A State-of-the-Art Survey on Software Merging

Tom Mens

Abstract—Software merging is an essential aspect of the maintenance and evolution of large-scale software systems. This paper provides a comprehensive survey and analysis of available merge approaches. Over the years, a wide variety of different merge techniques has been proposed. While initial techniques were purely based on textual merging, more powerful approaches also take the syntax and semantics of the software into account. There is a tendency towards operation-based merging because of its increased expressiveness. Another tendency is to try to define merge techniques that are as general, accurate, scalable, and customizable as possible, so that they can be used in any phase in the software life-cycle and detect as many conflicts as possible. After comparing the possible merge techniques, we suggest a number of important open problems and future research directions.

Index Terms—Software merging, large-scale software development, merge conflicts, conflict detection, conflict resolution.

1 Introduction

As clearly pointed out by Perry et al. [49], support for software merging is a necessity during large-scale software development, where separate lines of development are carried out in parallel by different software developers and have to be merged at regular intervals. The need for software merging arises in the context of distributed applications, computer-supported collaborative work and concurrency control [14], [15], [43], and in the context of software configuration management in particular [12].

Software configuration management is the discipline of managing the evolution of large and complex software systems [61]. As a development-support discipline, it provides techniques and tools to assist developers in performing coordinated changes to software products. These techniques include *version control mechanisms* [12] to deal with the evolution of a software product into many parallel versions and variants that need to be kept consistent and from which new versions may be derived via *software merging*.

Note that the need for software merging depends on the chosen version control mechanism [56]. With *pessimistic version control*, all participants work on the same set of software artifacts and parallel editing of the same artifact is prevented by locking. Such a pessimistic locking protocol assumes a strict consistency model, in the sense that all participants have exactly the same views and data at all times. In practice, this approach is inadequate as soon as the number of software developers working in parallel exceeds a certain—typically very low—threshold. With *optimistic version control*, each developer can work on a personal copy of a software artifact. The advantage of this approach is that software developers are allowed to work completely separate for some time. The price for this is the need for merging. From time to time, the personal copies need to be integrated into a new shared version and conflicts between parallel changes need to be resolved during this merge process.

Nevertheless, software merging remains a time-consuming, complicated, and error-prone process because many interconnected elements are involved and merging depends on both the syntax and semantics of these elements. Unfortunately, most existing approaches to software merging lack either flexibility or expressive power.

Commercially available merge tools are usually based on textual merge techniques. This makes them very flexible since any program can be considered as a piece of text, irrespective of the programming language in which it was written. To some extent, textual merging can even be used to merge other kinds of software artifacts, such as code documentation. An inherent limitation of textual merging is that it can only detect very basic conflicts. It does not take the specific semantics of software artifacts into account because everything is treated as an ordinary piece of text.

To cope with this problem, a number of research prototypes have been developed that deal with syntactic and semantic merge conflicts. Unfortunately, detecting all possible semantic conflicts between two versions of an arbitrary software artifact is undecidable [5], [53]. To reduce this problem, one could narrow down the merge algorithm to a well-defined domain, e.g., a specific programming language, a particular phase in the software life-cycle, or a particular type of application (such as distributed applications).

On the other hand, sometimes a more widely applicable merge technique is required. This is, for example, the case if we want a merge tool that does not only deal with implementation artifacts but also with software artifacts in the higher life-cycle phases of software development. In such a case, a *domain-independent approach* is more suitable [14], [39], [43], [63]. Unfortunately, this typically results in a decreased accuracy. A domain-independent tool cannot rely on the detailed semantics of the software artifacts being considered, as these differ significantly in

- *The author is with the Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussels, Belgium. E-mail: tom.mens@vub.ac.be.*

Manuscript received 29 Feb. 2000; revised 14 Feb. 2001; accepted 26 Apr. 2001.
Recommended for acceptance by A.A. Andrews.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 111615.

0098-5589/02/$17.00 © 2002 IEEE

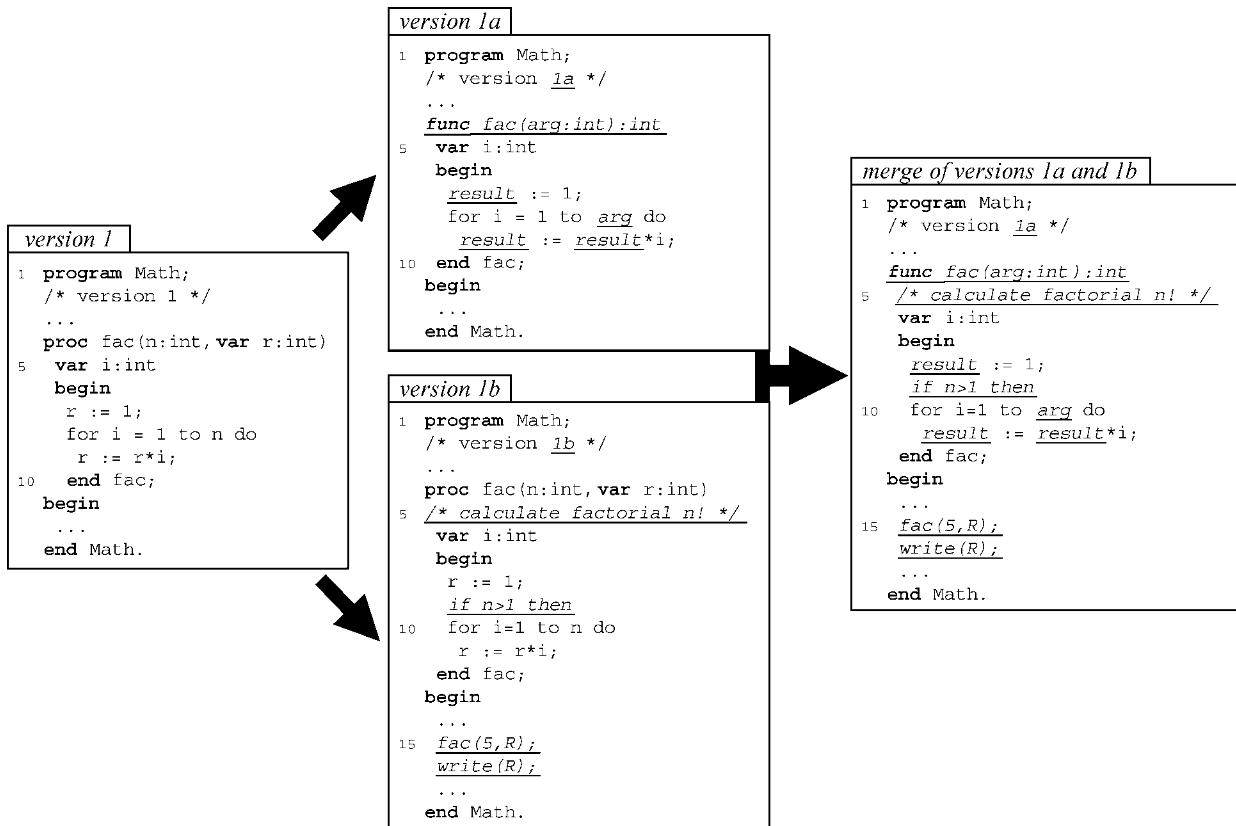

Fig. 1. A possible three-way line-based textual merge scenario.

different domains. Therefore, instead of detecting all possible conflicts, approximative techniques are required that try to detect as many conflicts as possible without compromising efficiency.

In order for a merge tool to detect syntactic and semantic merge conflicts rather than purely textual ones, a formal foundation is required. The complexity of the underlying formalism tends to increase with respect to the number of detectable syntactic and semantic conflicts. In order not to overly restrain the efficiency of a merge tool, a *lightweight* approach is often taken to detect as many conflicts as possible with a formalism that is as simple as possible.

The outline of this paper is as follows: Section 2 presents an overview of the different kinds of merge techniques and associated merge conflicts that can be distinguished. Section 3 elaborates on the mechanisms for detecting and resolving these conflicts. Section 4 relates merge techniques to delta algorithms that calculate the difference between two versions of a software artifact. Section 5 addresses some important criteria that need to be taken into account when designing and developing a merge tool. Finally, Section 6 concludes and proposes a number of future research directions.

2 Overview of Merge Techniques

This section provides a comprehensive overview of various merge techniques (used in research prototypes, as well as in commercial tools) that have been reported in literature and categorizes them according to a number of orthogonal dimensions. More precisely, the following alternatives are discussed: two-way or three-way merging; textual, syntactic, semantic, or operation-based merging; state-based or change-based merging; and reuse versus evolution.

2.1 Two-Way or Three-Way Merging

A first distinction can be made between two-way and three-way merge techniques. *Two-way merging* attempts to merge two versions of a software artifact without relying on the common ancestor from which both versions originated. With *three-way merging*, the information in the common ancestor is also used during the merge process. This makes three-way merging more powerful than its two-way variant, in the sense that more conflicts can be detected. Consequently, almost all currently available merge tools make use of three-way merging.

