Extensible Language-Aware Merging

James J. Hunt
Forschungszentrum Informatik
Karlsruhe, Germany
jjh@fzi.de

Walter F. Tichy
University of Karlsruhe
Karlsruhe, Germany
tichy@ira.uka.de

Abstract

Parallel development has become standard practice in software development and maintenance. Though most every revision control and configuration management system provides some form of merging for combining changes made in parallel, these mechanisms often yield unsatisfactory results. The authors present a new merging algorithm, that uses a fast differencing algorithm and renaming analysis to provide better merge results. The system is language aware, but not language dependent and does not require a special editor, so it can be easily integrated in current development environments.

1. Introduction

Modern software development requires the coordination of new development with ongoing system maintenance, so that corrections to released code are incorporated in following releases. Revision control and configuration management systems provide facilities managing parallel work on the current development line and maintenance of previous releases. Current systems do provide mechanisms for combining changes made in parallel, but these mechanisms for merging variants of a software document often yield unsatisfactory results.

Merging may seem quite straight forward. In fact, even early revision control systems, like RCS[14], incorporate a feature for merging revisions from different branches of development. Unfortunately, this and similar tools do not perform as well as one would hope. The traditional solution uses a line based comparison because it can be used with most text formats and it is relatively fast, but the quality of the results is quite low. Other solutions have been proposed, but they have failed to gain acceptance.

The standard practice has remained with the line based tools. The main difficulty for differencing and merging is making a good trade-off between accuracy, applicability, and performance. To this end, the authors have developed **Extensible, Language-Aware Merging**, or **ELAM** a new algorithm for producing an integrated revision by combining two variants of a common base revision.

2. The Problem

Merging two variants of a software document means producing a new version of the document which incorporates all changes made in both variants of the document. Three way merging uses a common base version to ascertain exactly what changes were made. The chief problem of merging is to determine which changes can be integrated automatically and which changes can not.

The notion of conflict is used to make this determination. Most systems use the notion of collocation to define conflicts. If two changes occur at the same point in a program, they conflict. A more exact definition says that two changes conflict when the code in one change can influence the result of the code in the other change, i.e., they exhibit semantic interference.

These two definitions are not equivalent. Many collocated changes have no influence on one another. Also changes resulting in non-local conflicts, like those resulting from changes to a method or procedure signature and renaming, are not collocated. Though **ELAM**, as will be seen, uses collocation for identifying most conflicts, the semantic interference criteria is use to define conflicts. A collocation conflict that do not exhibit semantic interference is termed an apparent conflict.

Since general conflict resolution is an intractable problem, the design of **ELAM** places the emphasis on improving conflict detection and isolation. Semantic knowledge about the programs in question provides the basis for distinguishing between apparent conflicts, which result from non-interfering changes made at the same position in both variants, and actual semantic conflicts. The resolution of apparent conflicts is heuristic, so the system can not guarantee that all apparent conflicts will be identified. Nevertheless, simple renaming conflicts are actually resolved as well.

Proceedings of the International Conference on Software Maintenance (ICSM'02)
0-7695-1819-2/02 $17.00 © 2002 IEEE

3. Related Work

A merge can be computed directly from two variants and a common parent or base revision by extending the matrix used by the Longest Common Subsequence (LCS) algorithm[6] into three dimension to accommodate both variants along with the base. In practice, this is not done because the resulting algorithm is rather slow. Three dimensional LCS[7] has a worst case execution of $O(n^3)$ in the length of the base. Furthermore, only local conflicts could be detected. Instead, current research and practice concentrates on three way merging based on pair differencing.

There are two variations to three way merging independent of other consideration: symmetric and asymmetric merging. In the symmetric case, differences are calculated between the base version and each of the variants; however, in the asymmetric case, the second difference is calculated between the two variants instead of the base and the second variant. The advantage of the symmetric method is that the results are independent of the order of the variants; whereas the advantage of the asymmetric method is that like changes, made in both variants, are easier to find.

Aside from the issue of symmetry, the remaining differences in strategy revolve around how the difference information is obtained, what form it has, and how it is processed. Line based algorithms are the simplest and semantic based analysis are the most complex. Between the extremes lie process and structural based approaches.

3.1. Line Based Algorithms

The line based approach for merging was the first to be devised and is still the most widely used. The best example is Unix *diff3*[11], which is still in widespread use today. The algorithm is asymmetric and uses Unix *diff* for differencing. There are two main difficulties with this algorithm: changes that are irrelevant to the meaning of the program, like reformatting, can make the output unusable, and some kinds of conflicts are missed. For example, when a variable is renamed in one variant and a line is added in the other that references that variable. In this case, though the variable name from the second reference is wrong, no conflict will be flagged.

3.2. Process Based Algorithms

As an improvement, Mr. Lippe[8] took process based approach. It is an attempt to produce better merging by requiring the editor to be "smart." The editor must maintain a list of all changes made to each file, where a change is a well defined operation. The algorithm uses change lists to compute the merge. Conflicts are determined on a strictly local basis. This approach has the advantage that some conflicts resulting from syntactic changes which have no semantic effect, e.g. reformatting and rudimentary identifier renaming, can be resolved. However, the cost of this approach is rather high. One is restricted to using a particular editor when doing development. Furthermore, knowledge of the program structure is buried in the editor, via special operators like "rename all occurrences of variable x to y" or by promoting simple operations like insert, add, and delete to more complex operations like "move block to subroutine." Still, process based methods are not powerful enough to resolve more than simple renaming conflicts.

