Search-based Model Merging

Marouane Kessentini, Wafa Werda
Computer Science Department
Missouri University of Science and Technology
Rolla, USA
{ marouanek, werdaw}@mst.edu

Philip Langer, Manuel Wimmer
Business Informatics Group
Vienna University of Technology
Vienna, Austria
{langer,wimmer}@big.tuwien.ac.at

ABSTRACT
In Model-Driven Engineering (MDE) adequate means for collaborative modeling among multiple team members is crucial for large projects. To this end, several approaches exist to identify the operations applied in parallel, to detect conflicts among them, as well as to construct a merged model by incorporating all non-conflicting operations. Conflicts often denote situations where the application of one operation disables the applicability of another operation. Whether one operation disables the other, however, often depends on their application order. To obtain a merged model that maximizes the combined effect of all parallel operations, we propose an automated approach for finding the optimal merging sequence that maximizes the number of successfully applied operations. Therefore, we adapted and used a heuristic search algorithm to explore the huge search space of all possible operation sequences. The validation results on merging various versions of real-world models confirm that our approach finds operation sequences that successfully incorporate a high number of conflicting operations, which are otherwise not reflected in the merge by current approaches.

Categories and Subject Descriptors
D.2.9 [**Software Management**]: Software development

Keywords
Search-based software engineering, model-driven software engineering, model evolution, genetic algorithm.

1. INTRODUCTION
Nowadays, software systems are complex and large; therefore, Model-Driven Engineering (MDE) [24] is applied increasingly to cope with the complexity of software systems by raising the level of abstraction. To address the size of software systems, teams of developers have to cooperate and work in parallel on software models. Thus, techniques to support building models collaboratively are highly required.

When models are changed in parallel, they have to be merged eventually to obtain a consolidated model. Therefore, several approaches have been proposed for detecting the operations that have been applied in parallel by developers. Once the applied operations are available, conflict detection algorithms are used to identify pairs of operations that interfere with each other (cf. [16] for a survey on model versioning approaches). In this regard, a conflict denotes a pair of operations, whereas one operation masks the effect of the other (i.e., they do not commute) or one operation disables the applicability of the other. An example for the former is a pair of parallel operations that update the same feature in the model with different values. The latter case is at hand if one operation's preconditions are not valid anymore after applying the operations of the other developer. Such a scenario frequently occurs if composite operations, e.g., model refactorings [4, 28], are applied, because they have potentially complex preconditions that may easily be invalidated by parallel operations.

For resolving conflicts, empirical studies [27] showed that users prefer to work with a tentative merged model acting as a basis for reasoning about possible conflict resolutions, instead of working with the list of operations in terms of choosing to reject one or the other conflicting operation for creating a merged model. A few approaches respect this preference and produce a merged model by applying all non-conflicting operations; conflicting operations are omitted. However, especially in case of a large number of conflicts, many operations are not merged with this strategy, leading to a tentative merged model that lacks in reflecting the *maximal combined effect of the parallel operations*.

With this paper, we address this issue by proposing a method for producing a tentative merged model that reflects the maximum combined effect of the parallel operations by also considering conflicting operations. As mentioned already, an important kind of conflict denotes pairs of operations where one operation disables the applicability of the other. Whether one operation disables the other, however, often depends on the order in which they are applied. Thus, we aim at computing an operation sequence that minimizes the number of disabled operations among all parallel operations, including the conflicting ones. To cope with the huge number of possible operation sequences, a heuristic method is used to explore the space of possible solutions. To this end, we propose to consider model merging as an optimization problem. Our approach takes as input the initial model and the revised ones (i.e., the different parallel versions) and a list of the applied operations, which is computed as described in previous work [25, 26], and generates as output a sequence of operations that minimizes the number of disabled operations. Therefore, we use and adapt genetic algorithm as global heuristic search, which is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution [5].

The remainder of this paper is structured as follows. Section 2 provides the background of model merging and demonstrates the challenges addressed in this paper based on a motivating example. In Section 3, we give an overview of our proposal and explain how we adapted the genetic algorithm to find optimal operation sequences. Section 4 discusses the results

GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands

of the evaluation of our approach. After surveying related work in Section 5, we conclude with some future work in Section 6.

