Structure-Oriented Merging of Revisions of Software Documents

Bernhard Westfechtel
Lehrstuhl für Informatik III
Aachen University of Technology
Ahornstr. 55
D-5100 Aachen

Abstract

Merging revisions of software documents after development has branched into multiple lines is a difficult task. Previous approaches to merging are either based on text files or refer to specific languages. These approaches do not meet the requirements to a merge tool which is to be integrated into a multilingual structure-oriented environment. In this paper, we present a structure-oriented merge tool that is applicable to software documents (requirements definitions, software architecture descriptions, module implementations, etc.) written in arbitrary languages, preserves their context-free correctness, and also takes binding of identifiers to their declarations into account.

1 Introduction

In the course of the construction of software systems **software documents** such as requirements definitions, software architecture descriptions, module implementations, etc. pass through different states of development. In order to support the maintenance of long-lived software systems, it is essential to record snapshots of documents which reflect their development histories. These snapshots are called **revisions**.

In the simplest case, the revisions of a document are **ordered linearly** along the time axis. However, the development history may also **branch**, e.g. when bugs in an old revision have to be fixed while simultaneously a new revision is being developed. Later-on, it may be desirable to **merge** the changes performed on different branches, i.e. to produce a common successor that incorporates all of these changes.

Merging revisions manually is both time-consuming and error-prone. Thus, a **tool** is urgently needed which **automates** the process of **merging** as far as possible. Since the changes performed on different branches may interfere, merging cannot be automated completely. Rather, the merge tool should combine non-interfering changes automatically and consult the user when interference is detected.

The designer of a merge tool has to balance the following contradictory requirements:

- On the one hand, the merge tool should be **general**, i.e. it should be applicable to arbitrary software documents. To this end, the tool has to abstract from the languages in which the documents are written.
- On the other hand, the merge tool should be **intelligent**, i.e. it should be based on a high-level concept of change in order to produce a result which makes sense. To this end, the tool has to incorporate language-specific knowledge.

This paper deals with merging in a **structure-oriented environment** that provides tools for many different kinds of documents. In this context, we feel that previous approaches to merging are either too general or too specific and fail to achieve an appropriate balance of the requirements stated above:

- On the one hand, there are merge tools based on text files /AGM 86, LCS 88, Ti 85/. These tools do a good job in tool-kit environments, but they are not suited for a structure-oriented environment because they do not take the underlying syntactic and semantic structures into account.
- On the other hand, there are approaches tailored to a specific language /HPR 89, YHR 90, Be 86, Fe 89/. These approaches allow to make more intelligent merge decisions, but they are highly specialized. Furthermore, practical tools relying on such approaches are not yet available. Finally, writing a merge tool for

each language anew is impractical because there are lots of languages.

In order to fill this gap, we have developed a merge algorithm which has the following properties:

- It is applicable to documents written in **arbitrary languages**.
- It preserves the **context-free correctness**: Starting from context-free correct revisions, the merge tool produces a result which is also correct with respect to the context-free syntax of the underlying language.
- It detects **context-free conflicts**, i.e. interfering changes to the context-free structure.

Furthermore, we have also developed an extension of the context-free merge algorithm that deals with an important aspect of the context-sensitive syntax, namely the **binding of applied occurrences of identifiers** to their definitions. This extension does also not refer to a specific language; however, it is currently confined to a standard model of binding. The context-sensitive extension detects conflicts which are frequently overlooked when the merge algorithm only takes the context-free structure into account.

The remainder of the paper is organized as follows: In section 2, we explain the context of our work. In section 3, we describe context-free merging. In section 4, we extend the approach developed in section 3 in order to take bindings of identifiers into account. In section 5, we compare our approach with previous work on merging. In section 6, we summarize our results and outline topics of future research.

2 Background

The work presented here is part of a research effort which aims at extending an integrated structure-oriented environment with **revision control** /We 91/. Previous papers dealt with other aspects of revision control, namely efficient storage of revisions /We 89a/ and control of external consistency between revisions of interdependent documents /We 89b/. While we do not require knowledge of these papers here, we do have to give a short introduction to some general features of our structure-oriented environment.

The work presented in this paper has been carried out within the research project **IPSEN** (Integrated Project Support ENvironment, /Na 90/). IPSEN is dedicated to the development of structure-oriented environments covering the whole software life cycle. At the heart of an IPSEN environment, there are structure-oriented editors for textual languages. These editors enforce the context-free correctness of documents; furthermore, they incrementally check, but do not enforce their context-sensitive correctness (correctness with respect to static semantics).

The context-free syntax of a textual language is defined by means of a **normalized EBNF grammar** /En 86/ which is close to a GRAMPS grammar /Cl 84/. It solely consists of productions of the following kinds[1]:

- **alternative productions** which define a selection, e.g.
  Declaration ::= ConstDeclList | VarDeclList |
  TypeDeclList | ProcDecl
- **list productions** which define list structures, e.g.
  VarDeclList ::= "VAR" { VarDecl } VarDecl
- **structure productions** which define record structures, e.g.
  VarDecl ::= DeclIdentList ":" TypeDef ";"

The **context-free structure** of a software document is internally represented by an **abstract syntax tree** which abstracts from keywords and delimiters. An abstract syntax tree is modeled as a graph which contains typed nodes and edges and whose nodes may carry attributes. Each subtree corresponds to an **increment**, i.e. an instance of a syntactic unit of the underlying language. Fig. 1 shows an example for the mapping from concrete to abstract syntax. A list is represented by a list node (nodes 1 and 3) which is connected with the first, the last, and every element by ToFirst-, ToLast-, and ToElem-edges, respectively. ToNext-edges define the order of the list elements. A structure is represented by a structure node (node 2) which is connected with its sons by ToFirstSon-, ToSecondSon-, ... edges. An unexpanded alternative is represented by a placeholder node (node 6). Finally, an identifier is represented by an identifier node (nodes 4 and 5) which carries the corresponding name as an attribute.

