Operation-based Merging

Ernst Lippe
Software Engineering Research Centre, University of Utrecht
Norbert van Oosterom
Software Engineering Research Centre
e-mail: lippe@serc.nl, oosterom@serc.nl

Abstract

Existing approaches for merging the results of parallel development activities are limited. These approaches can be characterised as state-based: only the initial and final states are considered. This paper introduces operation-based merging, which uses the operations that were performed during development. In many cases operation-based merging has advantages over state-based merging, because it automatically respects the data-type invariants of the objects, is extensible for arbitrary object types, provides better conflict detection and allows for better support for solving these conflicts. Several algorithms for conflict detection are described and compared.

1 Introduction

Parallel activities, where each user has private copies of shared data, occur frequently in many different kinds of computer supported cooperative work. After a certain period the results of these separate lines of development must be integrated. In this paper this process is called merging. Software development, where different versions of a program source must be integrated, is an important example, but merging problems arise in one form or another in all forms of cooperative work. In general, merging is a complex process, certainly when many interrelated objects are involved. Merging depends on the semantics of the objects that are involved: merging two program texts is different from merging pictures.

Tool support for this merging process is desirable, but hardly existent. This paper describes the approach to merging that is used in the **CAMERA** system [Lip92, FLD+92], a support system for cooperative work in loosely coupled distributed environments. **CAMERA** contains an extensible Object Management System (OMS) that belongs to the group of object oriented databases. Techniques were needed to merge developments that were performed on this database. Because new data-types can be added by defining new classes the approach has to be extensible.

Existing approaches to merging are too limited for this purpose. The next section of this paper give an overview of existing merge tools. Existing merge techniques can be characterised as being state-based, they only use the initial and final state of a development. Furthermore, these techniques can only be applied to a limited set of data-types.

In this paper we introduce a new approach called *operation-based merging*. Operation-based merging is based on the operations that were performed during the developments. Operation-based merging can in principle be applied arbitrary abstract data-types, and guarantees that data-type invariants are respected. Operation-based merging is introduced in section 4. Later sections describe algorithms for detecting and solving conflicts based on this approach.

ACM-SDE-12/92/VA, USA

2 Existing merge tools

Several systems are available that support some form of merging. These systems mainly support merging of single text files and are described in section 2.1. Merge tools providing support for more complex components are dealt with in section 2.2. Unfortunately, most existing merge tools are not very satisfactory. A list of deficiencies is identified in section 2.3.

2.1 Text-based merge tools

Merging of ASCII text files is supported by several systems, e.g. `RCS` [Tic85], Sun's `filemergetool` [AGMT86], and

DSEE [LC84]. All these systems are line-based, and attempt to detect common lines and lines that have been inserted/deleted or moved. They do so by finding the longest common subsequences between the texts to be merged.

These tools either use *two way merging*, which attempts to merge the two final versions, or *three way merging* that also uses the common ancestor of the versions to be merged. Three way merging is far more powerful than two-way merging because more information is available. If a line is only available in one of the versions, some two way merge algorithms assume that this line has been added and by default include it in the final merge result. This however is not always the desired behaviour. When the line has been deleted in one of the final versions (but not in the other) the most appropriate default behaviour is to delete the line in the merged version. Three way merges are able to detect deletions, while two way merges obviously cannot. Furthermore, if a certain line of a file has different values in both versions, two way merges cannot make a decision which one to choose. Three way merges can detect whether one of the values is equal to the value of the line in the original file, and use the other value by default in the merged version.

In Tandem's version control system [SK90] individual text lines are tagged with a unique identification and recorded in a database. Modifications create new lines with different identifications. These tags are used to detect insertions/deletions, and this information is then be used to perform merges. The algorithm currently does not seem to handle the operation of moving lines to a different position.

2.2 Other merge tools

The combination of the Unix `diff` program and Larry Wall's `patch` [Wal88] can be used to merge sets of text files. `diff` is used to generate context diffs that list both the original and the changed lines plus a small number of surrounding context lines. `patch` can then be used to apply the diffs to another file hierarchy. `patch` has a certain amount of flexibility. The files to which it is applied need not be identical to the versions that were used to compute the diff. Differences occur if the receiver has already modified the original file. When `patch` cannot find the original lines at the location it expects, it examines the neighbourhood of this location. If matching lines are found, it applies the delta at this modified location. This approach only works if the lines at this location have not been modified themselves. Although, `patch` is useful in many practical circumstances, it is still rather limited. When deltas cannot be applied they must be handled manually.

Horwitz and Reps [HPR88, HPR89] have developed a method for merging programs in a very simple language that is based on the semantics of this language. This approach is very interesting but unfortunately only works for a limited language. It is not yet clear if this approach can be extended to handle "real" programming languages.

Sun's NSE [CFH88] provides merging of environments. An environment contains a set of files plus some specialised objects. During the merging process NSE compares the contents of the environments and it invokes a file merge tool for merging individual text files. NSE also contains small databases, essentially no more than a list of ⟨key,value⟩ pairs. NSE provides tools to merge these.