2. MODEL MERGING *CHALLENGES*

In this section, we define briefly the context and concepts of model merging required for this paper. Then we describe the challenges addressed in this paper based on a motivating example.

2.1 Background on Model Merging

In general, two kinds of merge approaches can be distinguished [15]. First, *state-based* merge approaches aim at merging two model versions by combining their model elements into one merged model. Second, *operation-based* merge approaches in contrast do not reason about the models' states, but consider recorded change histories and apply the combination of the parallel histories to the common initial version to compute the merged version.

For both approaches, the notion of conflict is essential, because when having two parallel evolutions of one model, not all changes may be combined to compute one unique merged model. Basically, we can distinguish between two kinds of conflicts. First, two operations are conflicting if one operation *masks* the effect of the other operation in the merged version: e.g., for update/update conflicts, the latter update in the change sequence applied on the model is effective, while the former update is lost. Thus, such conflicting operations are not confluent: different operation sequences result in different models. Second, a conflict also occurs if one operation *disables* the applicability of the other. Every operation has specific preconditions, e.g., an update of an element can only be performed when the element still exists; otherwise a delete/update conflict is raised.

Operation-based merge approaches [17] usually consider besides atomic operations (i.e, additions, deletions, and updates), also *composite operations*, such as *model refactorings*, which consist of a set of atomic operations and additional potentially more complex preconditions. For instance, in the case of model refactorings, certain conditions have to be fulfilled before a refactoring is applicable in order to preserve the semantics of a model after the refactoring has been applied. While in the change history produced by one single developer, the preconditions of the operations are clearly fulfilled, this is not guaranteed when two change histories from two different developers are combined. But this is exactly what is required to perform operation-based merging with the goal of maximizing the combined effect of the parallel operations.

A pure phasing-based approach is in general not solving this problem. For instance, applying first the change history of developer A and afterwards the change history of developer B, only the applicability of the operations of developer A is maximized. But there may be better solutions by intermingling the change histories of developer A and B. Considering all possible permutations of two change histories, we end up with a complexity of *n!* (where *n* is the amount of all applied changes of both developers). Considering the length of change histories in practice, using an enumeration based approach is not feasible.

2.2 Motivating Example

For making the problem statement more concrete, we make use of a motivating example. The starting point is the UML class diagram shown in Figure 1. This version of a person management system has been subject to parallel evolution by two developers who concurrently applied a set of atomic and composite changes.

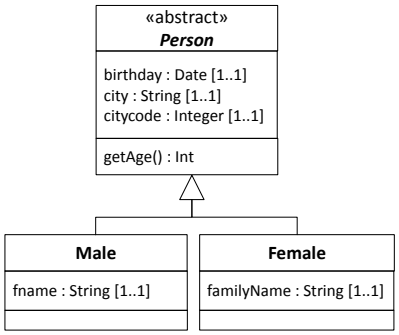

Figure 1: Initial Model v_0.

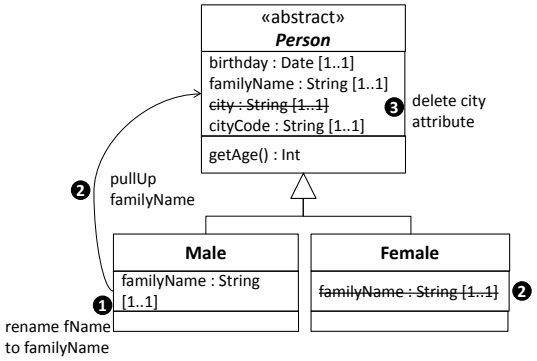

Figure 2: Revised Model v_{1a}.