In order to model the **context-sensitive structure** of a document, its abstract syntax tree is augmented with **non-tree edges** which represent context-sensitive relations. In this paper, we will only be concerned with bindings of identifiers to their declarations. Such a binding is represented by an edge which connects the applied occurrence with its declaration. Then, the internal representation of a document is an attributed, node and edge labeled graph. In general, the use of **graphs** allows for the uniform mod-

1. All examples in this paper are drawn from Modula-2.

concrete syntax:

VAR V1, V2 : (<TypeDef>);

abstract syntax:

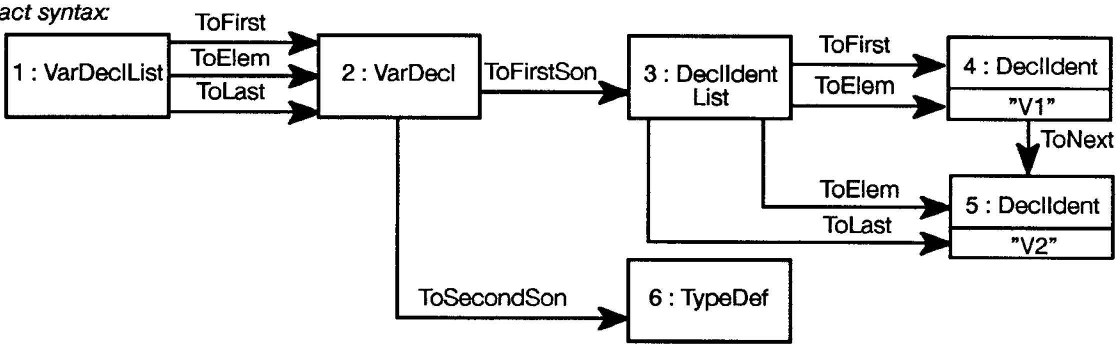

Fig. 1 Relation between concrete and abstract syntax

eling of context-free and context-sensitive relations.This distinguishes our approach from attribute grammars which use different data models (and data structures) for the representation of context-sensitive relations /RT 88/.

Documents of all types are implemented as abstract data types with a **uniform interface.** Each change operation which is provided by that interface modifies the abstract syntax tree and collects information required for incremental context-sensitive checks. These checks are performed on demand and update context-sensitive relations according to the modifications of the abstract syntax tree. The following **primitive change operations** are provided by the interface: ExpandPlaceholder, InsertElementInEmptyList, PreInsertElementInList, PostInsertElementInList, DeleteIncrement (deletion of a subtree corresponding to a syntactic unit), and ExtendIdentifier.

3 Context-free Merging

In this section, we present a merge algorithm which merges **tree-oriented changes** of documents written in **arbitrary languages.** The merge algorithm preserves the **context-free correctness** and detects **context-free conflicts.** In section 3.1, we describe its underlying principles in an informal way. In section 3.2, we present (a cut-out of) the algorithm in a Modula-2-like notation.

3.1 Principles

Fig. 2 illustrates the merge problem: Starting from a **base revision** b, two **alternative revisions** a_1 and a_2 were developed. Now, a **merge revision** m is to be constructed which combines a_1 and a_2 with respect to b.

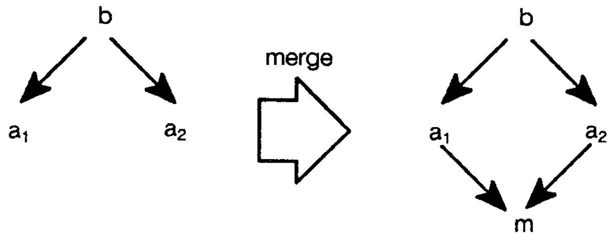

Fig. 2 The merge problem

Fig. 3 restates the merge problem in a different way: The merge algorithm is supplied with two **deltas** (sequences of change operations)

d_i = delta(b, a_i) (i = 1, 2) ,

from which a single **merge delta**

d = delta(b, m)

is to be constructed. Note that the change operations we consider here are the primitive change operations mentioned at the end of section 2.

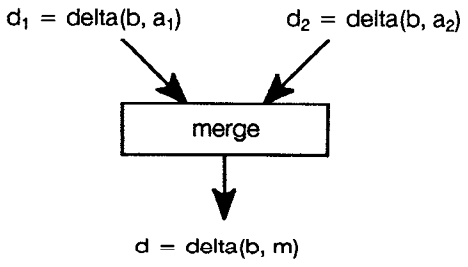

Fig. 3 Merging of deltas

The merge delta d has to meet the following **requirements**:

- Each operation in d must come from either d_1 or d_2.
- d must not contain duplications.

- d must not contain invalid operations, i.e. operations that change an increment which is deleted on the other branch.