Westfechtel [Wes91] has developed a method for merging trees. This method can be used to merge programs (represented as parse trees). This method differs from simpler methods because it can handle certain contextual dependencies, e.g. it tracks the declaration and use of identifiers. Thus it is able to merge developments where identifiers have been renamed.

The PACT merge tool [PAC89] provides support for merging composite objects in PCTE [ECM90]. It is used to merge composite objects that have a similar structure, described by the same merging template. It only merges structural links, and cannot be used to merge the contents of the atomic objects that form the composite object.

2.3 Deficiencies

The main deficiencies of these systems can be summed up as follows:

- Line based tools particularly suffer from the following problems:
  - They cannot handle multiple changes within a single line. Thus if a line has been modified in both lines of development, only one of the versions can be selected. The other version must be integrated manually. Since some tools do not support edit operations while merging, this requires an explicit separate bookkeeping by the users, which is a cumbersome and error-prone process.
  - There are several operations that change the contents of many lines, without having a real semantic meaning. Examples are the pretty-printing of a program and the reformatting of a text to give all lines a similar length. In this case merging tools flag many lines as changed and detect many conflicts.
  - A similar problem exists when in a program the indentation is changed, e.g. when an if statement is added around an existing sequence of statements. The indentation of all lines is changed, and therefore all these lines are regarded as changed during the merge.
- The merging applies only to a limited set of types. Most merge tools are limited to merging texts. Furthermore, merge tools for merging structured sets of objects are hardly available. The PACT merge tool handles the relationships between arbitrary types of objects but cannot merge the contents of any object type.
- These merge-tools do not offer much help when conflicts are detected. In most cases when a conflict arises the user must choose a value from one of the lines of development and if necessary perform manual editing afterwards.

- The merge-tools are state-based in the sense that they use no information about actual developments, but only information about the initial and final states. This can lead to unnecessary conflicts, conflicts that go unnoticed, and to incorrect results. An example of the first conflict occurs when in both lines of development the resulting source is pretty-printed. Even when the "real" modifications do not conflict, pretty-printing leads to conflicts. An example of the second type is when one user changes the name of a procedure while the other adds a new call to this procedure. After merging there is one call with the old procedure name. An example of the third is a counter for the total time that has been spent by programmers on a project. In each line of development programmers will regularly invoke an operation to increment the local value of the counter with the time that has been spent in this line of development. State based merges will not detect a conflict when an equal amount of time that has been spent in each line of development, but will simply use the resulting value of the counter. When the amount of time differs, state-based merges will offer the user the selection between one of the two final values, but not the option to add the sum of the increments, which would be the correct answer.
- Most merge-tools offer hardly any support for merging modifications that change the structure among objects. For example, none of the file-based merge systems appears to be able to automatically merge developments where files are moved to a different directory.

3 Design goals

The design goals for a merge procedure for the **CAMERA** system were the following:

- The technique should give flexible support for different object types, thus not be restricted to merging a specific class of objects. It should also be possible to merge instances of user defined classes.
- Detect as many conflicts as possible, and attempt to resolve them automatically. Extensive support is more important than execution speed. Merging is a difficult process where every bit of support is helpful. Furthermore, merging does not occur very frequently.
- Completely automatic merging is not possible due to the only partially known semantics of objects and changes. Therefore user intervention cannot always be avoided. The merge tool must have a good user interface, in order to present conflicts and to show potential solutions. Furthermore, the number of decisions on how conflicts are to be solved must be minimised.

4 Operation-based merging

All existing merge tools are state-based. The operations that have been performed in each line of development are ignored: only the end-results count. The previous sections have shown the problems with the state-based approach.

We therefore propose a different approach based on the operations that were performed. In **CAMERA** these operations are recorded in the form of *transformations*. Transformations are composed of sequences of OMS-operations that are called *primitive transformations*. Transformations are treated as functions, that take an initial state of the OMS as input and return a new OMS state as result. Thus $T_a(s)$ is the state that is the result of a line of development whose operations are recorded in T_a when the starting point was the state s.

An important notion in operation-based merging is commutation of operation, that is defined as follows. Given an equivalence relation (i.e. a reflexive, symmetric, and transitive relation) $\approx$, we define two operations T_a and T_b to *commute locally* on an input s if and only if $T_a(T_b(s)) \approx T_b(T_a(s))$. In a similar way we define that two operations T_a and T_b *commute globally* if and only if $\forall x : T_a(T_b(x)) \approx T_b(T_a(x))$.

Basic assumption for operation-based merging: When two transformations commute locally on their initial state, the final result is a good candidate for the result of the merge.

Using an equivalence relation instead of equality gives added flexibility, that allows ignoring differences between values that are not important from the user's perspective. In an implementation of an abstract data-type two values that are not bit-wise equal can represent the same abstract value. In a linked-list implementation of the abstract data-type "set" several different lists can represent the same abstract set value. In this case using the equality on the implementation values is not appropriate. The equivalence relation on the implementation values that represents the data-type equality on the corresponding abstract values should probably be used instead. In some cases, (see section 7 for an example), users may even want to decide that two different abstract values can be considered equivalent. In most cases the preferred equivalence relation will simply represent the equality of the abstract data-type.