3.3. Structure Based Algorithms

The structure based approach falls midway between the semantic and the line and character based approaches. In order to calculate the difference between two revisions, each is parsed into a tree describing the syntactic structure of the program code. Difference and merge are then computed on the parse trees. This method is not as complex as the semantic methods, but its results can not be disturbed by reformatting, as is the case with the line and character methods. Messrs Buffenbarger[2] and Westfechtel[15] have investigated the structural approach to merging. Mr. Westfechtel's work is more interesting, since he uses context-sensitive edges added during editing to capture binding information for resolving non-local name conflicts. Though both techniques solve many of the problems of the line and character based algorithms, there are still significant deficiencies. The binding model for identifier analysis is primitive. Special language features like identifier overloading, structure referencing, and identifier importation can not be handled. Both are unable to resolve conflicts that go beyond simple renaming because no language specific information is available to the merge algorithm.

3.4. Semantic Based Algorithms

The semantic based approach is the holy grail of merging. A true semantic algorithm is concerned with what a program computes, not how it is written. In general, semantic based differencing and merging are undecidable; therefore an approximation must be made. Most of the work in this area has been done by the team of Ms. Horwitz and Messrs Reps, Binkley, Yang, and Prins[3, 13, 16, 1]. They have adapted slicing and compiler techniques for basic block optimization to provide semantic analysis for merging, but the technique has only been demonstrated on toy languages. Though they have developed heuristics for analyzing procedures, no solution has yet been found for handling languages with arrays, pointers, and methods. Semantic based algorithms are beginning to show some promise, but they still appear to be a long way from eclipsing other techniques.

3.5. Summary

Despite the promise of new technology, LCS, line base methods remain standard practice in the software industry for merging. All other methods are too costly in terms of limited applicability or require the use of a special development tools. All methods are limited to single file merging.

4. Design and Implementation

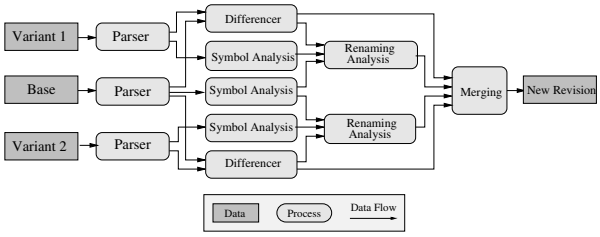


Figure 1. ELAM Data Flow

ELAM improves on structural based merging by using a more sophisticated differencing algorithm and including some semantic information in a language independent form. It is a symmetric, three way merge based on the *linearized tree differencer*. **LTDIFF** is a fast, token based differencing algorithm to identify changes between a base version of a software document and each of variant. Renaming analysis assists the merge with conflict detection and resolution via the **Renaming Detector** [9].

The system is language aware because the core algorithm operates with abstract language constructs instead of language specific constructs. Parser generation[12], modularized symbol analysis, and rule based name analysis generate the underlying abstract representation. This division between input and internal representation makes the system easy to extend and refine to handle new languages. Supporting a new language requires only a new grammar for the parser generator, one new Java class, and a hand full of semantic rules. It took approximately 60 hours to extend the original system to include support for Scheme.

LTDIFF [4] combines the core string comparison techniques of a fast delta compression algorithm[5] with syntactic information from the program versions being compared. It simulates tree comparison with linear token matching. After parsing into a language neutral form, token strings are compared to find the maximal match coverage between two revisions. Then structural information from the parse is used to find the best break point at all places where matches overlap. Thus lexical differencing is used to speed up syntactic comparison.

The **Renaming Detector** inherits the language-aware features of **LTDIFF**: it is insensitive to formatting, matches moved code properly, and treats comments and programming language statements differently. The **Renaming Detector** finds name changes that span multiple files—an important consideration for practical use. Its adaptation to multiple languages is achieved by using **LTDIFF**'s parser, an implementation for an abstract symbol analysis module, and some language specific rules for renaming analysis.

The actual merging phase of **ELAM** sits at the end of the analysis process undertaken by **LTDIFF** and the **Renaming Detector**. First, each of the three input revisions—a common ancestor revision, *base*, and two variants thereof, *left variant* and *right variant*—are parsed and analyzed to determine which tokens are symbols and what attributes they have. The result is two token strings and two symbol tables. Then the token strings and symbol tables for both base/variant pairs are processed to produce two change lists—*left difference* and *right difference*—which describe *left variant* and *right variant* in terms of *base*, respectively. Finally the actual merging process uses the symbol tables, along with *left difference* and *right difference* annotated by renaming analysis to generate a new revision which represents the combination of *base* with all changes made in *left variant* and *right variant*. Figure 1 illustrates how the various parts work together to produce the desired result.

Though not illustrated in figure 1, there is some sharing between the two difference calculations. A suffix tree[10] is used by **LTDIFF** to provide a fast substring index into *base*. Since the same suffix tree can be used for both differencing steps, the suffix tree need be generated only once. This saves considerable time in the overall merge process.

There are not just savings in this process. Conflict resolution requires information from symbol analysis that is not needed for renaming detection but generated there. In order to determine if an apparent conflict between two program statements actually conflict or not, the conflict analysis phase needs to know which references are pure, i.e. do not change the value of the referent or its content, and which are not. The assumption is that all references are impure until shown to be otherwise. The symbol analysis phase determines which references should be marked pure, i.e., references that do not change the value of the referent. For instance, a variable appearing as the destination of a set routine, as in `(set a 5)` in Scheme, is impure, but a variable appearing as an argument to a function in a purely functional language is pure. This part is language specific and is implemented conservatively. Only references that can be shown to be pure are marked so. It is up to each of the language implementations to decide how much processing is worth investing to mark pure references, since the resolution of apparent conflicts works even when no references are marked pure, just not as well.