Considering merging as the combination of deltas leads to the following simple definition of a **merge conflict**:

- Let operations op_1 and op_2 come from d_1 and d_2, respectively. Then, op_1 and op_2 interfere iff the changes which they perform depend on the order of their execution.

Where do the **input deltas** d_1 and d_2 come from? Here, two cases have to be distinguished:

- The actual development history is known. In this case, deltas are given (either implicitly or explicitly, see below).
- The actual development history is unknown. In this case, deltas have to be reconstructed.

In the second case, complex heuristic tree comparisons have to be performed /ZS 89, Se 77, Ta 79/. In order to avoid this problem, we have modified our structure-oriented environment so that the merge algorithm has access to the **actual development history**. This modification is simple: Instead of logging change operations (explicit deltas), the nodes of abstract syntax trees are decorated with **tags** having the following properties[2]:

- When a node is created, it is assigned a unique tag which is never reused.
- Tags are immutable.
- The operation which copies a revision preserves the tags.

By means of tags, the changes that were actually performed by proceeding from one revision to another may easily be reconstructed by **pair-wise comparisons** of the input revisions b, a_1, and a_2 (implicit deltas). Thus, the merge tool realizes the functionality "combination of deltas" without dealing with sequences of operations explicitly[3].

Before discussing these comparisons, let us first explain how the **language independence** of our merge tool is accomplished (fig. 4). This is achieved by means of an **object-oriented design**: The class Document is an abstract superclass (i.e. a class which has no instance) which provides a uniform interface to documents written in arbitrary languages. This interface includes the tree-oriented operations mentioned in section 2. These operations are deferred methods (i.e. no implementations are provided in the superclass). For each language, a subclass of Document (e.g. ModulaDocument) is defined which provides implementations for the deferred methods of the superclass. The merge algorithm exclusively relies on the interface provided by Document; therefore, it is language independent.

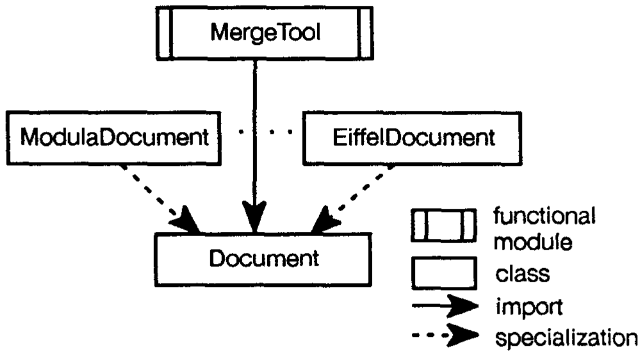

Fig. 4 Language independence of the merge tool

The merge tool described in this paper is an integrated part of the multilingual, structure-oriented IPSEN environment the components of which may be classified as follows:

- **Language independent components** which are reused for all kinds of languages without any modification.
- **Language specific components** which are **generated** from a normalized EBNF grammar.
- **Language specific components** which have to be **written by hand.**

In order to extend IPSEN with a new language L, one has to write a normalized EBNF grammar. Then, the context-free part of a structure-oriented editor for L ist generated automatically. Incremental context-sensitive checks have to be written by hand (with considerable assistance of some language independent components). One of the results of these activities is a subclass of Document which provides tree-oriented operations which also include context-sensitive checks. No additional effort is required to apply the language independent merge tool to documents written in L: The tool is supplied with (revisions of) L-documents and calls procedures exported by the superclass Document. By means of dynamic binding, their language specific implementations are executed.

2. Tags are also used in other approaches to merging, e.g. /HPR 89/.
3. Note that tags play a central role in our approach to revision control. In /We 89b/, tags are used for incremental updating of revisions of dependent documents.

class of i	*alternative a_1*	*alternative a_2*	*base b*	*merge result m*
identifier	i.Name = n_1	i.Name = n_2	i.Name = n_1	i.Name = n_2
identifier	i.Name = n_1	i.Name = n_2	i.Name = n_3	conflict
structure	i.Son$_k$ = i_1	i.Son$_k$ = i_2	i.Son$_k$ = i_1	i.Son$_k$ = i_2
structure	i.Son$_k$ = i_1	i.Son$_k$ = i_2	i.Son$_k$ = i_3	conflict
list	$i_1 \in i$	$i_1 \notin i$	$i_1 \in i$	$i_1 \notin i$
list	$i_1 \in i$	$i_1 \notin i$	$i_1 \notin i$	$i_1 \in i$, potential conflict

Fig. 5 Structure–oriented 3–way merge rules

The merge algorithm performs a **structure–oriented 3–way–comparison**. When a difference between a_1 and a_2 is detected, the base is consulted to find out which changes were performed on each branch. If there is no interference, merging proceeds automatically; otherwise, the user is asked to resolve the conflict.

Fig. 5 summarizes the **3–way–rules** which direct the merge process. Each rule refers to an increment i which is in all input revisions and therefore will be included in the merge revision. Instances of the same increment are identified by means of their tags.

The merge algorithm operates **symmetrically**, i.e. the result which it produces does not depend on the order in which the alternative revisions are passed as parameters. Note that the symmetric cases are not contained in fig. 5.