Handling conflicts

In many cases two transformations commute locally. For example, if T_a and T_b operate on disjoint sets of objects, operation-based merging integrates the transformations without conflicts.

However, in other cases transformations do not commute locally. In these cases we cannot merge automatically and some form of user intervention will be necessary. One possibility is to allow the user to edit the two transformations until they do commute. However, it is possible to offer more assistance, by examination of the primitive transformations constituting the transformation. In general, it can be expected

that only a very small subset of these primitive transformations actually conflicts. These sets of conflicting primitive transformations are isolated and presented to the user, together with possible suggested solutions. The details of this process are described in section 5 and further.

Solving conflicts

When a set of conflicting primitive transformations has been detected these conflicts must be solved by the user. The merge tool illustrates the conflict by showing the state of the OMS before application of the transformations, the two sets of conflicting primitive transformations (one set for each of the two original transformations), and the two states that are the results when these sets are applied in different orders. The user can select one of the following approaches:

- Resolve conflicts by *imposing an ordering* on the primitive transformations. For example, if a pretty-printer is invoked on a source file in one development line while small changes have been made to the file in the other line, many conflicts can be avoided by first applying the changes and then pretty-printing. Similarly, if an identifier is changed in one line of development (using a global substitute command) and a new use of the old identifier is added in the other line of development, the conflict can be solved by performing the global substitute command last.
- Resolve conflicts by *deleting a primitive transformation*. When multiple calls to a pretty-printer are a source of conflicts, normally all but the last one can be deleted without harm.
- Resolve conflicts by *editing the transformation*. If the previous options are not sufficient it is also possible to modify the transformations by modication of their primitive transformations or the addition of new ones.

The last two solutions may change the existing conflicts. The merge tool needs to check for this situation, and possibly recalculate part of its commutation information.

4.1 Advantages of operation-based merging

Operation-based merging could be attractive because:

- The approach is extensible, it can potentially be applied to arbitrary object types (even user-defined types).
- It can be used to merge entire object systems.
- It can avoid several of the conflicts mentioned before. Multiple updates to the same text line will not automatically lead to conflicts. When as a final step in both lines of development the source is pretty-printed, state-based merging can detect conflicts that are only caused by the pretty-printing. However, when pretty-printing is an idempotent operation (which is very likely), operation-based merging will not flag this as a conflict.
- In some cases it can give more adequate merge-results than a state-based approach. In the example of a time counter (see previous section) that was incremented in two different lines of development, operation-based merging will increase the counter by the sum of the increments.
- It can detect conflicts that are not noted by state-based approaches. As we have seen above, state-based approaches will not notice the potential conflict that arises when the name of a procedure was changed in one line of development and a new call to this procedure was added in the other line of development. In this case operation-based merging will detect a potential conflict, because the result of first substituting the new name of the procedure for the old one and then introducing a new call is different from the result that is obtained when these two operation are executed in the reverse order. It is likely that the user will solve this conflict by ordering these operations so that the introduction is performed first.
- It can offer more support for conflict resolution. A single global operation, such as a global substitution or pretty-printing, can introduce conflicts at many different points. State-based tools will force the user to take a decision about each local conflict (e.g. each conflicting line). When such a conflict is presented the user may not be aware that this conflict was caused by a global operation. With operation-based merging the global operation is presented as one unit. Only one central decision is needed (e.g. to postpone the renaming or to omit the pretty-printing). This decision will be used to solve all local conflicts. Furthermore, with operation-based merging the operation is presented in the context that is similar to the one in which it was originally performed. With state-based merging conflicts are normally presented in the order in which they occur within the object, and it can be difficult to determine in what context these modifications were made.
- The merged result is more likely to be consistent from the user's point of view than with the state-based approach. Implementations of abstract data-types have an internal binary representation that must satisfy certain consistency requirements. In general, it is impossible for state-based merge tools to merge the contents of such an abstract data-type correctly unless the internal binary format is known to the merge tool. However, operation-based merging does not need information about the internal representation because the operations of the data-type respect the data-type invariants of the object.
- Operation-based merging is more a general approach than state-based merging. State-based merging can be seen as a restricted form of operation-based merging, that attempts to reconstruct transformations after the fact. This is difficult, since there usually is more than one function, that can be used to transform an initial state into the final state. With operation-based merging

the operation that is actually used is recorded, so the actual function is known.

5 The merge process

This section describes the merge process.

Every transformation consists of a sequence of elementary steps called primitive transformations. E.g. $T_a X = T_{a,n} T_{a,n-1} \ldots T_{a,1} X$. These are the smallest grain of operations that have been recorded.

Operation-based merging partitions the total set of the primitive transformations from both lines of development into sub-sets called *blocks*. The transformations in a block may conflict with one another, while transformations from different blocks by definition do not.

The algorithm requires as part of its input two decision procedures to determine whether two given primitive transformations commute globally and locally. Section 6 describes some ways commutation can be computed.

The merge process starts with the partitioning of conflicting sets of primitive transformations in blocks using global commutation properties (see section 5.1). The second stap (see section 5.2 uses local commutation properties for identification of conflicts, attempts to solve the conflicts and presents the unsolved conflicts to the user.