The output of the merging process is a list of merge blocks or token subsequences. The list can contain in-

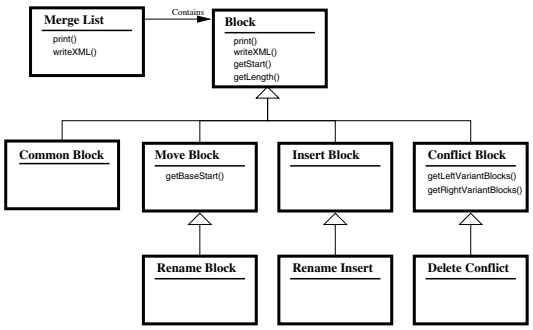


Figure 2. Merge Output

stances of seven block types according to from whence the sequence derives and whether or not there is a conflict: common block, move block, insert block, rename block, rename insert block, conflict block, and delete conflict block. Of these, all but common blocks and conflict blocks have left and right instances (for *left variant* and *right variant*, respectively). The class structure, as depicted in figure 2, is more complicated than what would be needed to textually present the merged revision. Each block class behaves differently during some phase of the merging process. Common blocks, move blocks, and insert blocks represent the unchanged blocks, move blocks, and insert blocks from the difference structures of the input where no conflicts exist. Rename blocks and rename insert blocks reflect the additional blocks added to a difference as a result renaming analysis. Finally, conflict and delete conflict blocks represent conflicts. The structure is designed for further automatic processing and both graphic and textual display with the possibility of manual intervention; therefore it is necessary to record additional information about the blocks found.

The merge algorithm itself has three phases:

1. integrating renamings from each *variant* into the other;
2. building a rough merge from the *variants*; and then
3. resolving apparent conflicts with semantic information.

Renaming integration is carried out directly on the change lists. The second phase generates the first merge list. Finally, the resolution phase removes conflicts by combining the changes from both *left difference* and *right difference* contained in the conflict blocks of the merge list.

4.1. Integrating Renamings

The first phase of merging prepares the change lists for the next two phases. At this stage, each change list contains records for the name changes found by **Renaming Detector** in their respective *variants*. These name changes now need to be projected into the opposite change list. Handling name changes before the main merge phase helps reduce the number of apparent conflicts in the second phase, because marked name changes no longer appear as differences. Furthermore, non local conflicts that the next phase can not find are either removed or marked.

The only place where name changes are difficult for merging is when one *variant* introduces a name change and the other adds a new reference of the identifier whose name has changed. Projecting the name change into the other *variant* removes what would otherwise be a non local merge conflict.

There are two kinds of changes that are of interest: pure name changes and signature changes. As long as only the name of a given identifier is changed in one of the variants, there is no conflict. If there is a change in both, both may be rejected without changing the semantics of the final program. Care must only be exercised with name changes where an identifier's new name already existed in the *base*. In that case, a name change conflict could cause more than one name change to be rejected. Signature changes pose a more complex problem. As with name changing, if no new reference to the changed identifier is added, then local conflict resolution is sufficient to handle the change. However, if a new reference is added in the other variant, the system does not try to automatically resolve the conflict; rather it is simply marked in the change list as a conflict.

4.2. A Rough Merge

The core of the merge algorithm is quite simple. It relies on locality to determine which sequences to include in the resultant merge. This phase of the algorithm is not dissimilar to how the **Unix** command *diff3* works, though *diff3* is an asymmetric merge using the results of a line based differencing algorithm as input.

The algorithm calculates the rough merge in a single pass through the two differences. The algorithm relies on finding corresponding unchanged blocks in both differences and using these as reference points for the merge. Starting with zero as the last alignment point, **ELAM** loops through the following steps, while there are still blocks in both change lists:

1. find next alignment point at an unchanged block or a match rename block;
2. fill the gap between the new alignment point and the last one; and then
3. add a common block or rename block and set the next alignment point to the end of the common or rename block.

Alignment is defined as two unchanged blocks, one from each difference, that contain a token located at a common position in *base*. A block alignment has a length defined as the number of tokens in common with the *base* between

Table 1. Base Decision Table for Filling Gaps

left difference	*right difference*	output
common	—	none
—	common	none
move	—	move block
—	move	move block
move	move	move block for both
common	move	move block
move	common	move block
common	insert	insert block
insert	common	insert block
insert	—	insert block
—	insert	insert block
insert	insert	conflict block
—	insert & common	delete conflict block
insert & common	—	delete conflict block
combination	—	combination of blocks
—	combination	combination of blocks
combination	combination	conflict block

the blocks. There can only be one, possibly empty, string of tokens between any two common blocks, though one unchanged block may have aligned token strings in more than one unchanged block in the other difference. The start position in *base* of the aligned token string is used as the next alignment point in the current iteration, and the last position of the aligned token string is used as the last alignment point for the next iteration.

Once an alignment is established, blocks are added to the merge to reflect the changes. Table 1 illustrated the possible combinations and the resultant actions. In general, if a change has been made to only one of the variants, then there is no conflict. Otherwise a conflict block is inserted. The exception is that, when deletes and moves coincide, the move is simply accepted. Actually, some semantic information is used here as well. Whenever a delete contains a definition of an identifier and a new reference is added in the other variant, a delete conflict block is added to the merge. Each iteration ends with the addition of a common block to represent the aligned token substring.

Figure 3 shows a simple example of the alignment process and figure 4 shows the resultant merge. In this example, there are two insertions, one move, one conflict, and one deletion in the results. The first insertions in the left and the right variant occur at distinct points relative to the base, so both are accepted. The second insertions, marked left conflict and right conflict respectively, occur at the same location relative to the base, thus they conflict. Unchanged C is moved and Unchanged E is deleted.