According to the class of increment i, the merge rules are divided into the following groups:

- **Identifier rules** compare the name attributes in a_1, a_2, and b. If only the names in a_1 and b are equal, then a name change was performed in a_2. Therefore, in m the name n_2 is assigned to i.Name. A conflict occurs if all names are pair-wise distinct. In this case, the user has to select either n_1 or n_2[4].
- **Structure rules** compare corresponding components (the first son, the second son, etc.) of structures by means of their tags. If a component was replaced on exactly one branch, it is automatically copied into m. A conflict occurs if the same component was replaced on both branches. In this case, the user has to determine which component is copied into m.
- **List rules** handle changes to list structures. If a list element i_1 is both in a_1 and b, but not in a_2, it was deleted from a_2 and therefore is excluded from m. If i_1 is not in the base, it was inserted into a_1. In this case, it is also inserted into m. A conflict occurs if two elements were inserted at the same position. Then the user has to determine the order in which the elements appear in m.

4. Of course, we could have allowed the user to type in a new name. However, this is a special case of an edit operation. So far, integration of merging and editing is not supported. In principle, integration is possible, but requires a restricted edit mode.

In addition to 3–way rules, **2–way–rules** are also required. 2–way rules are applied to increments which are in both alternatives, but not in the base. For example, this might happen in the following situation (fig. 6): After having derived r_2 from r_1 and r_3, r_4 from r_2, the development history is purged by removing r_2. Subsequently, r_3 and r_4 are merged with respect to r_1. In this situation, r_3 and r_4 may contain common increments which were originally created in r_2 and therefore are not in r_1.

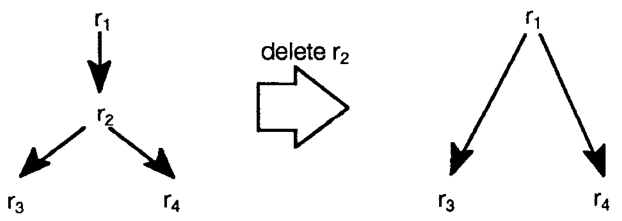

Fig. 6 Example for the necessity of 2–way rules

2–way rules differ from 3–way rules inasmuch as any **difference** has to be handled as a **conflict**. Note that the base does not contain any information which may be exploited to automate merge decisions. For example, if a list element is in only one alternative, the user has to decide whether it shall be inserted into the merge revision. In the sequel, we will not further discuss 2–way rules because they can be treated similarly to 3–way conflicts.

The structure-oriented rules presented above have the following **properties**:

- The merge delta which they (conceptually) produce meets the requirements that were specified at the beginning of section 3.1. Furthermore, the definitions of the various sorts of conflicts are consistent with the general delta-based definition of a merge conflict.
- The application of merge rules preserves the context-free correctness. Note that only name changes, delete and copy operations are involved. Since source and destination of a copy operation are located at the same place in different revisions, a copy operation is guaranteed to preserve the context-free correctness.
- The merge rules do not contain any language-specific knowledge. First, they only refer to general classes of increments (e.g. lists) and not to specific instances of such classes (e.g. VarDeclList). Second, all change operations may be performed without reference to the underlying language.

Fig. 7 presents an **example** which illustrates the application of the merge rules presented above. To keep the figure simple, revisions are represented textually rather than as abstract syntax trees. Insertions are shown in bold face, while changes are italicized. For the ease of reference, lines are numbered consistently in all revisions. The following rules are applied:

- By means of a list rule, the declaration of ColourType (line 2 in a_2) is inserted into m.
- By means of a structure rule, the type definition of the variable Colour (line 3) is replaced with an applied occurrence of the type ColourType.
- By means of an identifier rule, a conflict is detected in line 5: In b, a_1, and a_2 the variable Colour is assigned pair-wise distinct values (which syntactically are applied occurrences of identfiers that are introduced in the enumeration type definition). In the figure, we assume that the user selects the value Grey.

base revision b:

```
1 MODULE M;
3     VAR Colour : (White, Grey, Black);
4 BEGIN
5     Colour := White
6 END M.
```

alternative revision a_1:

```
1 MODULE M;
3     VAR Colour : (White, Grey, Black);
4 BEGIN
5     Colour := Grey
6 END M.
```

alternative revision a_2:

```
1 MODULE M;
2     TYPE ColourType = (White, Grey, Black);
3     VAR Colour : ColourType;
4 BEGIN
5     Colour := Black
6 END M.
```

merge revision m:

```
1 MODULE M;
2     TYPE ColourType = (White, Grey, Black);
3     VAR Colour : ColourType;
4 BEGIN
5     Colour := Grey
6 END M.
```

Fig. 7 Example for the application of merge rules

3.2 The Algorithm

In the sequel, we present (parts of) the **algorithm** for the context-free merging of revisions of software documents. The notation which we use for that purpose is a Modula-2-like pseudo code. The merge algorithm operates on a copy of a_1 which is gradually modified by application of the merge rules which have informally been desribed in the last section. However, note that the merge result does not depend on which alternative revision is chosen as starting point (symmetry of merging, see section 3.1).

```
procedure Merge (var m : Rev; a2, b : Rev; i : Tag);
begin
  (* increment i is in m and a2 *)
  if i in b then
    3WayMerge(m, a2, b, i)
  else
    2WayMerge(m, a2, i)
  end
end Merge;
```

Fig. 8 Main merge procedure

The merge algorithm performs a **recursive descent** through the abstract syntax trees of the input revisions. During the merge process, the current focus is an increment i which is identified by its tag and occurs at least in m and a_2 (fig. 8). If i also occurs in b, a 3-way merge is performed. Otherwise, 2-way rules are applied which, however, will not further be described in the following.

```
procedure 3WayMerge(var m : Rev; a2, b : Rev; i : Tag);
begin
  (* increment i is in m, a2 and b *)
  case Class(m, i) of
    Placeholder : return;
    | Identifier : 3WayMergeIdentifiers(m, a2, b, i);
    | Structure : 3WayMergeStructures(m, a2, b, i);
    | List : 3WayMergeLists(m, a2, b, i)
  end
end 3WayMerge;
```