5.1 Partioning of conflicts

The first step uses the global commutation decision procedures to partion the primitive transformations (constituting the transformations that are to be merged) in *raw blocks*.

These blocks are constructed by imposing an ordering (actually a pre-ordering), named *before*, on the primitive transformations. The predicate $before(a, b)$ is true if primitive transformation a must be performed before b. The *before* relation is transitive. If two primitive transformations within the same transformation conflict, the one with the lowest index must be performed before the other. Cycles in the *before* relation (i.e. $before(a, b) \wedge before(b, a)$) indicate merge conflicts, because a and b cannot be ordered and a decision must be made. Conflicts between primitive transformations a and b, that are part of different transformations always lead to a cycle.

The *before* relation is used to determine the raw blocks. Blocks form a partitioning of the union of the primitive transformations in such a way that each block is the smallest set such that each cycle is contained within one block. The *before* relation on the primitive transformations can be used to define a partial ordering on the blocks, which by abuse of notation will also be called *before*. The *before* relation between blocks B_1 and B_2 is defined by:

$$before(B_1, B_2) \equiv \exists\, t_1 \in B_1, t_2 \in B2 : before(t_1, t_2)$$

Different blocks can be ordered with respect to one another, but obviously there cannot be any cycles among blocks. Thus, the *before* relation on blocks is a true partial ordering. A single merge conflict occurs within one block, due to the way blocks are constructed.

	T_{a1}	T_{a2}	T_{a3}	T_{b1}	T_{b2}	T_{b3}
T_{a1}		•		•	•	
T_{a2}	•		•			
T_{a3}		•				•
T_{b1}	•					
T_{b2}	•					
T_{b3}			•			

Figure 1: Example conflict matrix

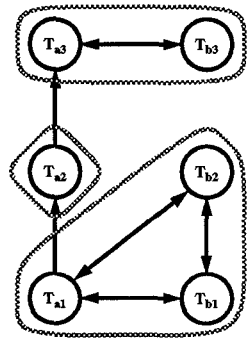


Figure 2: Corresponding blocks for figure 1
Arrows represent the *before* relationships. Grey lines indicate the resulting blocks.

An example is shown in figures 2 and 1. Based on the conflict matrix (figure 1) three different blocks are found (figure 2). The conflicts in the block containing T_{a1}, T_{b1} and T_{b2} must be solved before continuing with the other blocks. The second block contains a single primitive transformation T_{a2}. Such blocks do not pose any merging problems, but their transformations can only be applied after those of its predecessor blocks. In this example the blocks are completely ordered by the *before* relation, however, in general the *before* relation only determines a partial ordering on the set of blocks.

The inputs for step 1 of the algorithm are formed by two transformations plus a procedure to determine whether two primitive transformations commute globally. It attempts to minimise the number of calls to this procedure, because this could be an expensive operation. Some methods to determine whether two operations commute globally are described in section 6.

For the description of the algorithm we use the following notation. The set of primitive transformations is denoted by T. For each primitive transformation t, we will denote the number of the transformations to which it belongs (either 1 or 2) with $t.trn$. Similarly, its index within this transformation will be notated as $t.pos$. The function $conflict(t_a, t_b)$ returns *true* if the primitive transformations t_a and t_b do not commute globally.

Three different algorithms for computing raw blocks are described and compared. The basic algorithm 0 compares all

primitive transformations ($|T|(|T|-1)/2$ pairs) and then computes the transitive closure. The transitive closure can be computed in $O(|T|^3)$ by using the Warshall algorithm ([CLR91]).

Algorithm 0

forall $t_1, t_2 \in T \times T$
  $before(t_1, t_2) := conflict(t_1, t_2) \wedge$
    $(t_1.trn \neq t_2.trn \vee t_1.pos < t_2.pos)$
compute transitive closure of before

A disadvantage of Algorithm 0 is that it determines for all pairs of primitive transformations, whether or not they conflict. This could be an expensive operation and it seems sensible to reduce the number of comparisons as much as possible. This can be done by using the transitivity properties of *before*. When it is known that $before(a, b) \wedge before(b, c)$ it can be deduced that $before(a, c)$. Thus in this case there is no need to compare a and c in order to know whether $before(a, c)$ holds. The following two algorithms use this transitivity property to reduce the number of comparisons.

They further attempt to minimise the number of comparisons by selecting the best candidates for these comparisons. The algorithms differ only in the function $new(i, j)$ that estimates the number of entries that can be filled in the *before* matrix if primitive transformations i and j do not commute. Algorithm 2 computes the exact number of entries while algorithm 1 uses an approximation that only gives an upper-bound for this value.

In the description of the algorithms the following auxiliary functions are used. $A(i, j)$ computes the set of entries of *before* that can be set to *true* when i and j do not commute. $S(i)$ ($L(i)$) computes the set of transformations that are less (greater) than or equal to i according the ordering that is determined by the current *before* matrix. To simplify the description a special operator is introduced. For boolean valued expression c and an integer or set-valued expression v the value of $c \triangleright v$ is v when c is true. Otherwise, its value is 0 when v is integer-valued, or $\emptyset$ when v is set-valued.