To illustrate the difference more clearly, let us take a closer look at the merging of text files. As a prerequisite to achieve two-way or three-way textual merging, we need to be able to identify the differences between text files. The Unix *diff* utility [27], [28] provides such support for two-way merging, while the Unix *diff3* utility does the same for three-way merging.

To show the difference between two-way merging and three-way merging more clearly, consider the example of Fig. 1, which represents version 1 of a program `Math` for calculating mathematical functions and two of its evolved versions (1a and 1b). The changes made to obtain each of these evolved versions are displayed in *underlined italics*.

With two-way merging, we only compare the differences between versions 1a and 1b. For example, line 4 in version 1b contains `proc fac(n:int,var r:int)`, while the corresponding line in version 1a contains `func fac(arg:int):int`. Similarly, version 1b contains a line `fac(5,R);` (line 15) that is not present in version 1a. Unfortunately, this information is insufficient to determine whether the differences are caused by a line addition, removal, or modification in only one of the evolved versions or by a simultaneous modification in both versions.

Three-way merging does not have this shortcoming. For example, line `proc fac(n:int,var r:int)` of version 1b is also present in ancestor version 1, implying that only version 1a made a modification to this particular line. Similarly, because the line `fac(5,R);` of version 1b did not occur in version 1, it must have been newly introduced by version 1a. This extra information is used by the merge algorithm to infer which lines of version 1a and version 1b should be included in the merged version. A possible outcome of the line-based merge is shown on the right in Fig. 1. The merge incorporates all changes/additions/deletions from both versions 1a and 1b and the changes of version 1a take precedence when there is a merge conflict. Note that this textual merge is not semantically correct, as will be explained in the next section.

2.2 Textual, Syntactic, Semantic, or Structural Merging

An important distinction between merge tools can be made based on how software artifacts are represented. Text-based merge tools treat software as a flat text file. A better alternative is to use a more structured form of software artifacts (such as a parse tree), or even to consider semantic information. More specifically, a distinction can be made between *textual*, *syntactic* and *semantic* merging.

2.2.1 Textual Merging

Text-based merge tools consider software artifacts merely as text files (or binary files). The most common approach is to use *line-based merging*, where lines of text are taken as indivisible units [27]. Examples of this are the *rcsmerge* tool in the Revision Control System (RCS) [60], Sun®'s *filemerge* tool [1], the DOMAIN Software Engineering Environment (DSEE) [33], [34], [37], the Concurrent Version System (CVS) [4], and merge tools that can be found in commercial configuration management tools.

With line-based merging of text files, common text lines can be detected in parallel modifications, as well as text lines that have been inserted, deleted, modified, or moved. Fig. 1 already showed an example of such a line-based merge. Because of its too coarse granularity, however, line-based merging has the disadvantage that it cannot handle two parallel modifications to the same line very well. Only one of the two modifications can be selected, but the two modifications cannot be combined.

In spite of its disadvantages, line-based merging remains a very useful technique because of its efficiency, scalability, and accuracy. According to some measurements performed with a three-way, line-based merge tool in an industrial case study [49], about 90 percent of the changed files could be merged without asking any questions. The more challenging task lies in providing automated ways to deal with those 10 percent of the situations that cannot be merged automatically. The reason why text-based merging fails in those cases has to do with the fact that text-based merge tools do not consider any syntactic or semantic information.

2.2.2 Syntactic Merging

Syntactic merging [10] is more powerful than textual merging because it also takes the syntax of software artifacts into account. A text-based merge technique often gives rise to unimportant conflicts such as a code comment that has been modified in parallel by different developers or conflicts that arise because of the introduction of some line breaks and tabs that make the code easier to read. A syntactic merger can ignore all these conflicts. It will only issue a merge conflict when the merged result is not syntactically correct. As a concrete example, consider the conditional program statement `if (n mod 2)=0 then m:=n/2`. Two parallel developers decide to change this program fragment in a different way, but with the same overall effect. The first developer extends the fragment to `if (n mod 2)=0 then m := n/2` *`else m:=(n-1)/2`*, while the second developer modifies it to *`m:=n div 2`* (where `div` is the whole division). Although both changes have exactly the same effect, a purely textual merge will probably combine these two evolutions into something like `m:=n div 2 else m:=(n-1)/2`, which is syntactically incorrect. A syntactic merger will detect this conflict, so that appropriate actions can be taken to resolve the problem. (In this case, it would suffice to remove the else part manually.)

General syntactic merge techniques can be categorized according to their underlying data structure: the ones that use (parse) *trees* as an underlying data structure and the ones that use *graphs* as data structure.

References [2], [63], and [67] are examples of tree-based merging that propose a domain-independent, three-way merge tool that can detect syntactic conflicts when merging parse trees or abstract syntax trees. *Cdiff* [20], another tree-based merge approach, is used to find differences between C++ programs by comparing their parse trees. The approach is restrictive because it relies on two-way merging and can only be used to merge C++ programs. A similar tool that allows merging C++ programs in a syntactic way is *TurboMixer*, which was developed in the course of the *BeyondSniff* project [9].

Rho and Wu [52] use attributed graphs as an underlying representation for software artifacts. The same is true for Mens [40], who additionally makes use of graph rewriting techniques in order to provide a formal foundation for software merging.

2.2.3 Semantic Merging

Syntactic merge tools are unable to detect some frequently occurring conflicts. For example, in the merge of versions 1a and 1b in Fig. 1, we encountered a problem on line 9 (`if n>1 then`), where a variable `n` was used that is not declared in the program. This is because version 1a and 1b each use another variable name (`arg` and `n`, respectively) for the same purpose. A syntactic merger is unable to detect this conflict since the program is still syntactically correct. Therefore, we

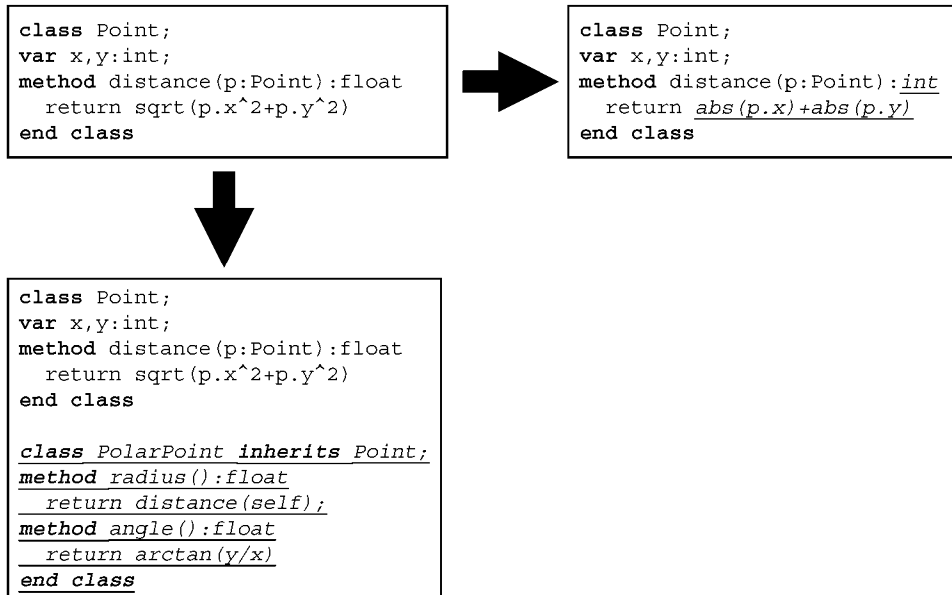

Fig. 2. Behavioral conflict due to late-binding.

will call it a *semantic conflict*. More specifically, it is a *static semantic conflict* since most compilers will detect the problem and issue an "undeclared variable" error. Observe that the merge of versions 1a and 1b contains a second semantic problem. Line 15 contains a procedure call `fac(5,R);` while the procedure definition has been replaced by a function definition `func fac(arg:int):int` on line 4. Obviously, these two changes at different locations in the code are incompatible and give rise to an error when the merged program is compiled.

Tree-based merge approaches based on parse trees cannot detect this kind of conflict because the relationship between procedure (or function) definition and procedure (or function) invocation is not made explicit in a parse tree or abstract syntax tree. *Graph-based* merge approaches (such as [40]) can detect this conflict because a graph representation maintains an explicit link between a definition and its invocations, thus making it easy to detect incompatibilities between them. This was also the idea of the *context-sensitive merge* approach of Westfechtel [63], where the abstract syntax tree is augmented with context-sensitive relationships that express the bindings of identities to their declarations.

Sometimes, even static semantic merging is insufficient because independent changes by parallel developers may still give rise to unexpected behavior in the merged result. Indeed, one has no guarantees about how the execution behavior of the merged program relates to the behavior of the programs being merged. The resulting *behavioral conflicts* can only be countered by resorting to even more sophisticated semantic merge techniques that rely on the runtime semantics of the code. Most approaches for detecting behavioral conflicts—like [7], [8], [24], [65]—rely on complex mathematical formalisms, such as denotational semantics, program dependence graphs, and program slicing. There are also more lightweight approaches available, such as *Semantic Diff* [31], which makes use of local dependence graphs.