Fig. 9 3-way merge procedure

The operations which have to be performed depend on the **class** of the current increment (fig. 9). If i is a placeholder – i.e. an unexpanded increment –, nothing has to be done at all. Otherwise, one of the procedures is called which handle the merging of identifiers, structures, and lists, respectively.

```
procedure 3WayMergeIdentifiers
          (var m : Rev; a2, b : Rev; i : Tag);
  var d : boolean;
begin
  if Name(m, i) ≠ Name(a2, i) then
    if Name(m, i) = Name(b, i) then
      (* i was only changed in a2 *)
      ExtendIdentifier(m, i, Name(a2, i))
    elsif Name(a2, i) ≠ Name(b, i) then
      (* conflict : i was changed in both
         alternative revisions *)
      UserDecision(d);
      if d then
        (* the user has chosen the name in a2 *)
        ExtendIdentifier(m, i, Name(a2, i))
      end
    end
  end
end 3WayMergeIdentifiers;
```

Fig. 10 3-way merging of identifiers

In case of **identifiers**, their name attributes are compared (fig. 10). If the name was only changed in a_2, this modification is also applied to m (see also fig. 5). In the symmetric case, no action is required because i already carries the desired name in m. Finally, a user interaction takes place if the names were changed in both alternatives. This interaction is modeled by the procedure UserDecision which returns a boolean value that determines the desired name.

```
procedure 3WayMergeStructures
          (var m : Rev; a2, b : Rev; s : Tag);
  var  d : boolean;
       n : cardinal;
       cm, c2, cb : Tag;
begin
  (* corresponding components of s are compared
     based on their tags *)
  for n := 1 to NoOfSons(m, s) do
    cm := Son(m, s, n);
    c2 := Son(a2, s, n);
    cb := Son(b, s, n);
    if cm = c2 then
      (* merging proceeds recursively *)
      Merge(m, a2, b, cm)
    elsif cm = cb then
      (* the component was replaced in a2 *)
      ReplaceIncrement(m, cm, a2, c2)
    elsif c2 ≠ cb then
      (* conflict : the component was changed
         in both alternative revisions *)
      UserDecision(d);
      if d then
        (* the user has chosen the component
           in a2 *)
        ReplaceIncrement(m, cm, a2, c2)
      end
    end
  end
end 3WayMergeStructures;
```

Fig. 11 3-way merging of structures

In case of **structures**, corresponding components are compared based on their tags (fig. 11). If corresponding components in m and a_2 have the same tags, merging proceeds recursively. If a component was only replaced in a_2, it is copied into m. This is performed by the procedure ReplaceIncrement which is language independent and relies on the general interface provided by class Document[5]. Note that the copy operation preserves the tags. This is particularly important for subsequent merge operations (i.e. merging of m with another alternative revision). In the symmetric case – the component was only replaced in m –, no action is required. Finally, a user interaction takes place if the components were replaced in both alternatives.

5. For the sake of brevity, the body of this procedure has been omitted.

```
procedure 3WayMergeLists(var m : Rev; a2, b : Rev; l : Tag);
  var  em, e2 , e : Tag;
       d : boolean;
begin  (* the lists are compared based on the tags of
          their elements. *)
  em := First(m, l);  e2 := First(a2, l);
  loop
    if em = nil then
      if e2 = nil then
        exit
      elsif not (e2 in b) then
        (* e2 in a2, but not in b => e2 was inserted into a2
           and therefore is inserted into m *)
        InsertAtEnd(m, a2, l, e2); e2 := Next(a2, e2)
      else
        (* e2 in a2 and b, but not in a1 => e2 was
           removed from a1 and is not inserted into m *)
        e2 := Next(a2, e2)
      end;
    elsif e2 = nil then
      if em in b then
        (* em in b, but not in a2 => em was removed from a2
           and therefore is removed from m *)
        e := Next(m, em); Delete(m, em); em := e
      else
        (* em in m (= a1), but not in b =>
           em was inserted into a1 and remains in m *)
        em := Next(m, em)
      end
    elsif em = e2 then
      (* merging proceeds recursively *)
      Merge(m, a2, b, em);
      em := Next(m, em); e2 := Next(a2, e2)
    elsif em in a2 then
      if not (e2 in b) then
        (* e2 not in b, but preceeds em in a2 =>
           e2 was inserted into a2 before em and therefore
           is inserted into m *)
        PreInsert(m, em, a2, e2); e2 := Next(a2, e2)
      else
        (* e2 in a2 and b, but not in a1 => e2 was removed
           from a1 and therefore is not inserted into m *)
        e2 := Next(a2, e2)
      end;
    elsif em in b then
      (* em in b, but not in a2 => em was removed from a2
         and therefore is removed from m *)
      e := Next(m, em); Delete(m, em); em := e
    elsif e2 in m then
      (* em neither in b nor in a2 , and em before e2 in m =>
         em was inserted before e2 into a1 and remains in m *)
      em := Next(m, em)
    elsif e2 in b then
      (* e2 in b, but not in m => e2 was removed from a1
         and therefore is not inserted into m *)
      e2 := Next(a2, e2)
    else (* em only in m, e2 only in a2 =>
            conflict : em and e2 were inserted at the
            same position into a1 and a2, respectively *)
      UserDecision(d);
      if d then
        (* the user has decided that e2 comes first *)
        PreInsert(m, em, a2, e2);
        em := Next(m, em); e2 := Next(a2, e2)
      else (* the user has decided that em comes first*)
        PostInsert(m, em, a2, e2);
        em := Next(m, e2); e2 := Next(a2, e2)
      end;
    end
  end
end 3WayMergeLists;
```