Both algorithms maintain a set *uncomputed* of the primitive transformations in both transformations that have not yet been compared. From this set they select the pair $\langle i, j \rangle$ with the highest value of *new*. This pair $\langle i, j \rangle$ is removed from *uncomputed*. If the corresponding primitive transformations could add new entries to the *before* matrix it is computed whether they actually conflict. If this is the case the *before* matrix is updated.

Skeleton for Algorithm 1 and 2

uncomputed :=
  $\{\langle u, v \rangle : T \times T \mid u.trn < v.trn \vee u.pos < v.pos\}$
forall $t_1, t_2 \in T \times T$
  $before(t_1, t_2) := false$
while uncomputed $\neq \emptyset$ *do*
  select $\langle i, j \rangle$ *from uncomputed*
    such that $new(i, j)$ *is maximal*
  uncomputed := *uncomputed* $- \{\langle i, j \rangle\}$
  if $new(i, j) = 0$ *then exit*
  if $conflict(i, j)$ *then*
    foreach $\langle p, q \rangle$ *in* $A(i, j)$
      $before(p, q) := true$

$S(i) = \{t : T \mid before(t, i)\} \cup \{i\}$
$L(i) = \{t : T \mid before(i, t)\} \cup \{i\}$

$A(i, j)$
  $= (\neg\, before(i, j) \triangleright (S(i) \times L(j))) \cup$
    $(\neg\, before(j, i) \triangleright (S(j) \times L(i))$
      when $i.trn \neq j.trn$
  $= (\neg\, before(i, j) \triangleright (S(i) \times L(j)))$
      when $i.trn = j.trn \wedge i.pos < j.pos$
  $= (\neg\, before(j, i) \triangleright (S(j) \times L(i)))$
      when $i.trn = j.trn \wedge i.pos > j.pos$

Algorithm 1

$new(i, j)$
  $= (\neg\, before(i, j) \triangleright (|S(i)|\,|L(j)|)) +$
    $(\neg\, before(j, i) \triangleright (|S(j)|\,|L(i)|)$
      when $i.trn \neq j.trn$
  $= (\neg\, before(i, j) \triangleright (|S(i)|\,|L(j)|))$
      when $i.trn = j.trn \wedge i.pos < j.pos$
  $= (\neg\, before(j, i) \triangleright (|S(j)|\,|L(i)|))$
      when $i.trn = j.trn \wedge i.pos > j.pos$

The function *new* for algorithm 1 is not very precise, in that it only provides an upper-bound on the number of entries that are added to the *before* matrix. A variant is algorithm 2 that uses the actual number of entries that are added. The only essential difference is in the function *new* which omits existing entries in *before* from the count.

Algorithm 2

$new(i, j)$
  $= |\{\langle p, q \rangle \in A(i, j) \mid before(p, q) = false\}|$

The function $new(i, j)$ can be implemented as an array. When a new tuple $\langle i, j \rangle$ is added to *before*, at most those elements $new(u, v)$ must be updated for which $\{i, j\} \cap (S(u) \cup S(v) \cup L(u) \cup L(v)) \neq \emptyset$. Furthermore when $new(u, v) = 0$ there is also no need to recompute *new*.

The selection of the i, j with the highest value of *new* can be implemented by storing all values of *new* in an appropriate data structure (e.g. a heap).

If the same primitive transformations have multiple occurrences in the transformations, it is better to convert *conflict* to a memo function.

Performance of the algorithms

Algorithms 1 and 2 take in worst case $O(n^2)$ global commutation computations (usually better). Algorithm 0 always has $O(n^2)$ global commutation computations and $O(n^3)$ operations to compute the transitive closure.

In order to compare the algorithms a small simulation study was performed. In these simulations a random *conflict* matrix was used in which two elements conflicted with a given probability p (i.e. a fraction of p entries of the conflict matrix was *true*).

The results are shown in the figures 3, 4, and 5.

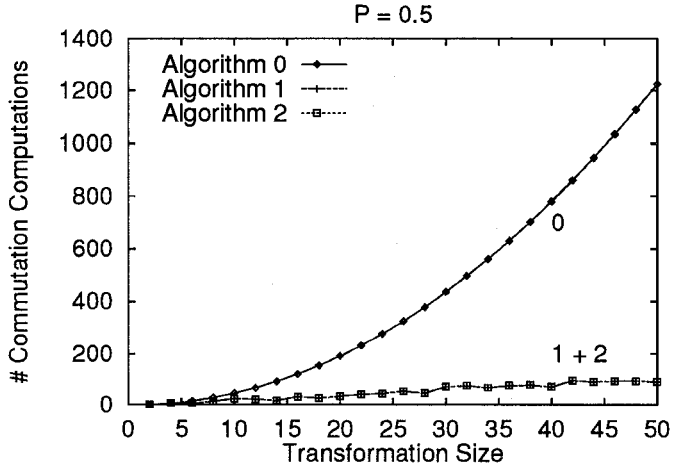


Figure 3: Transformation size versus number of commutation computations
The curves for Algorithms 1 and 2 overlap.