Aligning common sequences between the two variants is necessary for any three way merge algorithm; however, the conflict detection in this phase of the algorithm is rudimentary and conflict resolution is essentially nonexistent. Though the previous phase insures that all non local con-

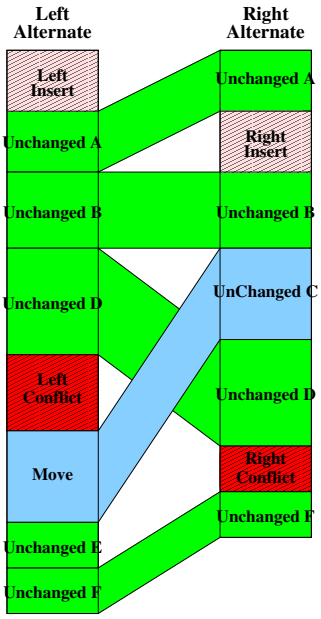


Figure 3. Aligning Blocks

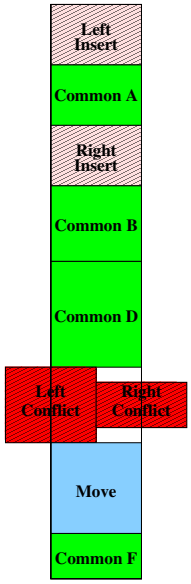


Figure 4. Resultant Merge

flicts are detected, many apparent conflicts remain. For instance, two different methods inserted in each of the two variants at the same position relative to *base* will be flagged as a conflict, when in fact, both can usually be added without causing any difficulty. All apparent conflicts are then examined in the last phase.

4.3. A Little Semantics

The goal of the last phase is to reduce the number of apparent conflicts included in the merge. The strategy is to segregate code sequences into classes according to the difficulty of determining whether or not the two sequences interfere with one another. The strategy has three parts, given a token string from each of the two variants in a conflict block. First, reduce each token string to the smallest set of parse tree nodes that span the token substring exactly. Then, determine a compatibility class for each string by combining the compatibility class of the spanning nodes. Finally, the algorithm uses the two compatibilities to choose the appropriate action.

The system relies on five node compatibility classes. Listed from least restrictive to most restrictive, they are compatible, signature compatible, name compatible, reference compatible, and incompatible. Furthermore, for reference compatibility, references that definitely do not change the state of the referent—pure references—are distinguished from references that might modify the referent—impure references. Table 2 defines the compatibility types

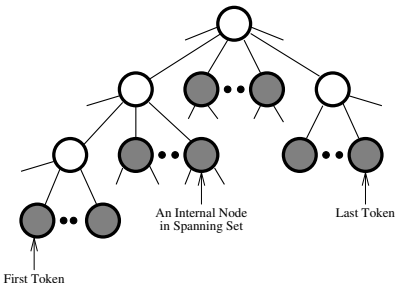


Figure 5. Token String Spanning Nodes

by the addition requirements needed to demonstrate compatibility between token strings of a given compatibility.

Table 2. Compatibility Types

Compatible	two nodes with this attribute are always compatible with other nodes.
Signature Compatible	two nodes with this attribute must differ in name, argument types, or number to be compatible.
Name Compatible	two nodes of this type must differ in name to be compatible.
Reference Compatible	these nodes are compatible when the set of impure reference, in their subnodes, are disjoint.
Incompatible	nodes of this type can never be combined.

The first three classes are easy to handle. Compatibility is the simplest class. It is used mostly for complete comments. Comments can not interfere with program code. A method in Java is an example of signature compatibility. Two methods may be both included in the final merge as long as their name, number of arguments, **or** arguments types differ. The system need only compare the names and argument lists. Name compatibility is a bit more restrictive. A Scheme function is an example of this call. The name alone must be unique.

Reference compatibility is the most difficult. A statement in Java is an example of this compatibility. In this case, the program must examine all identifiers in both token sequences. If no identifier references are common to the nodes or if only pure references are common, then there is no conflict. Otherwise the conflict is unresolvable. If two code segments do not share identifier references or if all the references are pure then no information can flow between the two code segments and no collision is possible. Reference comparison is accomplished by inserting all identifier references from one side of a conflict into a hash table and then looking up all references from the other side.

Finally incompatibility is used for most other token sequences. This is appropriate, since only larger program units can be analyzed reasonably. This class inhibits all compatibility checking. Conflicts involving this node type are unresolvable.

In general, the algorithm needs to consider several nodes from each side of the conflict, not just one. The dominate node, or nodes, of a segment of code is determined, first by finding the nearest common ancestor, and then finding the highest sequence of nodes under that ancestor that do not have subnodes outside the range of the token string. As can be seen in figure 5, the nodes need not all be at the same depth in the tree. Both of these operations are done by a linear search up the parent chains of the first and last token in the range. If there is no single node that spans the code segment exactly, then the compatibility type is that of the most restrictive compatibility.

The compatibility class of each node in the parse tree is given by the grammar for each language. Once the compatibility class for a code segment is determined, conflict resolution can be performed without regard to the underlying language. Only the compatibility types need be considered. Except for incompatible nodes, which can not lead to conflict resolution, nodes of different compatibility types do not conflict and both segments are included in the output. Nodes of the same compatibility class that are compatible according to table 2 do not conflict as well. Again, both segments are included in the output. Otherwise the segments being considered are marked as a conflict.

At the end of this phase, the merge is finished. The results can either be reduced to a text file or used to drive an interactive tool for final conflict resolution. An Emacs extension would be a convenient means of providing such a tool.