An example of a merge that leads to a behavioral conflict due to the unexpected interaction of object-oriented inheritance with merging is shown in Fig. 2. A class *Point* contains two variables x and y, and a method *distance* to calculate the Euclidean distance of a given point to the origin. One software developer decides to add a new subclass *PolarPoint* that additionally implements two methods *radius* and *angle*. *radius* is implemented by performing a self-send to *distance*. A second software developer decides to change the implementation of *distance* by calculating the Manhattan distance instead. Because of this, however, the implementation of *radius* becomes invalid in the merge of these two evolutions. Indeed, *radius* should always be calculated using the Euclidean distance, which is not the case anymore.

2.2.4 Structural Merging

Restructuring and refactoring have been advocated as techniques to increase the maintainability of software systems and to reverse the effects of software aging and software erosion [18], [21], [45]. To this extent, refactoring and restructuring transformations are introduced that have the special property of being behavior preserving, i.e., they do not change the semantics of a software artifact, although its structure may change significantly. *Structural merge conflicts* arise when one of the changes to a software artifact is a restructuring and the merge algorithm cannot decide in which way the merged result should be structured. As an example of such a conflict, consider the situation of Fig. 3. The software artifacts used in this example represent an object-oriented class hierarchy (shown in UMLTM notation). Starting from an abstract class *Shape* with three subclasses *Circle, Rectangle*, and *Square*, one modifier decides to create a new subclass *Parallelogram* (by means of the operation *CreateSubclass*). Another modifier independently decides to

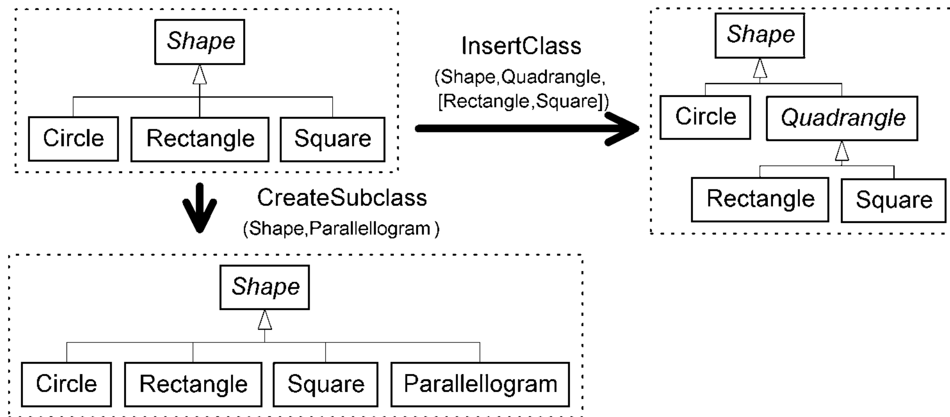

Fig. 3. Structural conflict.

refactor the subclasses *Rectangle* and *Square* by introducing a new intermediate superclass *Quadrangle* that captures the common behavior of both (using the operation *InsertClass*). When merging, the question arises whether *Parallelogram* should be made a subclass of *Quadrangle* or should it remain a direct subclass of *Shape*? In the case of a parallelogram, we should clearly choose the first alternative. In the case of a triangle, however, the second alternative should be selected. Hence, input from the user is necessary in order to resolve the conflict.

Unfortunately, most of the existing merge tools and techniques provide no support whatsoever for detecting structural merge conflicts. The main reason is that the information required to detect these conflicts is usually implicit and cannot be inferred from the source code only.

2.3 State-Based, Change-Based, or Operation-Based Merging

Yet another distinction can be made between state-based and change-based merging. With *state-based merging,* only the information in the original version and/or its revisions is considered during the merge. In contrast, *change-based merging* additionally uses information about the precise changes that were performed during evolution of the software. Obviously, two-way merge techniques are always state-based since they can only compare two revisions without taking into account how these revisions have been obtained.

Operation-based merging is a particular flavor of change-based merging that models changes as explicit operations (or transformations) [3], [17], [36], [40]. These evolution operations can be arbitrarily complex and typically correspond to the commands issued in the software development environment. For this reason, the sequences of change operations are sometimes referred to as *command histories* [3].

Operation-based merge approaches can improve conflict detection and allow for better support when solving these conflicts [17], [35], [36], [39]. An example of this enhanced expressiveness is shown in the example of Fig. 1. We already mentioned the presence of a semantic conflict because of the fact that version 1a introduces a call to the `fac` procedure somewhere in the main body of the Math program, while version 1b changed the procedure `fac` into a function. With an operation-based versioning tool, both changes can be expressed in terms of evolution operations `AddInvocation(Math,fac)` and `ProcToFunc(fac)`, respectively. To detect the conflict, we simply need to compare both operations to see that they affect the same entity `fac` in incompatible ways.

In other words, in order to detect merge conflicts, we do not need to compare the parallel revisions entirely since it suffices to compare only the evolution operations that have been applied to obtain each of the revisions. For certain combinations of operations, conflicts will be reported. Although the used operations, reported conflicts, and proposed resolution strategies may vary from one approach to another, the underlying idea seems to be generally agreed upon [14], [17], [39], [43], [57].

Operation-based merging is general in the sense that it can be used for detecting syntactic, structural, and even some kinds of semantic conflicts. For example, Edwards [14] takes an operation-based approach to detect and resolve semantic conflicts that arise purely from application-supplied semantics. Although only the applications themselves “know” how their behavior can create the presence of conflicts, this knowledge is captured by specifying the particular operations that can be performed in each application and how these operations can give rise to conflicts.

An operation-based approach makes it easier to implement a multiple *undo/redo mechanism*. For *undo*, it suffices to perform the last applied operations in the opposite direction and, for *redo*, we simply reapply the operations. In the collaborative application framework GINA [3], merging of command histories is achieved by selectively using a *redo* mechanism to apply one developer’s changes to the other’s version. This approach is not very efficient in the case of long command histories or when the granularity of the history is too fine.

2.4 Reuse vs. Evolution

Merge conflicts do not only arise when changes are made to the same software by *different* software developers. They also occur if there is only one developer that makes changes to a piece of software that is being reused by other parts of

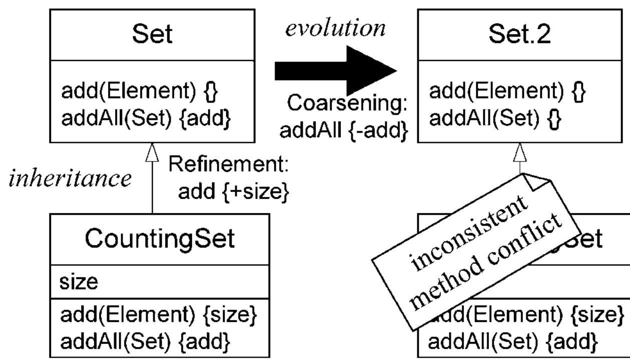

Fig. 4. Base class exchange.

the software. This is, for example, the case in object-oriented software development, where a class can be "reused" by employing inheritance or some other incremental modification mechanism. In this particular context, a merge conflict corresponds to the so-called *fragile base class* problem, which indicates undesired effects in independently developed subclasses of a given base class when this base class evolves in unexpected ways.

Just like we can have syntactic and semantic merge conflicts, there is also a syntactic and semantic variant of the fragile base class problem. The syntactic variant is dealt with in the SOM approach [30], allowing a base class interface to be modified without needing to recompile clients and derived classes dependent on that class. In the *reuse contract* approach [57], the semantic variant is addressed as well. Mezini [41] proposes a prototype tool based on this technique.

As an illustration of the reuse contract approach, let us take the example of Fig. 4, which is adopted from Steyaert et al. [57]. A class *Set* has two methods *add* and *addAll* for adding a single element or a set of elements, respectively. *addAll* is implemented by calling *add* repeatedly (indicated between curly braces). *CountingSet* is a subclass of *Set* that specializes its behavior by keeping an extra attribute *size* that is augmented each time new elements are added to the set. For efficiency reasons, the method call from *addAll* to *add* is inlined in a new version of *Set* (base class exchange). As a result, *CountingSet* will not have the intended behavior anymore. It relied on the assumption that *addAll* always invokes *add*, which is not the case in the evolved version of *Set*.

Steyaert et al. [57] detect this semantic conflict using an operation-based merge approach: The changes are explicitly documented by means of predefined operations (in this case, *Refinement* and *Coarsening*) and these operations are compared in a conflict table. In this case, they yield an *inconsistent method* conflict.

3 Merge Conflicts

Section 2 distinguished three kinds of inconsistencies that can arise during a merge: syntactic, structural, and semantic conflicts. Moreover, the semantic conflicts can be subdivided further depending on whether they can be detected at compile-time or at runtime. This section discusses several issues related to these conflicts in some more detail. First, we examine the various approaches that can be used for *detecting* merge conflicts. Next, we discuss some techniques for *resolving* merge conflicts. Finally, we address some heuristics for trying to reduce the—sometimes very high—number of detected conflicts.