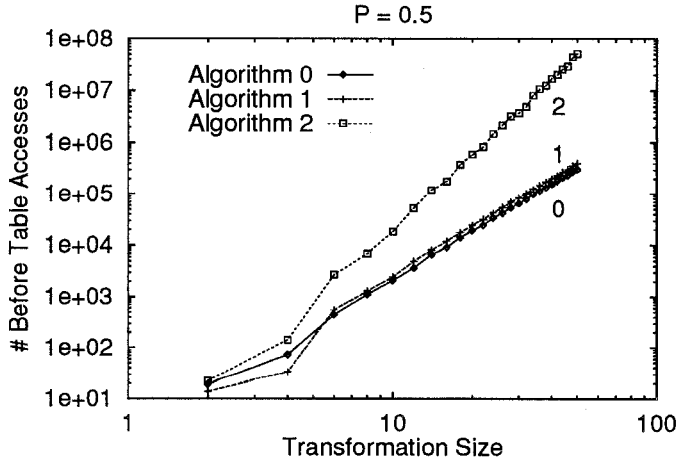


Figure 4: Transformation size versus number of accesses to before array

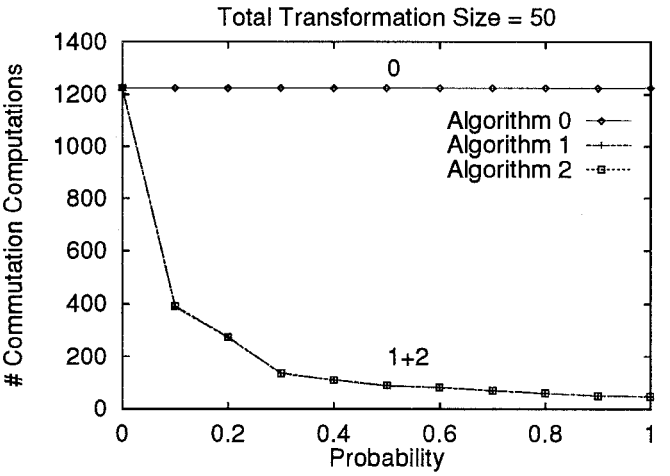


Figure 5: Conflict probability versus number of commutation computations
The curves for Algorithms 1 and 2 overlap.

When *conflict* can be computed cheaply, algorithm 0 is the obvious candidate. However, it can be expected that the cost of these computations is far from negligible. Algorithms 1 and 2 avoid many unnecessary comparisons. Algorithm 2 on the other hand is very expensive, while the complexity of algorithm 1 is similar to that of algorithm 0 ($O(n^3)$).

In summary, when the computation of conflicts between primitive transformations is expensive, algorithm 1 is most suitable. Otherwise, algorithm 0 is to be preferred.

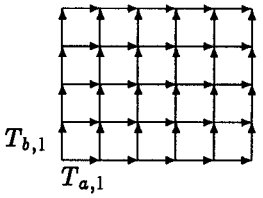
5.2 Identifying and solving conflicts

The second step takes as input the raw blocks that were found in the first step, and locates the conflicts between the primitive transformations in each block. The first step has only used the global commutation behaviour of the primitive transformations. It is well possible that two transformations that do not commute globally nevertheless commute locally for a large set of inputs.

The second step works on one raw block at a time, and determines the primitive transformations that are truly conflicting. This step also uses information about local commutation behaviour of the transformation.

In order to locate conflicts the merge tool examines all *weaves* of the primitive transformations that must be merged. A *weave* (sometimes called interleaving) is a permutation of the union of the primitive transformations in both transformations in which the original order of the primitive transformations is respected. Thus in a weave $T_{a,i}$ must occur before $T_{a,i+1}$.

If we set out a grid with the primitive transformations for T_a on one axis and the primitive transformations for T_b on the other, every weave corresponds to a path in this grid.



A point in this grid represents a set of values, the origin represents the original value X and the set of values at some other point are obtained by applying all transformations, that are represented by paths from the origin to this point, to the original value X. If the set of values at every point contains precisely one value, then the operations T_a and T_b commute. Conversely if T_a and T_b do not commute there is a point that contains more than one value. A point is called *single-valued* if its set of values contains one element, otherwise the point is *multiple-valued.*

The frontier set

Now we can take the set of multiple valued points that have only single-valued points on all paths from the origin to this point, we call this set the *frontier set.* This frontier set gives useful information about which primitive transformations in T_a and T_b actually do conflict. For a frontier point $x_{i,j}$ its value set contains two different values: $T_{a,i}(x_{i-1,j})$ and $T_{b,j}(x_{i,j-1})$. So $T_{a,i}$ and $T_{b,j}$ do not commute on $x_{i-1,j-1}$.

The information about the frontier set can help the user to decide how this conflict should be solved. A list of the decisions that a user can make was given in section 4. The first possible decision is to impose an ordering on the primitive transformations, i.e. to state that $T_{a,i}$ must always be performed before (or after) $T_{b,j}$. In this case the merge tool only considers weaves that satisfy this constraint. The other decisions change the original transformations. When a frontier point is "resolved", a new frontier is defined. This frontier will is computed and presented to the user, until all conflicts have been resolved.