5. Evaluation

A lack of real project data makes evaluating **ELAM** difficult. Only very few non trivial merge examples were at hand. During the course of development, only one student worked with the authors on the project, therefore, given the modularity involved only two real merge examples were found. The rest of the examples are constructed to illustrate various features of the merge process.

There are several aspects of the system to evaluate: format independence, name change propagation, and collocated change resolution. Format independence is exhibited when disjoint sets of adds, removes, or changes of code segments can be integrated even when one developer changes the formatting of the entire software document. **ELAM** demonstrates name change propagation by actively changing the name of an identifier when that identifier references a definition whose name was changed in the other variant. Collocation resolution refers to the process of determining whether or not changes made in both variants of a software document are compatible, and hence both allowable, or if they do, in fact, conflict. Eight examples have been constructed to cover each of these cases.

Base

```
class Rename
{
  int _value;
  int getFoo()
  {
    return _value;
  }
}
```

Left Variant

```
class Rename
{
  int _foo;
  public Rename(int foo)
  {
    _foo = foo;
  }
  int getFoo()
  {
    return _foo;
  }
}
```

Right Variant

```
class Rename
{
  int _value;
  int getFoo()
  {
    return _value;
  }
  void setFoo(int value)
  { _value = value; }
}
```

Merge

```
class Rename
{
  int _foo;
  public Rename(int foo)
  {
    _foo = foo;
  }
  int getFoo()
  {
    return _foo;
  }
  void setFoo(int value)
  { _foo = value; }
}
```

Figure 6. Renamed Identifier Example

Base

```
class Test
{
  int _foo;
  int getFoo()
  {
    return _foo;
  }
}
```

Left Variant

```
class Test
{
  public int test1;
  int _foo;
  int getFoo()
  {
    return _foo;
  }
  void setFoo(int value)
  {
    _foo = value;
  }
}
```

Right Variant

```
class Test {
  public float test2;
  int _foo;
  public int getFoo() {
    return _foo;
  }
}
```

Merge

```
class Test
{
  public int test1;
  public float test2;
  int _foo;
  public
  int getFoo()
  {
    return _foo;
  }
  void setFoo(int value)
  {
    _foo = value;
  {
}
```

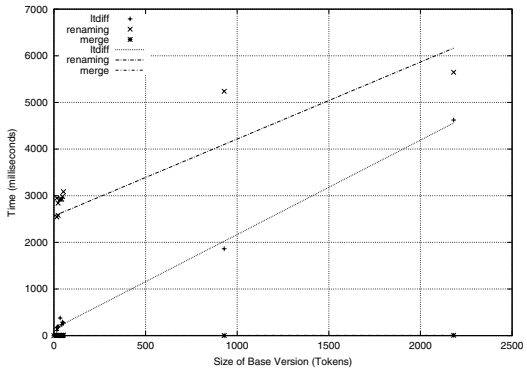
Figure 7. Collocated Change Example

Figures 6 and 7 provide examples of these change types with the merge that **ELAM** produces. In both figures, changes from the left variant are shaded and changes from the right are highlighted with halftone bold type. Figure 6 illustrates how a name change in one variant is propagated into an addition in the other. The left variant, depicts not only an added constructor, but also a variable name change. A new reference to the same variable exists in the right variant. **ELAM** integrates both variants, while correctly changing the name of the variable reference added in the right variant. For clarity, the effected token is boxed in the figure. Figure 7 illustrates both format independence and collocated change. Both variants introduce new public variables at the beginning of the class. Each contain additional changes as well. In particular, the right variant has been reformatted. The resulting merge correctly incorporates all changes. The two new variable definitions do not conflict.

The test cases were also evaluated for runtime performance. The execution environment is an IBM Thinkpad with 300MHz Intel Mobile Pentium II under Linux 2.4 with the Java JDK 1.3.1 from Sun Microsystems. In order to minimize fluctuations due to just-in-time compilation, a trail was run before the actual data was captured.

Ten examples are insufficient to quantify the performance of the algorithm, however the graph in figure 8 does suggest that the actual merge part of the algorithm is insequential in terms of performance. In fact, it appears to be uncorrelated to the size of the input in tokens. Even when renaming analysis is only undertaken within each file singularly, renaming still accounts for the largest portion of the total execution time. With the two preliminary paths that are required for external reference analysis, renaming analysis would dwarf all other costs. Though overall performance is acceptable, renaming analysis provides a ample opportunity for performance improvements.

Figure 8. Runtime Performance of ELAM



Correlations: ltdiff 0.998; renaming 0.765; merge 0.153

Figure 9. TokenAST.java Base Version

```
...
public class TokenAST extends Token implements Atom, AST
{
  ...
  public int getPosition() {
      int pos = super.getPosition();
      if (pos >= 0) return pos;
      if (getFirstChild() != null)
          return ((TokenAST)getFirstChild()).getPosition();
      else
          return pos;
  }
  ...
  /** Get the first child of this node; null if not children */
  public AST getFirstChild() { return _down; }
  /** Get       the next sibling in line after this one */
  public AST getNextSibling() { return _right; }
  ...
  public void setFirstChild(AST child) { _down = (TokenAST)child; }
  public void setNextSibling(AST next) { _right = (TokenAST)next; }
  /** Print out a child-sibling tree in LISP notation */
  public String toStringList()
  {
    TokenAST token = this;
    String token_string = "";
    if (token.getFirstChild() != null) token_string += " (";
    token_string += " " + this.toString();
    if (token.getFirstChild() != null)
      token_string += ((TokenAST)token.getFirstChild()).toStringList();
    if (token.getFirstChild() != null) token_string += ")";
    if (token.getNextSibling() != null)
      token_string += ((TokenAST)token.getNextSibling()).toStringList();
    return token_string;
  }
  ...
}
```

5.1. An Example from the Field

Figures 9 through 13 depict a merge case encountered in actual code development. Merging was necessitated by changes to the same file in parallel. Merge results are given both for *diff3* and for **ELAM**. Though some of the original code has been replaced by ellipsis, so as to shorten the example for improved readability, the example clearly illustrates the advantages of **ELAM** over *diff3*.