Fig. 12 3-way merging of lists

The most complicated part of the merge algorithm is the procedure for merging **lists** (fig. 12). Again, the comparison is based on tags. We assume that common list elements appear in the same order in all input revisions. This assumption is met as long as only the change operations listed at the end of section 2 are applied (i.e. no Cut & Paste). The body of the merge procedure consists of a loop which iterates over all elements in m and a_2. Common elements are merged recursively, elements deleted in a_2 are also removed from m, and elements inserted into a_2 are also inserted into m. A conflict arises if elements were inserted at the same position into different alternatives. Here, the user determines the order of the elements. Due to the lack of space, we can't discuss the algorithm in detail. However, the interested reader should be able to go through it on his (her) own.

4 Context-sensitive Merging

The merge algorithm presented in section 3 does not take any context-sensitive relations into account. In this section, we describe an extension which deals with **binding of identifiers** to their declarations. The description is informal: Due to the lack of space, we have to refrain from presenting the algorithm in a formal way. Our goal is to make the merge algorithm more intelligent without sacrificing its language independence. This goal is achieved to a certain degree: The extension does not refer to a specific language, but it is currently confined to a standard model of binding.

As already stated in section 2, bindings of identifiers are represented by context-sensitive edges which are maintained incrementally according to the modifications of the abstract syntax tree. The extension presented below relies on the maintenance of bindings which is provided by each document class. It solely uses general resources provided by the class Document (see fig. 4); therefore, it is **language independent**.

However, the extension only works properly if the model of binding is sufficiently simple. We assume a **block-structured language** with the following properties:

- Each declaration is associated to one block.
- Identifiers which are declared in the same block have to be pair-wise distinct (no overloading).
- Applied occurrences must not depend on each other (e.g. no references to record components).
- Declaring and applied occurrences must be disjoint (e.g. no import clauses : an import is an applied occur-

rence with respect to the corresponding export and a declaring occurrence with respect to its uses in the importing block).

Furthermore, we assume that all input revisions are correct with respect to the binding rules (no multiple declarations, no undeclared identifiers). This assumption simplifies the following discussion. However, note that the following considerations may be extended so that errors in the input revisions can also be taken into account.

The **shortcomings of context-free merging** are illustrated by means of the following example (fig. 13)[6]: A Modula-2 program that reads a number and outputs its factorial[7] was modified in two ways:

- $b \rightarrow a_1$: In the body of the procedure Fac, an extra case was added (lines 10, 11) to avoid one recursive call.
- $b \rightarrow a_2$: The procedure was turned into a function (lines 5, 9, 15, and 21). Furthermore, the name of the input parameter was replaced with Argument (lines 3, 7, and 15).

With respect to the context-free structures, the changes do not interfere. Thus, context-free merging proceeds without user interactions. However, the result is not what we desire: The merge algorithm fails to apply the changes of the procedure heading to the extension of the IF-statement in its body. Furthermore, the situation is extremely bad because the merge result does not contain context-sensitive errors: The applied occurrences of Input and Result (lines 10, 11) are bound to the declarations in the enclosing block (line 2). Thus, the inadequacy of the merge result might turn out only at run-time. Note that the program crashes only when the user inputs the value 1. In general, such a rarely occurring error might cause considerable trouble when it creeps into a large and complex software system.

The context-sensitive extension presented below deals with **changes to the bindings of identifiers**. In our example, it detects the problems in lines 10 and 11 and partially solves them:

- The applied occurrence of Input (line 10) is a new one which comes from a_1. In a_1, it was bound to the parameter declaration in line 3. From these facts, the algorithm infers that the applied occurrence should be bound to the same declaration in m. To achieve this goal, it automatically changes the applied occurrence from Input to Argument.
- Similarly, the algorithm detects that the applied occurrence of Result (line 11) is bound to different declarations in a_1 and m. In this case, it is not possible to make the applied occurrence refer to the original declaration. Therefore, the algorithm produces a warning which indicates a potential problem to the user. Note that error correction would require language-specific knowledge (the assignment has to be turned into a RETURN-statement).

In order to extend the context-free algorithm, we adopt a **3-phase approach** :

1. The context-free algorithm is run. Thereby, list and structure rules are applied as before. Identifier rules are restricted so that they are applied only to declaring occurrences.
2. Name clashes that were introduced by phase 1 are removed in order to enable the unique binding of applied occurrences in phase 3. This is done interactively either by renaming or by deleting declarations.
3. Finally, applied occurrences in m are handled as described below.

In phase 3, for each applied occurrence a in m its **expected binding** is inferred from its bindings in the input revisions. The inference rules are summarized in fig. 14. The entries have the following meaning:

- "-" denotes the absence of a.
- "*" denotes a wildcard (don't care condition).
- "$a \rightarrow d$" means that a is bound to d.