In many cases the primitive transformations in a compound transformation can be performed in a different order without affecting the outcome of the transformation, because the primitive transformations locally commute with each other. This can be used to move the frontier points away from the origin. When the frontier points are far removed from the origin, the user probably needs to make fewer decisions.

6 Computation of commutation behaviour

The algorithms in the previous sections assume that there is some procedure to determine whether two transformations commute (locally or globally). The merge tool needs information about the properties of the primitive transformations. Normally, this information will be supplied by the user when a new abstract data-type is defined.

The decision procedures must be conservative, they must never erroneously declare that there is no conflict. However, they may err on the safe side by declaring that there is a potential conflict even when this is not the case. Depending on the transformations this may lead to spurious conflict warnings by the merge tool.

In general, a more conservative decision procedure is faster than a less conservative one. When we have two primitive transformations that only read and write the attributes of one object, it is safe to say that they commute when they are applied to different objects. Thus a quick and dirty answer of the decision procedure would be to indicate a potential conflict when they are applied to the same object, and to indicate no conflicts when they are applied to different objects. It is possible that the two primitive transformations do not conflict even when applied to the same object, but this will require some additional computations.

In the implementation of these decision procedures there is a trade-off between the execution time and the number of unnecessary conflict warnings.

The remainder of this section describes several approaches for making these decision procedures available to the merge tool.

6.1 Computation of global commutation

There is a trade-off here between the amount of work that is done in the first step of the merge algorithm that uses global commutation information and the second step that also uses local commutation. The global commutation information does not need to be very precise, when that is too expensive: the next step corrects any overly conservative estimates. In general, it seems wise to use a fast but possibly imprecise decision procedure for global commutation, and a more precise procedure for local commutation. When the global commutation procedure is too conservative the next step in the merging is more expensive of course.

There are several ways to determine global commutation information. Of course several of these methods can be combined. The following approaches could be used:

- Using user supplied hook functions. The user can define one or several predefined procedures that determine whether two transformations commute. These are then called by the merging process.
- Using information about *read/write sets.* A read set is the part of the OMS (i.e. a set of object-attributes and relation tuples) that is examined by the operation. Similarly the write set is the part of the OMS that is modified by the operation. Two operations commute globally when the read set of one is disjoint with the write set of the other. One problem here is that it is usually hard to determine the read and write sets independently from the state of the OMS on which the operation is applied.
- Using a special rule base in which a user can declare which transformations commute, or a formal specifica-

tion of the methods from which the system can deduce there commutation behaviour.

6.2 Computation of local commutation

The same procedures as for global commutation can be used. There is one additional possibility:

- Execute the transformations and see whether the resulting OMS is the same.

7 Problems

This section describes some of the problems that may occur when using operation-based merging.

Replaying transformations

"Replaying" transformations can pose problems with respect to the reproducibility of the results. *Well-behaved* primitive transformations are functional with respect to the OMS, i.e. the resulting OMS state is a (mathematical) function of the initial state and the arguments of the transformation. Non well-behaved transformations are not replayable: on different occurrences they may result in different final states of the OMS when applied to the same initial OMS state. Not all operations on the OMS however are well-behaved. There are two main reasons why a primitive transformation is not well-behaved: user input and object creation. The result of a transformation that depends on user input, is not reproducible because the user input may vary from time to time. Object creation in **CAMERA** requires the generation of a unique object identifier. Thus when the same transformation is replayed multiple times, the objects may have a different object identifier each time.

One solution to the problems due to user interaction is by recording the user input, and supplying it as an additional input during the replay of transformations. In the current **CAMERA** OMS this problem has been avoided by making it illegal for methods in the OMS to communicate with the user directly. All communication is performed by tools that are implemented outside the OMS. These tools then send messages to the OMS. The execution of these messages will not involve any user-interaction.

The problem with the creation of object identifiers could potentially be solved in two different ways. The first method registers for a transformation which object identifiers were created when this transformation was originally executed. When the transformation is replayed, objects with the same identifiers will be created. With this method complications arise when the replayed transformations create a different number of objects than they did originally. The second approach uses equivalence of the end-results instead of strict equality (see the definition of commutation in section 4). In this case two OMSes are compared modulo the values of the object identifiers. Two OMSes are considered equivalent if and only if there is a bijective mapping between their object identifiers, that transforms them into each other. The current merge tool uses the latter approach.

Redundant operations

Transformations can contain a number of primitive transformations that are redundant. If for two primitive transformations $T_{a,i}$ and $T_{a,j}$ it holds that $T_{a,j}(T_{a,i}(x)) = T_{a,j}(x)$ the primitive transformation $T_{a,i}$ can safely be omitted from the transformation. Examples of redundant transformations are operations on objects that are deleted later on, and update operations whose results are overwritten by later updates. Removing redundant transformations is attractive because it:

- speeds up conflict detection
- removes unnecessary conflicts

In certain applications redundant operations may occur frequently. A user may first try out one approach and then decide to follow a different one. In this case the modifications that were specific to the first approach become redundant. For this type of applications a pre-processing phase could be added to operation-based merging to remove redundant operations.