Figure 9 shows the basis version of the TokenAST.java file. Figure 10 shows the first variant of TokenAST.java, where the developer added three new methods to the class. Figure 11 gives the second variant. Here, a method named `equals` is added and two methods are rewritten with new names given to the old methods: `getFirstChild` becomes `getFirstChildOrComment` and `getNextSibling` becomes `getNextSib-`

Figure 10. TokenAST.java Variant 1

```
...
public class TokenAST extends Token implements Atom, AST
{
  ...
  public int getPosition() {
      int pos = super.getPosition();
      if (pos >= 0) return pos;
      if (getFirstChild() != null)
          return ((TokenAST)getFirstChild()).getPosition();
      else
          return pos;
  }
  public boolean isFirst()
  {
    return (_parent == null) || (_parent.getFirstChild() == this);
  }
  public boolean isLast() { return this.getNextSibling() == null; }
  ...
  /** Get the first child of this node; null if not children */
  public AST getFirstChild() { return _down; }
  /** Get       the next sibling in line after this one */
  public AST getNextSibling() { return _right; }
  ...
  public void setFirstChild(AST child) { _down = (TokenAST)child; }
  public void setNextSibling(AST next) { _right = (TokenAST)next; }
  public Atom findBalanceAtom()
  {
    int end_type = getBalanceType();
    if (end_type == InputState.BALANCE_DEFAULT) return null;
    else
      {
        TokenAST result;
        for (result = this;
             (result != null) && (result.getType() != end_type);
             result = (TokenAST)result.getNextSibling());
        return result;
      }
  }
  /** Print out a child-sibling tree in LISP notation */
  public String toStringList()
  {
    TokenAST token = this;
    String token_string = "";
    if (token.getFirstChild() != null) token_string += " (";
    token_string += " " + this.toString();
    if (token.getFirstChild() != null)
      token_string += ((TokenAST)token.getFirstChild()).toStringList();
    if (token.getFirstChild() != null) token_string += ")";
    if (token.getNextSibling() != null)
      token_string += ((TokenAST)token.getNextSibling()).toStringList();
    return token_string;
  }
  ...
}
```

Figure 11. TokenAST.java Variant 2

```
...
/** NEW:  !!!!!    21.08.2001
 * getNextSibling() and getFirstChild() do not return comment-nodes anymore.
 * If you want to get comments, you need to use getNextSiblingOrComment() and
 * getFirstChildOrComment()
 */
public class TokenAST extends Token implements Atom, AST
{
  ...
  public int getPosition() {
      int pos = super.getPosition();
      if (pos >= 0) return pos;
      if (getFirstChildOrComment() != null)
          return ((TokenAST)getFirstChildOrComment()).getPosition();
      else
          return pos;
  }
  ...
  /**
   * Get the first child of this node; null if not children
   * getNextSibling() and getFirstChild() do not return comment-nodes anymore.
   * If you want to get comments, you need to use getNextSiblingOrComment() and
   * getFirstChildOrComment()
   */
  public AST getFirstChild() {
    TokenAST tmp = _down;
    while (tmp != null && tmp.isComment())
      tmp = tmp._right;
    return tmp;
  }
  public AST getFirstChildOrComment() { return _down; }
  /** Get the next sibling in line after this one
   * getNextSibling() and getFirstChild() do not return comment-nodes anymore.
   * If you want to get comments, you need to use getNextSiblingOrComment() and
   * getFirstChildOrComment()
   */
  public AST getNextSibling() {
    TokenAST tmp = _right;
    while (tmp != null && tmp.isComment())
      tmp = tmp._right;
    return tmp;
  }
  public AST getNextSiblingOrComment() { return _right; }
  ...
  public void setFirstChild(AST child) { _down = (TokenAST)child; }
  public void setNextSibling(AST next) { _right = (TokenAST)next; }
  public boolean equals(Atom other)
  {
    Atom other_parent = ((TokenAST)other).getParent();
    if (getParent() == null)
      return ((((Token)other).getType() == getType()) &&
              (((Token)other)._text == _text) &&
              (other_parent != null));
    else
      return ((((Token)other).getType() == getType()) &&
              (((Token)other)._text == _text) &&
              (other_parent != null) &&
              (((Token)other_parent).getType() == getParent().getType()));
  }
  /** Print out a child-sibling tree in LISP notation */
  public String toStringList()
  {
    TokenAST token = this;
    String token_string = "";
    if (token.getFirstChildOrComment() != null) token_string += " (";
    token_string += " " + this.toString();
    if (token.getFirstChildOrComment() != null)
      token_string += ((TokenAST)token.getFirstChildOrComment()).toStringList();
    if (token.getFirstChildOrComment() != null) token_string += ")";
    if (token.getNextSiblingOrComment() != null)
      token_string +=
        ((TokenAST)token.getNextSiblingOrComment()).toStringList();
    return token_string;
  }
  ...
}
```