The expected binding is compared with the actual binding in m. **Actions** are required in the following **situations**:

- The expected binding is unique, but the actual binding differs from the expected binding. In this case, the merge algorithm tries to modify the applied occurrence so that it is bound as expected. In case of failure, a warning is issued.
- The expected binding is not unique. If the applied occurrence may be bound to both declarations, the user has to perform a selection. If it may be bound to exactly one of the expected declarations, the algorithm ensures that the applied occurrence is bound to this declaration. Otherwise, a warning is issued.

6. As in fig. 7, insertions and changes are shown in bold and italic face, respectively. Deletions are not illustrated explicitly; however, they may be inferred from the line numbers.
7. Input and output statements were simplified to keep the program short.

base revision b:

```
1  MODULE Example;
2      VAR Input, Result : CARDINAL;
3      PROCEDURE Fac(Input : CARDINAL;
4                        VAR Result : CARDINAL);
6      BEGIN
7          IF Input = 0 THEN
8              Result := 1
12         ELSE
13             Fac(Input-1, Result);
14             Result := Input * Result
16         END
17     END Fac;
18 BEGIN
19     Read(Input);
20     Fac(Input, Result);
22     Write(Result);
23 END Example.
```

alternative revision a_1:

```
1  MODULE Example;
2      VAR Input, Result : CARDINAL;
3      PROCEDURE Fac(Input : CARDINAL;
4                        VAR Result : CARDINAL);
6      BEGIN
7          IF Input = 0 THEN
8              Result := 1
10         ELSIF Input =1 THEN
11             Result := 1
12         ELSE
13             Fac(Input-1, Result);
14             Result := Input * Result
16         END
17     END Fac;
18 BEGIN
19     Read(Input);
20     Fac(Input, Result);
22     Write(Result);
23 END Example.
```

alternative revision a_2:

```
1  MODULE Example;
2      VAR Input, Result : CARDINAL;
3      PROCEDURE Fac(Argument : CARDINAL)
5                        : CARDINAL;
6      BEGIN
7          IF Argument = 0 THEN
9              RETURN 1
12         ELSE
15             RETURN Argument * Fac(Argument-1)
16         END
17     END Fac;
18 BEGIN
19     Read(Input);
21     Result := Fac(Input);
22     Write(Result);
23 END Example.
```

merge revision m:

```
1  MODULE Example;
2      VAR Input, Result : CARDINAL;
3      PROCEDURE Fac(Argument : CARDINAL)
5                        : CARDINAL;
6      BEGIN
7          IF Argument = 0 THEN
9              RETURN 1
10         ELSIF Input =1 THEN
11             Result := 1
12         ELSE
15             RETURN Argument * Fac(Argument-1)
16         END
17     END Fac;
18 BEGIN
19     Read(Input);
21     Result := Fac(Input);
22     Write(Result);
23 END Example.
```

Fig. 13 Example for the shortcomings of context-free merging

alternative a_1	*alternative a_2*	*base b*	*expected binding in m*
$a \rightarrow d$	-	-	$a \rightarrow d$
$a \rightarrow d$	$a \rightarrow d$	*	$a \rightarrow d$
$a \rightarrow d_1$	$a \rightarrow d_2$	-	$a \rightarrow d_1 \vee a \rightarrow d_2$
$a \rightarrow d_1$	$a \rightarrow d_2$	$a \rightarrow d_3$	$a \rightarrow d_1 \vee a \rightarrow d_2$
$a \rightarrow d_1$	$a \rightarrow d_2$	$a \rightarrow d_1$	$a \rightarrow d_2$

Fig. 14 Inference of expected bindings

The reader may easily verify that the algorithm behaves as desired when it is applied to the example of fig. 13. Furthermore, note that is does not guarantee that the merge revision is correct with respect to the binding rules. For example, in fig. 13 the applied occurrence of Result (line 11) would be unbound if the variable in line 2 had been named differently (e.g. Output).

The model of binding that we have presumed in this section is rather restricted. We believe that the restrictions may further be relaxed. For example, dependencies between applied occurrences (e.g. WITH–statements in Modula–2) could be handled by a topological sort in phase 3. However, there are inherent limitations. For example, overloading in Ada would require intertwining of phases 2 and 3.

5 Relation to Other Work

Merge tools for **text files** have been used in tool–kit environments for several years (e.g. in RCS /Ti 85/, DSEE /LCS 88/, and SunPro /AGM 86/). These tools use line–oriented algorithms which compute deltas without relying on line tags (e.g. /HS 77, He 78/). The latter is essential in a tool–kit environment because it allows to use arbitrary tools for the manipulation of text files. Merge tools for text files do a good job in tool–kit environments. However, they are not suited for structure–oriented environments because they do not even guarantee the context–free correctness of the merge result.

On the other side of the spectrum, there are a few approaches which refer to **specific languages** and aim at putting much more intelligence into the merge algorithm. For example, in /HPR 89/ and /YHR 90/ approaches to merging programs are presented which rely on a sophisticated data and control flow analysis and take semantic changes into account. In both cases, the underlying programming language is severely restricted. An approach to semantic merging of applicative programs is described in /Be 86/. All references cited so far deal with programs. By way of contrast, /Fe 89/ deals with merging of specifications rather than programs.

