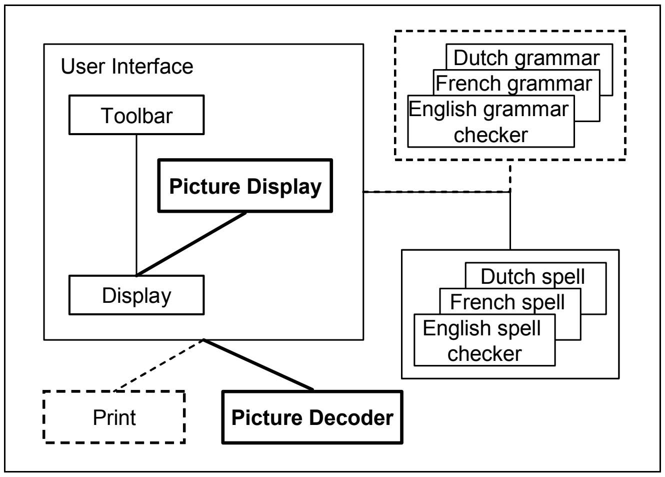


Figure 7. Resulting PLA.

5 Evaluation

We tested the functionality of our algorithms by running them on a PLA for a hypothetical entertainment system. The architecture contains 4 versions of the entertainment system, each consisting of roughly 30 components and connectors with a maximum hierarchical depth of four levels. The architectural layout of the entertainment system is shown in Figure 8 as it appears in Ménage (our architectural evolution environment).

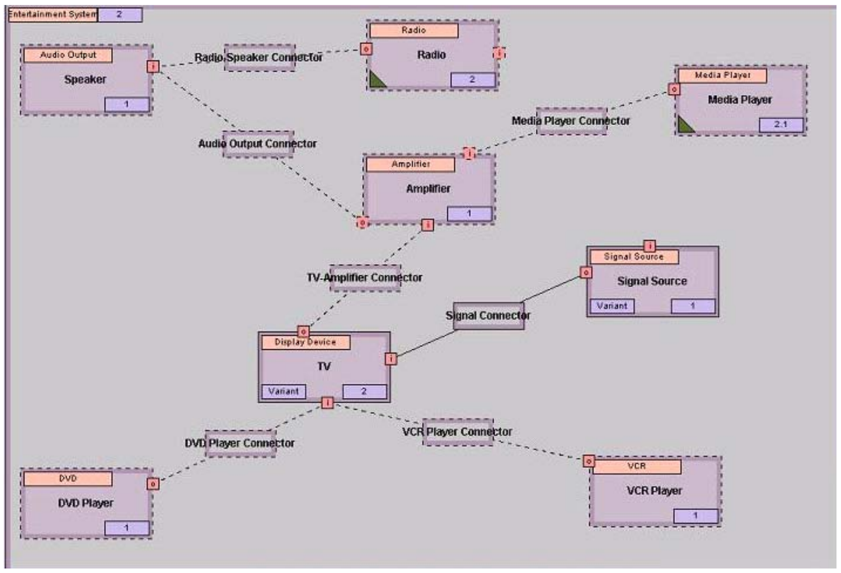
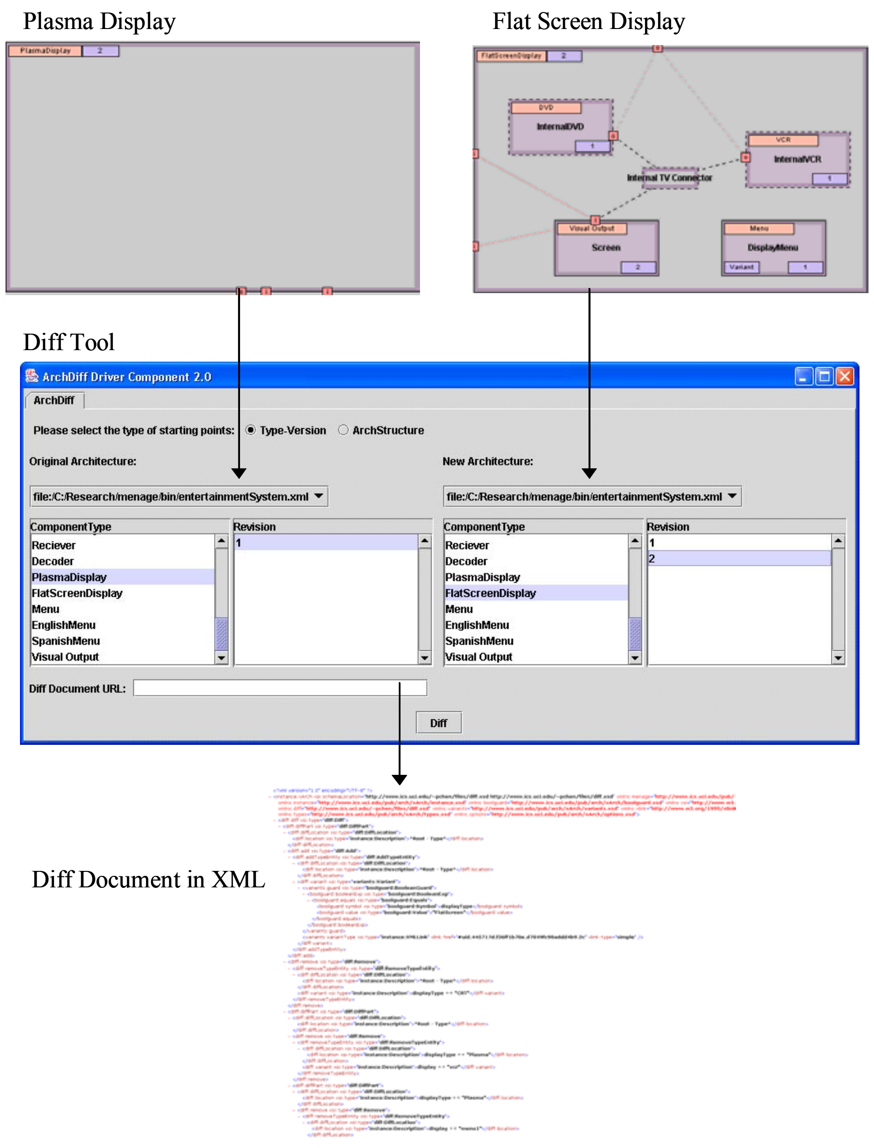