3.1 Conflict Detection

In this section, we discuss some interesting conflict detection techniques and see how and which kinds of conflicts they can detect.

3.1.1 Merge Matrices

Many merge tools take an ad-hoc approach for detecting merge conflicts. With an operation-based merge approach, conflicts can be detected in a more disciplined fashion by comparing the modification operations that have been applied in parallel by different developers. All pairs of operations that lead to an inconsistency are summarized in a so-called *conflict table* or *merge matrix*. This makes it possible to detect merge conflicts by performing a simple table lookup. Steyaert et al. [57] followed this approach to deal with the fragile base class problem.

A slightly more general approach is taken by Feather [17]. Instead of storing the operations directly in the merge matrix, they are analyzed to determine what changes to specification properties they induce and this information is stored instead. Conflicts can then be detected by comparing all possible changes to specification properties. This makes it easier to introduce new modification operations. One only needs to determine how the newly introduced operations can be decomposed in terms of changes to specification properties.

Munson and Dewan [43] propose a merging framework that is entirely based on the idea of merge matrices. In this framework, the matrices do not only specify the conflicts that can occur, but also the action that should be performed to resolve the conflicts. Merge matrices are consistently used at each level of granularity (i.e., they deal with operations on atomic as well as on composite software artifacts) and for each kind of software artifact being considered. Many different merge matrices may be used, depending on factors such as the kind of user doing the merge, the time when the operations were performed, or any other factor that is deemed relevant to steer the merge process.

Another way to generalize the merge matrix approach is explored in the domain-independent formalism of Mens [39], where software artifacts are represented as graphs, evolution is represented by graph rewriting, and type graphs are used to specify domain-specific constraints that have to be satisfied by all graphs in a particular domain. By relying on formal properties of graph rewriting (more precisely, the notion of parallel dependence), one can formally define which pairs of modification operations (formally represented as graph productions) yield a syntactic merge conflict. This leads to a formal and domain-independent characterization of the intuitive notion of merge matrix.

The main problem with merge matrices arises when we want to merge more than two versions. For example, if there

are M kinds of operations and there are N versions that need to be merged, we would get an N-dimensional merge matrix that requires polynomial space complexity of $O(M^N)$.

3.1.2 Conflict Sets

As an alternative to merge matrices, *conflict sets* are used by Edwards [14] in the context of collaborative applications to group together potentially conflicting combinations of operations based on the application-supplied semantics. Depending on the kind of application (e.g., a multiuser software development environment, a collaborative drawing editor, a distributed office furniture layout program), the kinds of operations and associated merge conflicts can differ dramatically. Conflict sets correspond to the types of conflicts that may exist in an application. As such, they are statically defined, in the sense that they remain fixed as long as the application semantics does not change. For example, in a distributed layout application one could distinguish between *temporal* and *spatial* conflicts. Operations that belong to the same conflict set may potentially cause conflicts when merged together. Any given operation may simultaneously participate in multiple conflict sets. Conflict sets address scalability since they restrict the number of operations that we must consider when searching for conflicts.

3.1.3 Semantic Conflict Detection Techniques

Horwitz et al. [24] were the first to develop a powerful algorithm for merging (or "integrating") program versions without semantic conflicts, based on the semantics of a very simple assignment-based programming language. The merge algorithm uses the underlying representation of *program dependence graphs* [26] and uses the notion of *program slicing* [25] to find potential merge conflicts. Despite its power, the algorithm poses a number of limitations. For example, if one version of a program changes the way a computation is performed without changing the external behavior (i.e., input/output), while another variant adds code that merely uses the results of the computation, a conflict will be issued. Yang et al. [65] overcome this limitation by introducing semantic-preserving transformations. Binkley et al. [8] propose another extension to handle programs that contain procedures (which may be mutually recursive). While all these approaches allow us to detect behavioral conflicts, they have the important disadvantage that they remain restricted to a particular implementation language.

Berzins [7] takes a more general approach, by providing a language-independent definition of semantic merging. To this extent, a generalization of traditional denotational semantics [58] is used in order to provide the additional structure necessary to formally define semantic merge conflicts. More specifically, Brouwerian algebras—which are a generalization of Boolean algebras—are used [38]. A realistic example of a semantic domain that needs to be modelled with Brouwerian algebras is PSDL, a language with hard real-time constraints. Dampier et al. [13] describe a method for merging changes in this particular language.

The main shortcoming of Berzins' approach is that it works on the underlying semantic interpretation of a model directly, so that it cannot be used to diagnose and locate conflicts between changes in the concrete syntactic representation of a program. This makes the approach impractical because it cannot be used to pinpoint the actual source of a semantic conflict in the software. Berzins [6] addresses this problem to some extent by restricting the general formalism to only some special cases.

Another technique, which appears to be more practical in the sense that it can be applied to real programming languages such as C, is used by *Semantic Diff* [31]. This two-way merge tool identifies differences between two versions of a procedure by using local dependence graphs to compare their observable input-output behavior. This approach is fully automatic, fast, and easy to use, but is limited in the sense that it can only find *local* semantic differences because it only compares procedures with each other. Hence, it cannot detect more global interprocedural semantic conflicts. While the approach has been illustrated for C programs, it seems to be generalizable to other programming languages as well. However, the presence of late binding and polymorphism in object-oriented programs makes it far from trivial to apply the approach in an object-oriented setting.

Mens [39] presented an alternative lightweight approach to detect semantic merge conflicts. To illustrate it, reconsider Fig. 4 of Section 2.4. The *inconsistent method* conflict occurred because the implicit assumption that *addAll* always invokes *add* was broken. This conflict can be detected by looking at the merged result only, if we document which changes have been introduced by which developers. In this particular case, a call from method *add* to attribute *size* was *added* by *developer 1*, while a call from method *addAll* to method *add* was *removed* by *developer 2*. If we use graphs to represent software artifacts, this conflict situation can be detected as an instance of a more general graph pattern (where we ignore the names of the entities that are removed). If an instance of this graph pattern can be found in the result of the merge, an *inconsistent method* conflict is reported. In a similar way, other semantic conflicts, such as *unanticipated recursion* can be detected by looking for instances of other predefined graph patterns. One disadvantage of this approach is that complex behavioral conflicts cannot always be detected. Another disadvantage is that looking for a subgraph in a graph can be a very expensive operation, so that the approach cannot be scaled up to large software systems.

3.2 Conflict Resolution

Once merge conflicts have been detected, techniques are needed to resolve these conflicts. These can range from a manual—and often time-consuming—process, over an interactive process where the resolution algorithm requires interaction with the software developer, to a fully automated conflict resolution tool. A lot depends on the kinds of conflicts the tool intends to solve and the level of accuracy that needs to be reached.

Depending on the kind of merge conflict, different resolution strategies may be necessary. One particular kind of conflict occurs when two parallel changes can only be merged if they are applied in a certain order because the inverse order gives rise to an inconsistency. This is typically the case when a renaming is involved. Since renaming is a

global operation that can affect many different places in the code, it should always be applied *after* the other parallel modifications to ensure that all references to the artifact being renamed are renamed as well. This is only one particular example of a situation where we need to impose an ordering on the parallel modifications that are involved.

Another automatic resolution strategy needs to be followed when the same modification is applied in parallel lines of development. For example, the same procedure or function can be introduced or removed twice. This conflict can be resolved easily by ignoring one of both identical modifications.

As an example of a conflict that cannot be resolved in an automated way, consider the situation where a function is renamed differently in parallel modifications. If this is the case, the merge algorithm cannot automatically decide which of both renamings is most appropriate. To deal with such situations, a tool could provide automated assistance for negotiating the resolution of conflicts in the style of the *Programmer's Apprentice* [54]. Negotiation of changes is also an important aspect of *Infuse* [47], the change management component of the *Inscape* environment [48], which is an integrated environment for developing large-scale software systems with large groups of developers.

In the context of operation-based merging, Edwards [14] proposes a number of interesting resolution strategies. The *explosion strategy* calculates all possible paths that may lead to a valid solution. Afterwards, the user can choose the most appropriate solution. The main disadvantage of this strategy is that it can lead to a combinatorial explosion of potential solutions. The *promotion strategy* tries to avoid conflicts by reducing the dependencies in a sequence of modification operations. Some operations that depend on earlier operations can be promoted into new operations that do not cause a dependency anymore. The *recursive acceptance strategy* represents a "what if" scenario in which the user can iteratively resolve cascading conflicts. If an operation conflicts with earlier operations, one can either remove the current operation or remove one of the earlier conflicting operations. This process is applied recursively until all conflicts have been resolved.