Figure 12. TokenAST.java Merge with diff3

```
. . .
/** NEW:  !!!!!    21.08.2001
 * getNextSibling() and getFirstChild() do not return comment-nodes anymore.
 * If you want to get comments, you need to use getNextSiblingOrComment() and
 * getFirstChildOrComment()
 */
public class TokenAST extends Token implements Atom, AST
{
  . . .
  public int getPosition() {
      int pos = super.getPosition();
      if (pos >= 0) return pos;
      if (getFirstChildOrComment() != null)
          return ((TokenAST)getFirstChildOrComment()).getPosition();
      else
          return pos;
  }
  public boolean isFirst()
  {
    return (_parent == null) || (_parent.getFirstChild() == this);
  }
  public boolean isLast() { return this.getNextSibling() == null; }

  . . .
  /**
   * Get the first child of this node; null if not children
   * getNextSibling() and getFirstChild() do not return comment-nodes anymore.
   * If you want to get comments, you need to use getNextSiblingOrComment() and
   * getFirstChildOrComment()
   */
  public AST getFirstChild() {
    TokenAST tmp = _down;
    while (tmp != null && tmp.isComment())
      tmp = tmp._right;
    return tmp;
  }
  public AST getFirstChildOrComment() { return _down; }
  /** Get the next sibling in line after this one
   * getNextSibling() and getFirstChild() do not return comment-nodes anymore.
   * If you want to get comments, you need to use getNextSiblingOrComment() and
   * getFirstChildOrComment()
   */
  public AST getNextSibling() {
    TokenAST tmp = _right;
    while (tmp != null && tmp.isComment())
      tmp = tmp._right;
    return tmp;
  }
  public AST getNextSiblingOrComment() { return _right; }
  . . .
  public void setFirstChild(AST child) { _down = (TokenAST)child; }
  public void setNextSibling(AST next) { _right = (TokenAST)next; }
<<<<<<< cases/TokenAST.4.java
  public Atom findBalanceAtom()
  {
    int end_type = getBalanceType();
    if (end_type == InputState.BALANCE_DEFAULT) return null;
    else
      {
        TokenAST result;
        for (result = this;
             (result != null) && (result.getType() != end_type);
             result = (TokenAST)result.getNextSibling());
        return result;
      }
  }
||||||| cases/TokenAST.0.java
=======
  public boolean equals(Atom other)
  {
    Atom other_parent = ((TokenAST)other).getParent();
    if (getParent() == null)
      return ((((Token)other).getType() == getType()) &&
              (((Token)other)._text == _text) &&
              (other_parent != null));
    else
      return ((((Token)other).getType() == getType()) &&
              (((Token)other)._text == _text) &&
              (other_parent != null) &&
              (((Token)other_parent).getType() == getParent().getType()));
  }
>>>>>>> cases/TokenAST.5.java
  /** Print out a child-sibling tree in LISP notation */
  public String toStringList()
  {
    TokenAST token = this;
    String token_string = "";
    if (token.getFirstChildOrComment() != null) token_string += " (";
    token_string += " " + this.toString();
    if (token.getFirstChildOrComment() != null)
      token_string += ((TokenAST)token.getFirstChildOrComment()).toStringList();
    if (token.getFirstChildOrComment() != null) token_string += ")";
    if (token.getNextSiblingOrComment() != null)
      token_string +=
        ((TokenAST)token.getNextSiblingOrComment()).toStringList();
    return token_string;
  }
  . . .
}
```

Figure 13. TokenAST.java Merge with ELAM

```
. . .
/** NEW:  !!!!!    21.08.2001
 * getNextSibling() and getFirstChild() do not return comment-nodes anymore.
 * If you want to get comments, you need to use getNextSiblingOrComment() and
 * getFirstChildOrComment()
 */
public class TokenAST extends Token implements Atom, AST
{
  . . .
  public int getPosition() {
      int pos = super.getPosition();
      if (pos >= 0) return pos;
      if (getFirstChildOrComment() != null)
          return ((TokenAST)getFirstChildOrComment()).getPosition();
      else
          return pos;
  }
  public boolean isFirst()
  {
    return (_parent == null) || (_parent.getFirstChildOrComment() == this);
  }
  public boolean isLast() { return this.getNextSiblingOrComment() == null; }
  . . .
  /**
   * Get the first child of this node; null if not children
   * getNextSibling() and getFirstChild() do not return comment-nodes anymore.
   * If you want to get comments, you need to use getNextSiblingOrComment() and
   * getFirstChildOrComment()
   */
  public AST getFirstChild() {
    TokenAST tmp = _down;
    while (tmp != null && tmp.isComment())
      tmp = tmp._right;
    return tmp;
  }
  public AST getFirstChildOrComment() { return _down; }
  /** Get the next sibling in line after this one
   * getNextSibling() and getFirstChild() do not return comment-nodes anymore.
   * If you want to get comments, you need to use getNextSiblingOrComment() and
   * getFirstChildOrComment()
   */
  public AST getNextSibling() {
    TokenAST tmp = _right;
    while (tmp != null && tmp.isComment())
      tmp = tmp._right;
    return tmp;
  }
  public AST getNextSiblingOrComment() { return _right; }
  . . .
  public void setFirstChild(AST child) { _down = (TokenAST)child; }
  public void setNextSibling(AST next) { _right = (TokenAST)next; }
  public Atom findBalanceAtom()
  {
    int end_type = getBalanceType();
    if (end_type == InputState.BALANCE_DEFAULT) return null;
    else
      {
        TokenAST result;
        for (result = this;
             (result != null) && (result.getType() != end_type);
             result = (TokenAST)result.getNextSiblingOrComment());
        return result;
      }
  }
  public boolean equals(Atom other)
  {
    Atom other_parent = ((TokenAST)other).getParent();
    if (getParent() == null)
      return ((((Token)other).getType() == getType()) &&
              (((Token)other)._text == _text) &&
              (other_parent != null));
    else
      return ((((Token)other).getType() == getType()) &&
              (((Token)other)._text == _text) &&
              (other_parent != null) &&
              (((Token)other_parent).getType() == getParent().getType()));
  }
  /** Print out a child-sibling tree in LISP notation */
  public String toStringList()
  {
    TokenAST token = this;
    String token_string = "";
    if (token.getFirstChildOrComment() != null) token_string += " (";
    token_string += " " + this.toString();
    if (token.getFirstChildOrComment() != null)
      token_string += ((TokenAST)token.getFirstChildOrComment()).toStringList();
    if (token.getFirstChildOrComment() != null) token_string += ")";
    if (token.getNextSiblingOrComment() != null)
      token_string += ((TokenAST)token.getNextSiblingOrComment()).toStringList();
    return token_string;
  }
  . . .
}
```