Figure 8. Entertainment System PLA.

Figure 9 illustrates the diff process with the two DISPLAY DEVICE variants. It can be seen that the PLASMA DISPLAY component has no substructure, whereas the FLAT SCREEN DISPLAY component contains an internal structure. The architect can run the differencing tool on the two DISPLAY DEVICE variants ending up with a diff document that describes the differences between the variants. Figure 10 shows the process of merging the diff document (captured changes) into the PLASMA DISPLAY component. The result of this merge is an evolved PLASMA DISPLAY component which now has an internal structure.

This example and our other tests lead to two critical observations regarding our algorithms. First, we note that they are efficient: both the differencing and merging algorithm run in polynomial time and scale up to large PLAs. Second, we observed an interesting effect of the merging algorithm: it is possible for a merge to have hidden side-effects, namely when other parts of the PLA use the same elements that are being changed. While this problem may occur during "traditional" merging of code, it is far more likely to occur in PLAs because of the high levels of reuse and sharing of types. We intend to upgrade our algorithms to issue a warning when this occurs, and intend to update our design environment (Ménage) with a process that will check out relevant elements such that side effects are avoided and impact is limited to individual branches only.



Please select the type of starting points: [✓ Type-Version] ArchStructure

Original Architecture: file:/C:/Research/menage/bin/entertainmentSystem.xml

ComponentType	Revision
Reciever	1
Decoder	
PlasmaDisplay	
FlatScreenDisplay	
Menu	
EnglishMenu	
SpanishMenu	
Visual Output	

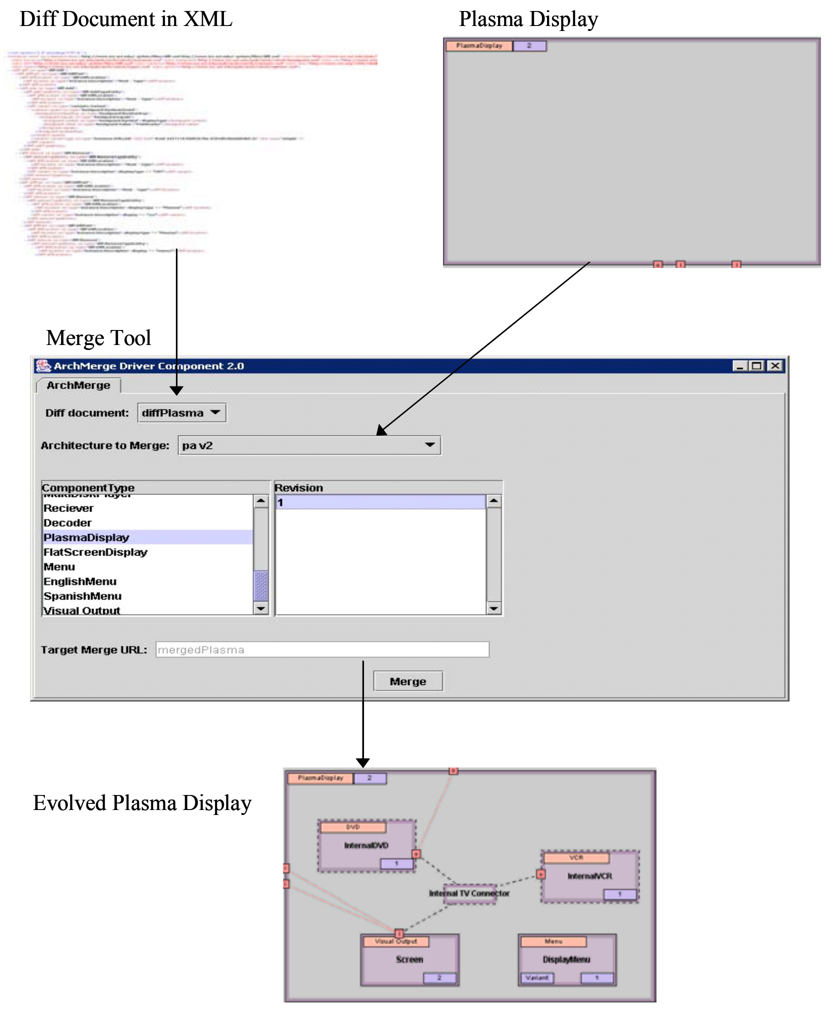
New Architecture: file:/C:/Research/menage/bin/entertainmentSystem.xml