Munson and Dewan [43] also propose a number of different merge strategies (which they refer to as *merge policies*) for three-way merging. All these resolution strategies are implemented in a uniform and customizable way using merge matrices that can be fine-tuned by the user. We will only discuss two of these strategies here. *Consolidation* is used when two revisions of a common base version need to be merged and it is expected or known that most of the parallel changes are complementary (e.g., when the changes are made to different program modules). In case of deletions, the merge proceeds automatically. In case of overlapping changes, one of both changes is chosen interactively. *Reconciliation* is used when the revisions to be merged are likely to introduce merge conflicts. In that case, simply selecting the appropriate revision in case of an overlap is not sufficient since one may interactively want to combine the changes that lead to the conflict. Reconciliation is typically needed if different software developers make independent changes to the same part of the code. If they make changes to different weakly coupled parts of the code, however, consolidation is the more appropriate strategy. In addition, in the case where one developer restructures the code (in a behaviorally-preserving way), while the other developer makes changes to part of the code, consolidation is probably more appropriate.

Note that it is not always necessary to resolve conflicts after they have been detected. In some cases, one might decide to tolerate conflicts temporarily [14]. *Conflict tolerance* means that the merge technique allows conflicts that have not been resolved in the merged result. The *Infuse* change management tool [47] provides such explicit support for handling temporary inconsistencies during changes.

3.3 Reducing Conflicts

One of the most important problems with software evolution is that small changes often have a large impact. This is referred to as *change propagation* or the *ripple effect* [68]. In practice, it implies that merge tools often give rise to an abundance of conflicts being reported. There are a number of ways to try to reduce the number of reported conflicts to a more manageable number.

The most obvious way to localize the effect of changes and, consequently, to reduce the likelihood of merge conflicts is by resorting to *information hiding techniques*. This is not always sufficient since changes often have effects beyond the imposed encapsulation boundaries. Nevertheless, approaches like *Semantic Diff* [31], that only detect local changes within each procedure of a program, but ignore interprocedural effects, seem to perform fairly well in practice.

A related approach that allows us to detect local inconsistencies first and inconsistencies that are more global later is based on *graph partitioning*. If we use graphs to represent software artifacts and their dependencies, we can partition all the nodes in the graph according to their number of dependencies. This yields a number of subgraphs for which the conflicts will be detected first. As soon as all conflicts in each subgraph have been resolved (local consistency), we start looking for conflicts between different subgraphs (to obtain global consistency). Both [36] and [47] use this idea. The main difference is that [47] supports a *hierarchy* of graph partitionings, which makes it possible to limit the number of potential conflicts at each phase and to resolve all conflicts in an incremental fashion. Because graph partitioning is intractable [19], efficient heuristics are needed to obtain partitions that approximate the ideal case. Another problem is that each modification that is being made can alter the dependencies, so that a new partitioning might be needed after each iteration.

Another way to reduce the number of reported conflicts is by *ignoring temporary inconsistencies*. This typically occurs when small changes are being made that are part of a larger coordinated change.

Asklund [2] proposes to minimize the number of reported merge conflicts by using *fine-grained revision control*, where the software changes should be as small as possible. By evolving the software with small increments, the number of merge conflicts will remain small in each step. This idea is supported by Perry et al. [49], who have noticed on a statistical basis in a large-scale industrial case

that making large changes tends to lead to parallel versions that cannot be merged without some very costly overhead and coordinated effort.

Still another way to reduce conflicts is by keeping parallel developers aware of each other's changes, so that they can proactively use this information to anticipate overlapping changes and take appropriate actions to avoid future merge conflicts. Tool support is very important here, to allow developers to find and resolve potential conflicts even before the merge actually takes place.

Finally, in situations where we have one main development line and several branches constituting parallel development, Bruegge and Dutoit [11] give two additional heuristics for minimizing merge conflicts. First, changes to the main development line should be restricted to bug fixes, and important new changes should be made in separate development branches. Second, the number of branches should be as small as possible. Only use branches when parallel development is required. Creating a branch is a significant event that should be carefully planned and approved by management.

4 Delta Algorithms

Delta algorithms or *difference algorithms* are used to calculate the difference (or delta) between a version and one of its revisions. Version control tools—that typically make use of text-based merging—rely on delta algorithms to reduce storage space [29]. Delta algorithms also speed up programs because there is less I/O required to read a delta from disk and the amount of time saved is higher than the decompression time required to regenerate the entire revision from the delta.

In this section, we will take a closer look at how the different kinds of delta algorithms are related to the various merge techniques described in the previous sections. The discussion will be structured according to the types of deltas that can be distinguished: symmetric versus directed deltas; textual, syntactic, or semantic deltas; forward or backward deltas; and state-based or change-based deltas.

4.1 Symmetric vs. Directed Deltas

A *symmetric delta* calculates the difference between two versions v_1 and v_2 as the set difference $v_1 \backslash v_2$ or $v_2 \backslash v_1$. A *directed delta* specifies the difference between two versions as a sequence of modification operations. Symmetric deltas are typically used in the context of two-way merging, whereas directed deltas are most useful in the context of three-way operation-based merging.

However, using directed deltas could give rise to unexpected redundancy [36]. Given an arbitrary sequence of operations, it is often the case that some operations can safely be omitted because they are redundant with earlier or later ones in the sequence. For example, an operation can make a change to an entity that is deleted afterwards or an update operation can be overwritten by later updates. Removing all this redundant information is necessary to reduce storage space, speed up conflict detection, and remove some unnecessary intermediate conflicts. Typically, redundancy removal can be performed during a preprocessing phase before the actual conflict detection algorithm is applied. Mens [39] describes such a redundancy removal algorithm and even combines it with a normalization algorithm. Given a predefined set of modification operations, each operation sequence can be transformed into a unique canonical form (provided that we impose a certain order on the operations). However, it is still unclear how this idea can be scaled up in the presence of more complex operations.

4.2 Textual, Syntactic, or Semantic Deltas

The difference between textual, syntactic, and semantic merging is also directly reflected in the differencing algorithm that is used.

If we want to perform *textual merging*, we need to compare two text files, which can be achieved by finding the longest common substring and then computing a distance delta from this common substring [28]. An example of a textual delta algorithm (used in RCS and some commercial products) is the Unix *diff* utility [27]. A finer-grained approach is taken by the algorithms *bdiff* [59] and *vdelta*, which have empirically shown to be more efficient than *diff* in time as well as compression ratio [29].

If we want to perform *syntactic merging*, we first need to compare the delta between two syntax representations (e.g., parse trees). Yang [66] describes a comparison tool for detecting syntactic differences between programs. An example of a syntactic delta algorithm specifically destined to find the difference between UML™ diagrams is the Rational Rose™ Visual Differencing tool.

If we prefer *semantic merging*, we need to calculate semantic differences between two versions of a program. This is achieved by *Semantic Diff* [31], which expresses its results in terms of the observable input-output behavior.

4.3 Forward or Backward Deltas

Delta algorithms can also be distinguished based on how subsequent versions of a software artifact are stored. With *backward deltas*, the latest revision is stored entirely and previous versions are stored as deltas. With *forward deltas*, it is the original version that is stored entirely, while newer versions are expressed as a delta of the original one.

Forward deltas may be more intuitive, but backward deltas are often more useful because they speed up retrieval of the last and probably most frequently accessed revision. SCCS [55] is an example of the use of forward deltas, while RCS [60] and most other approaches make use of backward deltas.

4.4 State-Based or Change-Based Deltas

In Section 2.3, we already discussed the distinction between state-based and change-based merging. The way deltas are stored is tightly related to whether a state-based or change-based approach is adopted. Approaches like the Source Code Control System (SCCS) [55] and RCS [60] are state-based, while approaches like EPOS [22], [35] are change-based.

Delta algorithms in *state-based* version control systems calculate the difference between a revision and its ancestor version and store only this difference instead of the entire revision. To reconstruct the revision from the delta, the inverse process is applied.

Within the *change-based* approach, a distinction can be made between *intensional* and *extensional* versioning (following the terminology of Conradi and Westfechtel [12]). With the *extensional* variant (also referred to as *embedded deltas*), changes are annotated to each version in order to specify its difference relative to another version. This is the approach taken by Asklund [2], who uses embedded deltas to specify the differences between two versions of a parse tree and by Rho and Wu [52], who use embedded deltas for specifying differences between graphs.

With *intensional* change-based versioning (also referred to as the *change set model*, used in systems like EPOS [22]), changes can be specified independently from the versions to which they are applied. This makes them more flexible since the same changes can be applied more than once, for example, to parallel versions of the software under development. This is very useful for making bug fixes and improvements to a piece of software after it has been released.

Approaches for *operation-based* merging rely on an intensional change-based model since the delta records the operations that have been used to obtain the revision from the base version, rather than the actual differences between both. In order to reconstruct a revision, the operations simply have to be *replayed* using the original version as a starting point.

5 Design Criteria

Many criteria influence the design of a merge tool. This section discusses the impact of each of the following design criteria: degree of formality, accuracy, domain independence, granularity of the considered software artifacts, scalability and efficiency of the chosen merge approach, and degree of customizability of the associated merge tool.