Reusability of operations

Operation-based merging depends on the re-usability of operations. The recorded operations must be replayable on other OMSes and should adequately represent the user's intention. Different operations may have the same result in the original OMS, but behave differently under other circumstances. An edit operation on a text file can be modelled in different ways. The operation can be modelled as a value replacement of the entire file with its new value. This approach gives unsatisfactory results with respect to merging. Edit operations on the same object in different lines of development always lead to conflicts. A slightly better approach treats a file as a sequence of characters and uses operations that insert or delete bytes at certain positions in this file. Another approach uses operations that are based on modifications to lines in this file. This is the level at which most text merge tools operate. A more sophisticated approach uses a richer set of operations that also use the structure of the text. A text may have an internal structure, e.g. a division in chapters and sections, or an even more complicated structure as a program source text. Operations that use this structure have a higher degree of reusability, e.g. it is much more useful to record that a user has replaced the second paragraph of section 1.2 than to record that some bytes at a specific offset in the file have been modified.

The requirement that operations must be re-usable imposes a certain overhead on users, since they may have supply extra information and/or have to define new operations. For example, take the operation that moves a text paragraph A to a different location between two other paragraphs B and C. The user's intention may have been to add A immediately after B or immediately before C. For the concrete version that the user's sees during this operation, there is no difference between these cases. Thus potentially this operation could be represented by the primitive transformations that inserts

A after B or by the one that inserts A before C. However, these two primitive transformations will behave differently during merging. When in the other line of development B or C are moved or when a new paragraph is inserted between them, the end-result will depend on the choice of the primitive transformation. In order to obtain optimal results with operation-based merging both choices should be available to the user. Instead of one operation to move paragraphs there should be two slightly different ones (insert before and insert after). The selection of the correct operation puts an extra load on the user.

8 Conclusions

Existing merge tools are either very crude, or very sophisticated but of limited applicability. Operation-based merging is an attractive alternative. It uses semantic knowledge about the objects and their operations. Additionally, it is extensible, new object types and operations can be added and these can then be merged as well. Because operation-based merging uses the operations of a data-type it automatically maintains its data-type invariants.

We believe that the concept of operation-based merging provides an elegant framework for attacking the problem of merging different lines of development. The main slogans of this paradigm are: "merging = composition of operations from each development line" and "merge-conflict = commutation conflict". Operation-based merging is certainly no panacea for all merge problems. For a fixed and well known domain it is certainly possible to construct better merge tools. But even in this case, we believe that it can serve as a useful starting point.

In our opinion the main niche for operation-based merge tools are extendible systems to which new data-types can be added. Most object management systems fall into this category.

A prototype of the merge tool has been implemented as part of the **CAMERA** system. It is currently used to evaluate the ideas that were presented in this paper.

Acknowledgements

We would like to thank Gert Florijn and Doaitse Swierstra for their comments.

References

[AGMT86] Evan Adams, Wayne Gramlich, Steven S. Muchnick, and Soren Tirfing. SunPro: Engineering a practical program development environment. In Reidar Conradi, Tor M. Didriksen, and Dag H. Wanvik, editors, *Advanced Programming Environments*, pages 86–96. Springer Verlag, June 1986. Proceedings of an International Workshop. Trondheim, Norway.

[CFH88] William Courington, Jonathan Feiber, and Mashiro Honda. NSE highlights, NSE tackles large-scale programming issues. *SunTechnology*, pages 49–53, Winter 1988.

[CLR91] Thomas H. Cormen, Charles E. Leiserson, and Ronald L Rivest. *Introduction to Algorithms*. MIT Press, 1991.

[ECM90] ECMA. Portable common tool environment (pcte) abstract specification, December 1990. Standard ECMA-149.

[FLD+92] Gert Florijn, Ernst Lippe, Atze Dijkstra, Norbert van Oosterom, and Doaitse Swierstra. Camera: Cooperation in open distributed environments. In *EurOpen and USENIX Spring 1992 Workshop; Jersey, Channel Islands*, April 1992.

[HPR88] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. In *Proceedings of the SIGPLAN 88 Symposium on the Principles of Programming Languages*, 1988.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[LC84] D.B. Leblang and Robert P. Chase, Jr. Computer-aided software engineering in a distributed workstation environment. In P. Henderson, editor, *Sigplan/Sigsoft Software Engineering Symposium on Practical Software Development Environments*, pages 104–112. ACM, May 1984. In Sigplan Notices.

[Lip92] Ernst Lippe. *CAMERA: Supporting Distributed Cooperative Work*. PhD thesis, University of Utrecht, 1992.

[PAC89] PACT. *Configuration Management Guide*, December 1989.

[SK90] Christopher R. Sheedy and Stephanie L. Kinoshita. Version management tool. Tandem corporation; US patent nr 4,912,637, March 1990.

[Tic85] Walter F. Tichy. RCS — a system for version control. *Software Practice and Experience*, 15(7):637–654, July 1985.

[Wal88] Larry Wall. Patch – a program to apply diffs to original files, 1988.

[Wes91] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In Peter H. Feiler, editor, *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 68–79, 1991.