To the best of our knowledge, merging of abstract syntax trees has not been addressed before in the literature. However, there are at least a few approaches to computing **tree deltas** in the absence of tags /ZS 89, Ta 79, Se 77/. First, these approaches are not applicable in our structure–oriented environment because they refer to other kinds of trees and edit operations. Second, we argue that node tags should not be computed a posteriori when the merge tool is invoked. A posteriori computations require complex heuristics which yield less precise results than tags maintained by the editor. The price we pay is that all edit operations have to be performed within our environment. However, this is consistent with our general philosophy: software developers benefit from using an integrated structure–oriented environment because of its improved functionality.

6 Conclusion

We have presented a merge algorithm with the following properties:

- It is applicable to software documents written in **arbitrary languages.**
- It is **structure–oriented.**
- It preserves the **context–free correctness** and detects **context–free conflicts.**
- In addition to the context–free syntax, it also takes **bindings of identifiers** to their declarations into account. In general, it does not preserve the correctness of all bindings, but it supports the user by automatically renaming applied occurrences and by detecting anomalies and conflicts with respect to the change of the bindings.

A **prototype implementation** of the context-free merge tool was built into the IPSEN environment. The merge tool consists of about 3,300 loc Modula-2; the IPSEN environment as a whole is a medium-sized system of about 150,000 loc.

In addition to implementing the context-sensitive rules as well, **future work** will address the following problems:

- The context-free algorithm has to be extended so that **Cut & Paste** operations are taken into account. Presently, we assume that increments do not change their positions.
- In addition to syntax-directed operations, IPSEN editors also support **free text input** of arbitrary increments. Currently, free input is not realized incrementally. First, the corresponding subtree is completely deleted; afterwards, it is rebuilt from scratch. Free input has to be made incremental so that only the actual changes are applied to the syntax tree.
- The model of binding which is assumed in the context-sensitive extension of our merge algorithm has to be generalized.

Acknowledgements

The author is indebted to M. Nagl (leader of the IPSEN project), A. Schürr, and U. Schleef for constructive and helpful comments on the initial version of this paper. Furthermore, the reviewers have also provided some valuable remarks on this paper. Finally, the auhor thanks C. Weigmann for the implementation of the context-free merge tool.

References

/AGM 86/ E. Adams, W. Gramlich, St.S. Muchnick, S. Tirfing : SunPro - Engineering a Practical Program Development Environment, Proceedings of the International Workshop on Advanced Programming Environments, Trondheim 1986, LNCS 244, 86-96

/Be 86/ V. Berzins : On Merging Software Extensions, Acta Informatica, vol. 23 (1986), 607-619

/CI 84/ R.D. Cameron, M.R. Ito : Grammar-Based Definition of Metaprogramming Systems, ACM Transactions on Programming Languages and Systems, vol. 6-1 (January 1984), 20-54

/En 86/ G. Engels : Graphs as Central Data Structures in a Software Development Environment (in German), VDI Verlag

/Fe 89/ M. Feather : Detecting Interference when Merging Specification Evolutions, Proceedings of the 5th International Workshop on Software Specification and Design, 169-176

/He 78/ P. Heckel : A Technique for Isolating Differences Between Files, Communications of the ACM, vol. 21-4 (April 1978), 264-268

/HS 77/ J.W. Hunt, T.G. Szymanski : A Fast Algorithm for Computing Longest Common Subsequences, Communications of the ACM, vol. 20-5 (May 1977), 350-353

/HPR 89/ S. Horwitz, J. Prins, T. Reps : Integrating Non-Interfering Versions of Programs, ACM Transactions on Programming Languages and Systems, vol. 11-3 (July 1989), 345-387

/LCS 88/ D.B. Leblang, R.P. Chase, H. Spilke : Increasing Productivity with a Parallel Configuration Manager, in: J. Winkler (Ed.) : Proceedings of the International Workshop on Software Version and Configuration Control 1988, Teubner Verlag, 21-38

/Na 90/ M. Nagl : Characterization of the IPSEN Project, Proceedings of the International Conference on System Development Environments & Factories, May 9-11, 1989, London, Pittman (1990)

/RT 88/ Th. Reps, T. Teitelbaum : The Synthesizer Generator, Springer-Verlag

/Se 77/ S.M. Selkow : The Tree-to-Tree Editing Problem, Information Processing Letters, vol.6-6 (December 1977), 184-186

/Ta 79/ K.-C. Tai : The Tree-to-Tree Correction Problem, Journal of the ACM, vol. 26-3 (July 1979), 422-433

/Ti 85/ W.F. Tichy : RCS - A System for Version Control, Software : Practice and Experience, vol. 15-7 (July 1985), 637-654

/We 89a/ B. Westfechtel : Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control, Techn. Report, Aachen University of Technology

/We 89b/ B. Westfechtel : Revision Control in an Integrated Software Development Environment, in: J. Winkler (Ed.) : Proceedings of the 2nd International Workshop on Software Configuration Management, ACM Software Engineering Notes, vol. 14-7 (November 1989), 96-105

/We 91/ B. Westfechtel : Revision Control in an Integrated Software Development Environment, Ph.D. thesis, Aachen University of Technology

/YHR 90/ W. Yang, S. Horwitz, T. Reps : A Program Integration Algorithm that Accomodates Semantics-Preserving Transformations, Proceedings of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments, ACM Software Engineering Notes, vol. 15-6 (November 1990), 133-143

/ZS 89/ K. Zhang, D. Shasha : Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems, SIAM Journal on Computing, vol. 18-6 (December 1989), 1245-1262