5.1 Degree of Formality

A first distinction is based on the degree of formality of a merge tool. One can opt for an *ad hoc* informal approach, a lightweight formalism, or a completely formal approach. More sophisticated formalisms typically allow us to detect more semantic merge conflicts. On the other hand, the degree of formality tends to be inversely proportional to the efficiency or practical use of the merge tool. Therefore, a balance should be made between these two conflicting goals.

Approaches such as line-based textual merging are typically *ad hoc* and can only be used to detect straightforward merge conflicts.

Most operation-based merge techniques [17], [36], [55] fall in the category of *lightweight formal approaches*. Even *Semantic Diff* [31], which allows us to detect semantic conflicts, can be considered a lightweight approach. Semantic differences between two versions of a procedure are detected by comparing local dependence graphs that express the dependency between the arguments and variables that are used inside the procedure. The local nature of this approach makes it impossible to detect interprocedural semantic inconsistencies. This was a deliberate choice of the designers of this merge technique, in order not to sacrifice the efficiency of the approach.

Binkley et al. [8] propose a *full-fledged formalism* that heavily relies on program dependence graphs [26], [46] and interprocedural program slicing [25], [62] in order to detect semantic inconsistencies. Horwitz [23] proposes an alternative to program slicing, by using a graph partitioning algorithm. If programs are represented as graphs, the partitioning ensures that programs belonging to the same partition have equivalent behaviors [64]. Another example of a completely formal approach is given by Berzins [7], who relies on Brouwerian algebras, an extension of the mathematical formalism of denotational semantics [58].

5.2 Accuracy

It is a utopia to try and create merge tools that can detect any kind of conflict when merging arbitrary software artifacts because any nontrivial property of a program's execution behavior—in this case the guarantee that there are no undesirable semantic interactions—is undecidable [5], [53]. For this reason, existing merge techniques always make a restriction on the kind of conflicts they can detect. In some situations, certain conflicts will remain undetected (false negatives) or conflict situations will be reported while in fact no problem exists (false positives).

In the case of line-based textual merging, measurements performed in a large-scale industrial case study showed that about 90 percent of the changed files could be merged automatically [49]. Only 1 percent of these merges were false positives, in the sense that the merge tool made an inappropriate decision. Moreover, most of these false positives could be detected easily since they resulted in compiler syntax errors. Nevertheless, a textual merge approach remains *unsafe* since there are no guarantees about how the execution behavior of a merged program produced by a textual merge tool relates to the execution behavior of the programs that have been merged [8].

So-called *conservative* approaches provide *safe* approximations, in the sense that they prohibit false negatives and, hence, guarantee that the merge is behaviorally correct. An example of such an approach is the semantic merge technique of [8], [24]. This approach avoids false negatives by restricting the software artifacts to a very simple programming language. False positives, on the other hand, cannot be avoided in general.

One of the disadvantages of conservative approaches is that they sometimes prohibit merges because the correctness of the merge cannot be guaranteed, even if the software developer knows that the merge will work correctly. For example, if one modification changes the way a computation is performed (without changing the values computed), while another modification adds code that uses the result of the computation, the merge algorithm of Horwitz et al. [24] will issue a conflict. This problem is addressed by Yang et al. [65] by relying on semantics-preserving transformations.

5.3 Domain Independence and Customizability

Another distinguishing feature of merge techniques is how domain independent they are. An ideal merge tool should be general enough to merge all kinds of software artifacts, independent of the chosen programming language, the adopted paradigm, the problem domain, the

considered phase in the development process, the kind of application, or the available technology. This generality is essential if we want to provide uniform support for evolution of software artifacts throughout the entire software development life-cycle.

To a certain extent, text-based merging can be considered as a domain-independent technique since any kind of software artifact can be treated as a pure text file or binary file.

Although domain independence appears to be in direct contradiction with semantic and syntactic merging, a few merge approaches have been proposed with the specific aim of being domain-independent [14], [39], [43], [63]. Their underlying idea is that they provide a domain-independent merge framework that can be customized with domain-specific and user-provided information. In order to obtain a language-independent merge tool, Yang [67] suggests to carefully separate language-dependent modules from language-independent ones. Additionally, Munson and Dewan [43] suggest making a clear separation between user-independent and user-specified information and between user interface issues and algorithmic details. Finally, there should be a clear separation between conflict detection and conflict resolution, so that each of them can be fine-tuned without influencing the other.

Operation-based merging seems to be particularly suitable to achieve domain independence. Mens [40] proposes to use *graph rewriting* as an underlying formalism for domain-independent operation-based merging. Munson and Dewan [43] propose a flexible domain-independent merging framework-based on the underlying technique of merge matrices-that can be used for textual, syntactic, as well as semantic merges. Edwards [14] proposes a domain-independent (object-oriented) framework based on *conflict sets*. In order to customize the framework to a particular application domain, typically only a small amount of code must be written. The approach features a clear separation between conflict detection and conflict resolution. Domain-specific conflict detection mechanisms must be provided to detect domain-specific conflicts. Conflict resolution is based on "pluggable" resolution policies that are capable of managing any set of detected conflicts, irrespective of their domain-specific semantics. If desired, however, policies with more intimate domain-knowledge may be provided as well. The policies can also cope with conflict tolerance and user interface issues.

5.4 Granularity

Yet another way to distinguish merge techniques is according to their level of granularity, i.e., the amount of detail to which merge conflicts can be detected.

Let us first look at text-based merging. The most fine-grained approach would be to use single *characters* (or bytes) as indivisible units, but this leads to algorithms that are too inefficient to be useful in practice. A more coarse-grained approach can be taken by using substrings or blocks (of characters or bytes). This is the approach taken by algorithms such as *bdiff* [59] and *vdelta*. An even more coarse-grained approach such as *diff* [60] considers lines instead of characters as indivisible units [27]. In the context of collaborative writing, Neuwirth et al. [44] propose a textual merge tool where the granularity can be controlled in a flexible way (either word, phrase, sentence, or paragraph).

Syntactic merge techniques can also have different levels of granularity depending on the amount of syntactic information that is taken into consideration. Yang [67] proposes to express syntactic differences up to the level of individual tokens in the parse tree, thus allowing fine-grained conflict detection between single program statements. For efficiency reasons, however, it is possible that the generated parse trees only represent code up to a certain level of detail, e.g., the parsing may stop at the level of procedures, thus not considering intraprocedural information. With this approach, the merge algorithm cannot detect conflicts between parallel changes that are made inside the same procedure. The opposite approach is also possible. Jackson and Ladd [31] only consider semantic information at a very fine level of granularity (intraprocedural merging), but ignore information at a more coarse-grained level (interprocedural merging) for efficiency reasons. Because of this, they only detect semantic conflicts that occur when making parallel changes inside the same procedure.

We conclude that, although an ideal merge tool should be able to merge software at any level of granularity, for efficiency reasons, we sometimes need to restrict the amount of information that is taken into consideration.

5.5 Scalability and Efficiency

Another important feature of a merge technique is its *scalability*. This has to do with how well the employed algorithms scale up in terms of efficiency (i.e., performance and memory usage) as the size of the software system increases. If the complexity of the algorithm grows polynomially—or even worse, exponentially—the merge technique is not scalable. This is the case with syntactic merge approaches that resort to global impact analysis techniques such as program dependence graphs [8], [24]. To address this problem, Jackson and Ladd [31] merge at the level of procedures only (intraprocedural merging). Because this approach only detects *local* semantic effects of a change, it gives rise to another scalability problem. Due to the propagation of changes, local semantic changes might affect the behavior in many other parts of the software, some quite remote from the site of the change. Despite this shortcoming, preliminary experiments with this approach in a large real-time software system involving several hundred developers seem promising.

Another aspect of scalability is the ability of the merge technique to cope with arbitrarily complex modifications. This heavily depends on the chosen approach. Line-based textual merge tools only detect conflicts by comparing lines of text in parallel revisions of the same version. As such, more global conflicts that have to do with modifications that change many lines simultaneously (such as changing the layout of a certain piece of code or performing a global renaming of some variable) will give a conflict for each line involved in the modification. For these situations, preferably, a more global conflict should be reported. This is only possible if we take an operation-based approach, where the actual modification operations are taken into account as well.

Operation-based merge approaches seem to be very promising from the viewpoint of scalability. Lippe and van Oosterom [36] address scalability issues on a more or less formal basis, by relying on a notion of local and global commutativity between primitive operations. This property is used to detect conflicts between two sequences of primitive operations. First, potentially conflicting primitive operations are partitioned into blocks using global commutation properties. Next, for each block, local commutation properties are used to identity the precise conflicts. This technique can be used to limit the number of reported merge conflicts as the system grows in size. Other techniques to address this problem have already been discussed in Section 3.3.

Mens [39] addresses the scalability of operation-based merging by combining primitive operations into composite ones and by defining merge conflicts directly in terms of these composite operations. Moreover, certain conflicts between the primitive constituents of a composite operation can be ignored by relying on the precise semantics of the composite operation.