ComponentType	Revision
Reciever	1
Decoder	2
PlasmaDisplay	
FlatScreenDisplay	
Menu	
EnglishMenu	
SpanishMenu	
Visual Output	

Diff Document URL:

[illegible]

Figure 9. Diff Process.



Diff document: diffPlasma

Architecture to Merge: pa v2

Target Merge URL: mergedPlasma

Figure 10. Merge Process.

6 Related Work

Our approach differs from other methods of differencing and merging [3,6,9,10] in that it is semantic in nature. Our difference representation and algorithms are specifically designed to deal with PLAs, and as such give us the distinct benefit of being able to difference and merge hierarchical structures at a fine-grained level. Other approaches are syntactic in nature and typically operate on text files [10]. At best, some semantics are built in, such as the differencing and merging tools for HTML [6] and XML [9]. Even though our representation is a xADL 2.0 XML Schema, neither of those approaches would work since it would not recognize related changes in different parts of the document. We have, therefore, chosen for a less generic approach that is more powerful and applicable to our situation (much like other approaches in which semantics is important [7]).

7 Conclusion & Future Work

The differencing and merging algorithms presented in this paper can accurately capture changes between different product architectures and propagate them back into the originating PLA. This provides automated support for deriving new product architectures that incorporate these changes—a property not present in previous work. The strength of the algorithms lies in their ability to handle, at a semantic level, fine-grained differencing and merging of hierarchically composed PLAs.

The approach presented in this paper makes a positive contribution to the management of evolving PLAs; however, it is clear that further research is necessary to explore related areas. While our approach allows for differencing and merging within PLAs, the issue of differencing and merging across different PLAs remains to be addressed. Moreover, we wish to create tools to visualize the process of differencing and merging to provide architects with support for detecting and resolving conflicts swiftly. Finally, we observe that our algorithms provide a particular form of architectural refactoring support. We intend to further explore this area in hopes of devising other, more advanced algorithms for this purpose.

Availability

Implementations of our algorithms can be downloaded from:

http://www.isr.uci.edu/projects/archstudio/

Acknowledgements

The authors thank Eric Dashofy for his valuable contributions to the development of this project. Effort funded by the National Science Foundation under grant numbers CCR-0093489 and IIS-0205724.

References

[1] T. Asikainen, T. Soininen, and T. Männistö. *Towards Intelligent Support for Managing Evolution of Configurable Software Product Families*. Proceedings of the ICSE Workshops SCM 2001 and SCM 2003 Selected Papers, 2003: p. 86-101.

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

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

[4] P. Clements and L.M. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, New York, New York, 2002.

[5] E.M. Dashofy, A. van der Hoek, and R.N. Taylor. *An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages*. Proceedings of the 24th International Conference on Software Engineering, 2002: p. 266-276.

[6] F. Douglis, et al., *The AT&T Internet Difference Engine: Tracking and Viewing Changes on the Web*. World Wide Web Journal, 1998. 1(1): p. 27-44.

[7] J. Estublier. *Defining and Supporting Concurrent Engineering Policies in SCM*. Proceedings of the Tenth International Workshop on Software Configuration Management, 2001.

[8] A. Garg, et al. *An Environment for Managing Evolving Product Line Architectures*. Proceedings of the International Conference on Software Maintenance, 2003.

[9] IBM, *XML Diff and Merge Tool*, http://www.alphaworks.ibm.com/tech/xmldiffmerge, 2002.

[10] T. Mens, *A State-of-the-Art Survey on Software Merging*. IEEE Transactions on Software Engineering, 2002. 28(5): p. 449-462.

[11] C. Van der Westhuizen and A. van der Hoek. *Understanding and Propagating Architectural Changes*. Proceedings of the Working IFIP Conference on Software Architecture, 2002: p. 95-109.