lingOrComment. All existing references to the old methods are changed to the new names. For convenience, the changes of variant 1 are highlighted with a gray background and the changes of variant 2 are set in halftone bold type.

The reader should note that the methods added in variant 1 reference the methods whose names have been changed in variant 2.

As can be seen in figure 12, *diff3* is not able to merge the first change correctly, thus resulting in two errors. One of the errors is clearly visible from the result. The method named `findBalanceAtom` added in variant 1 and the method named `equals` where both added at the same place relative to the base revision. Since *diff3* does not have any semantic knowledge about the code in question, it is not able to determine whether or not there is a semantic collision. It must flag this textual collision as a conflict. The other error slips by *diff3* undetected. The *diff* algorithm has no way of recognizing that the methods added in variant 1 reference the methods whose names are changed in variant 2. The references in the methods added in variant 1 to the methods renamed in variant 2 should be changed to use the new names. In figure 12, the places where this should happen are marked with a surrounding rectangle.

The merge produced by **ELAM**, given in figure 13, does not exhibit these errors. First, it recognizes that the references to `getFirstChild` and `getNextSibling` in the methods added in variant 1 must be changed to `getFirstChildOrComment` and `getNextSiblingOrComment` respectively. Second, the algorithm gathers enough semantic information to know that the new method named `findBalanceAtom` from variant 1 and the new methods named `equals` from variant 2 do not conflict. **ELAM** includes both methods in the new version resulting in a semantically correct merge.

6. Future Work

There are two areas that could be improved upon: language support and conflict resolution. The current implementation works well with languages like Java and Scheme, where no preprocessor is used, but languages that rely on preprocessors, like C, and C++, are more difficult to process. More sophisticated parsing techniques are needed to fully support these languages. The current conflict resolution strategy is rather simple. More experience should yield useful information about improving the strategy.

7. Conclusions

The **ELAM** algorithm is a new algorithm that provides a better trade off among performance, applicability, and accuracy. On the one hand, it can find non-local conflicts that line based merging misses and exclude apparent conflicts that line based merging generates. On the other hand, **ELAM** has wider applicability than current full semantic approaches. An implementation in Java demonstrated that the algorithm is effective. Parsers are available for Java and Scheme. The **ELAM** algorithm achieves the main goal of providing full syntactic merging with some semantic analysis in acceptable time.

References

[1] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Trans. on Software Eng. and Methodology*, 4(1):3–35, Jan. 1995.

[2] J. Buffenbarger. Syntactic software merging. *Lecture Notes in Comp. Sci.*, 1005: ICSE SCM-4 and SCM-5 Workshop Sel. Papers:53–67, 1995.

[3] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Trans. on Programming Languages and Systems*, 11(3):345–387, July 1989.

[4] J. J. Hunt. *Extensible, Language-Aware Differencing and Merging*. PhD thesis, Universitt Karlsruhe, Nov. 2001.

[5] J. J. Hunt, K.-P. Vo, and W. F. Tichy. An empirical study of delta algorithms. *ACM Trans. on Software Eng. and Methodology*, 7(2):49–66, 1998.

[6] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, May 1977.

[7] R. Irving and C. Fraser. Two algorithms for the longest common subsequence of three (or more) strings. In U. Manber, Z. Galil, M. Crochemore, and A. Apostolico, editors, *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, number 644, pages 214–229. Springer-Verlag, 1992.

[8] E. Lippe. Operation-based merging. In H. Weber, editor, *Software Eng. Notes: Proc. of the 5th ACM SIGSOFT Symp. on Software Development Environments*, pages 78–87. ACM, Dec. 1992.

[9] G. Malpohl, J. J. Hunt, and W. F. Tichy. Renaming detection. In *Proc. of the 15th Inter. Conf. on Automated Software Eng.* IEEE Computer Society Press, Sept. 2000.

[10] E. M. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 32:262–272, 1976.

[11] E. W. Meyers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[12] T. Parr and R. Quong. A Predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995.

[13] T. Reps, S. Horwitz, and J. Prins. Support for integrating program variants in an environment for programming in the large. In *Proc. of the Inter. Workshop on Software Version and Config. Control*, Stuttgart, FRG, Jan. 1988. Teubner Verlag.

[14] W. F. Tichy. RCS: A revision control system. In *Integrated Interactive Computing Systems*. North-Holland Publishing Co, 1983.

[15] B. Westfechtel. Structure-oriented merging of revisions of software documents. In *Proc. of the 3rd Inter. Workshop on SCM*, pages 68–79. ACM Press, 1991.

[16] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accommodates semantic-preserving transformations. *ACM Trans. on Software Eng. and Methodology*, 1(3):310–354, July 1992.