The conflict set approach of Edwards [14] addresses scalability by partitioning the search space while looking for conflicts. Only operations that belong to the same conflict set can cause a semantic conflict.

In spite of all these research efforts, empirical results about the practical applicability and scalability of operation-based merging in large-scale collaborative software development are still missing.

5.6 Degree of Automation

Merge tools can range from a manual—and often time-consuming—process, over a semiautomated process that requires interaction with the user, to a fully automated approach. While differencing and conflict detection is usually fully automatic, conflict resolution is typically interactive. Only in very specific situations is it possible to fully automate the merge process. In the flexible merge framework of Munson and Dewan [43], one can make a fine-grained combination of interactive and automatic approaches.

In order to resolve merge conflicts in an automated way, one can resort to *automatic* or *default conflict resolution strategies* [2], [42], [43]. For example, if all changes are tagged with a timestamp, we can decide to keep the entity with the most recent timestamp in case of a conflict. Another strategy would be to keep a list of user priorities and the change by the user with the highest priority is included in the result.

Although using default resolution strategies is useful, it is not foolproof. Occasionally, the proposed solution needs to be revised manually after the merge has been performed because the merge tool took the wrong decision. Additionally, some conflicts are too complex to be resolved in an automatic way. For these conflicts, user interaction will always be required.

6 Conclusion and Research Directions

Software merging is essential to deal with parallel software modifications carried out by different software developers that are not necessarily aware of each other's changes. This paper presented an overview of the state-of-the-art merge techniques and compared them based on a number of important criteria, such as expressiveness, formality, granularity, domain independence, customizability, scalability, efficiency, and accuracy.

Three-way operation-based merging is a particularly powerful approach that can detect syntactic and semantic merge conflicts rather than only textual conflicts. It is also able to detect structural conflicts caused by restructuring operations. Operation-based merge techniques can also be scaled up to deal with more complex modification operations.

Nevertheless, when looking at commercially available tools, most of them stick to the primitive approach of textual merging. This allows them to deal with a number of simple merge conflicts, but renders them inadequate in many situations. An interesting avenue of research would be to find out how to combine the virtues of different merge techniques. For example, one could combine textual merging with more formal syntactic and semantic approaches in order to detect and resolve merge conflicts up to the level of detail required for each particular situation.

Graphs look promising as an underlying representation for evolvable software and can provide a domain-independent approach to software merging. Many semantic merge techniques rely on graph theory: graph rewriting [39], [40], local dependence graphs [31], program dependence graphs [8], [24], graph partitioning [23], [36], [47], graph-based deltas [52], etc. It remains to be investigated, however, whether graph-based merge algorithms scale up to large software development efforts with dozens of parallel versions and with hundreds of changes per day.

In general, we need more empirical and experimental research on the validation and scalability of syntactic and semantic merge approaches, not only regarding conflict *detection*, but also regarding the amount of time and effort required to *resolve* the conflicts. We also need ways to manage the number of reported merge conflicts as the software system grows in size. A final aspect of scalability is how well existing merge techniques can cope with multiple developers. According to Perry et al. [49], in many situations, more than two (sometimes even up to 16) software developers make parallel changes to the same software artifact. This means that we need techniques for merging more than two parallel versions simultaneously. Unfortunately, most existing merge techniques only allow us to merge two parallel changes at a time.

While most approaches to software merging have been validated on imperative programming languages, it is not trivial to port these ideas to an object-oriented paradigm. Especially for semantic merging, this is a nontrivial problem, due to the concepts of late binding and polymorphism in object-oriented programming languages.

More research is also needed for the detection and resolution of *structural merge conflicts* that arise in the presence of restructuring transformations. To this extent, we need to provide more domain-specific information because the information that can be inferred from the source code only is insufficient to decide whether a

structural conflict occurs. In general, more domain-specific information can also be helpful to improve application-specific conflict detection and conflict resolution.

Finally, there is a need for a detailed-but language-independent-taxonomy of the kinds of changes (and corresponding conflicts) that can be made to software. This can lead to more efficient conflict detection and conflict resolution algorithms and to a better understanding of the underlying mechanisms of software evolution in general.

Acknowledgments

The author would like to thank Tom Tourwé, Johan Fabry, and, especially, Kim Mens and Dirk Deridder for providing valuable comments on drafts of this paper. He also thanks Anneliese A. Andrews, editor-in-chief of TSE, as well as the anonymous reviewers, for their useful recommendations. Tom Mens is a Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (F.W.O.-Vlaanderen). This research has been partially funded by the Brussels Region (Brussels Hoofdstedelijk Gewest) in the context of a research project with Getronics and by the Institute for Science and Technology (IWT) in the context of a research project with EDS Belgium and MediaGeniX.

References

[1] E. Adams, W. Gramlich, S.S. Muchnick, and S. Tirfing, "SunPro: Engineering a Practical Program Development Environment," *Proc. Int'l Workshop Advanced Programming Environments,* pp. 86-96, 1986.

[2] U. Asklund, "Identifying Conflicts During Structural Merge," *Proc. Nordic Workshop Programming Environment Research,* pp. 231-242, 1994.

[3] T. Berlage and A. Genau, "A Framework for Shared Applications with Replicated Architectures," *Proc. Conf. User Interface Systems and Technology,* Nov. 1993.

[4] B. Berliner, "CVS II: Parallelizing Software Development," *Proc. The Advanced Computing Systems Professional and Technical Association (USENIX) Conf.,* pp. 22-26, 1990.

[5] V. Berzins, "On Merging Software Extensions," *Acta Informatica,* vol. 23, pp. 607-619, 1986.

[6] V. Berzins, "Software Merge Models and Methods," *Int'l J. System Integration,* vol. 1, no. 2, pp. 121-141, Aug. 1991.

[7] V. Berzins, "Software Merge: Semantics of Combining Changes to Programs," *ACM Trans. Programming Languages and Systems,* vol. 16, no. 6, pp. 1875-1903, 1994.

[8] D. Binkley, S. Horwitz, and T. Reps, "Program Integration for Languages with Procedure Calls," *ACM Trans. Software Eng. and Methodology,* vol. 4, no. 1, pp. 3-35, 1995.

[9] W.R. Bischofberger, T. Kofler, K.-U. Mätzel, and B. Schäffer, "Computer Supported Cooperative Software Engineering with Beyond-Sniff," UBILAB Technical Report 94.9.1, 1994.

[10] J. Buffenbarger, "Syntactic Software Merging," *Software Configuration Management: Selected Papers SCM-4 and SCM-5,* J. Estublier, ed., pp. 153-172, 1995.

[11] B. Bruegge and A.H. Dutoit, *Object-Oriented Software Engineering—Conquering Complex and Changing Systems.* Prentice Hall, 2000.

[12] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys,* vol. 30, no. 2, June 1998.

[13] D. Dampier, Luqi, and V. Berzins, "Automated Merging of Software Prototypes," *Int'l J. System Integration,* vol. 4, no. 1, pp. 33-49, 1994.

[14] W.K. Edwards, "Flexible Conflict Detection and Management in Collaborative Applications," *Proc. Symp. User Interface Software and Technology,* 1997.

[15] C.A. Ellis and S.J. Gibbs, "Concurrency Control in Groupware Systems," *Proc. ACM SIGMOD Conf. Management of Data,* June 1989.

[16] J. Estublier and R. Casallas, "The Adele Configuration Manager," *Configuration Management: Trends in Software,* pp. 135-154, 1994.

[17] M.S. Feather, "Detecting Interference when Merging Specification Evolutions," *Proc. Fifth Int'l Workshop Software Specification and Design,* pp. 169-176, 1989.

[18] M. Fowler, *Refactoring: Improving the Design of Existing Programs,* Addison-Wesley, 1999.

[19] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman 1979.

[20] J.E. Grass, "Cdiff: A Syntax Directed Diff for C++ Programs," *Proc. The Advanced Computing Systems Professional and Technical Association (USENIX) Conf. C++,* pp. 181-193, 1992.

[21] W.G. Griswold, "Program Restructuring as an Aid to Software Maintenance," PhD Thesis, Univ. of Washington, Aug. 1991.

[22] B. Gulla, E.-A. Karlsson, and D. Yeh, "Change-Oriented Version Descriptions in EPOS," *Software Eng. J.,* vol. 6, no. 6, pp. 378-386, 1991.

[23] S. Horwitz, "Identifying the Semantic and Textual Differences Between Two Versions of a Program," *Proc. SIGPLAN '90 Conf. Programming Language Design and Implementation,* pp. 234-244, 1990.

[24] S. Horwitz, J. Prins, and T. Reps, "Integrating Non-Interfering Versions of Programs," *ACM Trans. Programming Languages and Systems,* vol. 11, no. 3, pp. 345-387, 1989.

[25] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 35-46, 1989.

[26] S. Horwitz and T. Reps, "The Use of Program Dependence Graphs in Software Engineering," *Proc. Int'l Conf. Software Eng.,* pp. 392-411, 1992.

[27] J.W. Hunt and M.D. McIllroy, "An Algorithm for Differential File Comparison," Technical Report 41, AT&T Bell Laboratories Inc., 1976.

[28] J.W. Hunt and T.G. Szymanski, "A Fast Algorithm for Computing Longest Common Subsequences," *Comm. ACM,* vol. 20, no. 5, pp. 350-353, 1977.

[29] J.W. Hunt, K.-P. Vo, and W.F. Tichy, "Delta Algorithms: An Empirical Evaluation," *ACM Trans. Software Eng. and Methodology,* vol. 7, no. 2, pp. 192-214, 1998.

[30] IBM, *The System Object Model (SOM) and the Component Object Model (COM): A Comparison of Technologies from a Developer's Perspective,* White Paper, IBM Co., 1994.

[31] D. Jackson and D.A. Ladd, "Semantic Diff: A Tool for Summarizing the Effects of Modifications," *Int'l Conf. Software Maintenance,* 1994.

[32] D.B. Leblang, "The CM Challenge: Configuration Management that Works," *Configuration Management: Trends in Software,* pp. 1-38, 1994.

[33] D.B. Leblang and R.P. Chase, "Computer-Aided Software Engineering in a Distributed Workstation Environment," *Proc. SIGPLAN/SIGSOFT Software Eng. Symp. Practical Software Development Environments, ACM SIGPLAN Notices,* vol. 19, no. 5, pp. 104-112, 1984.

[34] D.B. Leblang, R.P. Chase, and H. Spilke, "Increasing Productivity with a Parallel Configuration Manager," *Proc. Int'l Workshop Software Version and Configuration Control,* pp. 21-38, 1988.

[35] A. Lie, R. Conradi, T.M. Didriksen, and E.-A. Karlsson, "Change-Oriented Versioning in a Software Engineering Database," *Proc. Second Int'l Workshop Software Configuration Management, ACM SIGSOFT Software Eng. Notes,* vol. 14, no. 7, pp. 56-65, 1989.

[36] E. Lippe and N. van Oosterom, "Operation-Based Merging," *Proc. Fifth ACM SIGSOFT Symp. Software Development Environments, ACM SIGSOFT Software Eng. Notes,* vol. 17, no. 5, pp. 78-87, 1992.

[37] D. Lubkin, "Heterogeneous Configuration Management with DSEE," *Proc. Third Int'l Workshop Software Configuration Management,* pp. 153-160, 1991.

[38] J.C.C. McKinsey and A. Tarski, "On Closed Elements in Closure Algebras," *Annals Math,* vol. 47, no. 1, pp. 122-162, Jan. 1946.

[39] T. Mens, "A Formal Foundation for Object-Oriented Software Evolution," PhD Thesis, Dept. Computer Science, Vrije Univ. Brussel, Belgium, 1999.

[40] T. Mens, "Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution," *Proc. Int'l Agtive '99 Conf.,* 2000.

[41] M. Mezini, "Maintaining the Consistency of Class Libraries During their Evolution," *Proc. Object-Oriented Programming Systems, Languages, and Applications, (OOPSLA '97), ACM SIGPLAN Notices,* vol. 32, no. 10, pp. 1-21, 1997.
[42] T. Mikkelsen and S. Pherigo, *Practical Software Configuration Management.* Hewlett-Packard Professional Books, 1997.
[43] J.P. Munson and P. Dewan, "A Flexible Object Merging Framework," *Proc. ACM Conf. Computer Supported Collaborative Work,* pp. 231-241, 1994.
[44] C.M. Neuwirth, R. Chandok, D.S. Kaufer, P. Erion, J.H. Morris, and D. Miller, "Flexible Diff-ing in a Collaborative Writing System," *Proc. ACM Conf. Computer Supported Cooperative Work,* pp. 147-154, Oct. 1992.
[45] W.F. Opdyke, "Refactoring Object-Oriented Frameworks," doctoral dissertation, Technical Report UIUC-DCS-R-92-1759, Univ. of Illinois at Urbana-Champaign, 1992.
[46] K.J. Ottenstein and L.M. Ottenstein, "The Program Dependence Graph in a Software Development Environment," *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Practical Software Development Environments, ACM SIGPLAN Notices,* vol. 19, no. 5, pp. 177-184, May 1984.
[47] D.E. Perry and G.E. Kaiser, "Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems," *Proc. ACM Computer Science Conf.,* pp. 292-299, Feb. 1987.
[48] D.E. Perry, "The Inscape Environment," *Proc. 11th Int'l Conf. Software Eng.,* May 1989.
[49] D.E. Perry, H.P. Siy, and L.G. Votta, "Parallel Changes in Large Scale Software Development: An Observational Case Study," *Proc. Int'l Conf. Software Eng. (ICSE '98),* pp. 251-260, 1998.
[50] T. Reps, "Algebraic Properties of Program Integration," *Science of Computer Programming,* vol. 17, pp. 139-215, 1991.
[51] T. Reps and T. Bricker, "Illustrating Interference in Interfering Versions of Programs," *ACM Software Eng. Notes,* vol. 17, no. 7, pp. 46-55, 1989.
[52] J. Rho and C. Wu, "An Efficient Version Model of Software Diagrams," *Proc. Fifth Asia-Pacific Conf. Software Eng.,* pp. 236-243, 1998.
[53] H.G. Rice, "Classes of Recursively Enumerable Sets and their Decision Problems," *Trans. Am. Math. Soc.,* vol. 89, pp. 25-59, 1953.
[54] C. Rich and R. Waters, *The Programmer's Apprentice.* Addison-Wesley, 1990.
[55] M.J. Rochkind, "The Source Code Control System," *IEEE Trans. Software Eng.,* vol. 1, no. 4, pp. 364-370, 1975.
[56] D. Schefstöm and G. van den Broek, *Tool Integration Environments and Frameworks,* John Wiley & Sons, 1993.
[57] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt, "Reuse Contracts: Managing the Evolution of Reusable Assets," *Proc. OOPSLA '96, ACM SIGPLAN Notices,* vol. 31, no. 10, pp. 268-286, 1996.
[58] J. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press 1977.
[59] W.F. Tichy, "The String-to-String Correction Problem with Block Moves," *ACM Trans. Comput. Syst.,* vol. 2, no. 4, pp. 309-321, 1984.
[60] W.F. Tichy, "RCS: A System for Version Control," *Software Practice and Experience,* vol. 15, no. 7, pp. 637-654, 1985.
[61] W.F. Tichy, "Tools for Software Configuration Management," *Proc. Int'l Workshop Software Version and Configuration Control,* pp. 1-20, 1988.
[62] M. Weiser, "Program Slicing," *IEEE Trans. Software Eng.,* vol. 10, no. 4 pp. 352-357, July 1984.
[63] B. Westfechtel, "Structure-Oriented Merging of Revisions of Software Documents," *Proc. Third Int'l Workshop Software Configuration Management,* pp. 68-79, 1991.
[64] W. Yang, S. Horwitz, and T. Reps, "Detecting Program Components with Equivalent Behaviors," Technical Report 840, Dept. Computer Sciences, Univ. of Wisconsin Apr. 1989.
[65] W. Yang, S. Horwitz, and T. Reps, "A Program Integration Algorithm that Accommodates Semantics-Preserving Transformations," *ACM Trans. Software Eng. and Methodology,* vol. 1, no. 3, pp. 310-354, July 1992.
[66] W. Yang, "Identifying Syntactic Differences between Two Programs," *Software-Practice and Experience,* vol. 21, no. 7, pp. 739-755, 1991.
[67] W. Yang, "How to Merge Program Texts," *J. Systems and Software,* vol. 27, no. 2, pp. 129-135, 1994.
[68] S.S. Yau, J.S. Collofello, and T. MacGregor, "Ripple Effect Analysis of Software Maintenance," *Proc. Int'l Computer Software and Applications Conf.,* pp. 60-65, 1978.

Tom Mens finished his PhD on "A Formal Foundation for Object-Oriented Evolution" in September of 1999. He has been a postdoctoral fellow of the Fund for Scientific Research—Flanders (Belgium) since October 2000. As a researcher in software engineering, he is associated with the Programming Technology Lab of the Vrije Universiteit Brussel. His main research interest lies in the use of formal techniques for improving support for (object-oriented) software evolution and he published several papers on this research topic. In 1998, he was part of the organizing team of the European Conference on Object-Oriented Programming. Until October 2000, he was a scientific advisor for two industrial research projects carried out between the Programming Technology Lab and several industrial partners. At the end of 2000, he cofounded an international scientific research network on *Foundations of Software Evolution* (involving nine research institutes from five different European countries), which he is currently coordinating. In this context, he coorganized an international workshop on *Formal Foundations of Software Evolution* (Lisbon, Portugal, March 2001), as well as a European Conference for Object-Oriented Programming (ECOOP) workshop on *Object-Oriented Architectural Evolution* (Budapest, Hungary, June 2001).