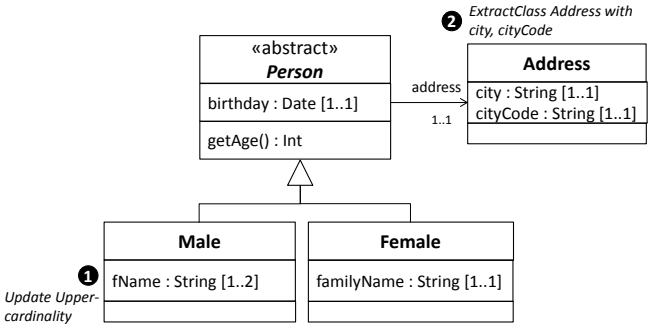

Figure 3: Revised Model v_{1b}.

Developer *A* is first renaming the *fname* attribute of class *Male* to *familyName* in order to apply the *PullUpAttribute* refactoring in a second step: the *familyName* attribute of class *Male* is moved to the class *Person* and the *familyName* attribute of class *Female* is deleted. This refactoring is represented by one composite operation which contains both atomic operations. The precondition of this refactoring is clearly fulfilled: both subclasses of class *Person* have the *familyName* attribute with the same property values, i.e., same data type and multiplicities. Finally, developer A deletes the *city* attribute of class *Person*, because she identifies that this attribute is redundant, as the information is already covered by *cityCode* attribute. The resulting model incorporating all mentioned changes is shown in Figure 2.

Developer *B* changes the upper bound cardinality of the *fName* attribute of class *Male* from one to two, and subsequently, applies the *ExtractClass* refactoring to create an explicit class for the address information. Again, this refactoring is represented by a composite operation consisting of several atomic operations. The resulting model is depicted in Figure 3.

Now a naive operation-based merge approach may apply the operations of developer *A* to the initial model, and subsequently, on this intermediate version, the changes of developer *B*. However, in this sequence, the *ExtractClass* refactoring is not applicable anymore, because the *city* attribute is already missing. Starting with the operations of developer *B* and continuing with the changes of developer *A*, also leads to one disabled operation: the *fName* attribute of class *Male* can be renamed to *familyName*, but the *PullUpAttribute* refactoring cannot be executed, because the *familyName* attributes in both subclasses have different cardinalities. Thus, the precondition of the refactoring is not fulfilled, and consequently, the operation cannot be applied.

Now the question arises whether there are more appropriate solutions in terms of sequences of operations that enable constructing a merged model that maximizes the combined effect of both developers. In our example, 118 different operation sequences have to be considered in addition to the previous two. In fact, solutions exist that allow for applying *all operations successfully* in our example by intermingling the operations of developer *A* and *B*. For instance, executing the *ExtractClass* refactoring, deleting the *cityName* attribute, renaming the *fName* attribute to *lastName*, executing the *PullUpAttribute* refactoring, and finally, setting the upper bound cardinality represents one operation sequence that is fully applicable. For five changes, an enumeration based approach is applicable, but when doubling the changes, we already have to explore 3.6 million combinations because *n!* solutions exist where *n* is the number of changes. Thus, in the next section a more scalable approach to solve this problem is presented.

3. FINDING THE BEST REFACTORING SEQUENCE FOR MODEL MERGING

We now describe our proposal and how we consider merging different model versions as an optimization problem. We start by giving an overview of our approach and provide subsequently a more detailed description on how we adapted and used the genetic algorithm, including the representation of refactorings and refactoring sequences, as well as the fitness function.[1]

3.1 Overview

The goal of our approach is to construct a tentative merged model that maximizes the (at least partial) combined effect of the parallel operations. Therefore, we use a mono-objective optimization algorithm to compute an optimal sequence of merging operations in terms of minimizing the number of refactorings that are disabled by preceding operations.

The general structure of our approach is sketched in Figure 4. The search-based process takes as inputs the sequences of operations that have been applied concurrently to a model by an arbitrary number of developers. These sequences can be detected using operation detection algorithms presented in previous work [25, 26]. Note that these sequences may also be obtained alternatively by tools that record operations directly in the modeling editor. The sequences are composed of operation applications, thereby each entry in a sequence states the operation type as well as the elements on which it has been applied. Having these sequences[2] at hand, we may now combine them into one common sequence of operations and compute the number of disabled operations. Therefore, we use composite operation specifications that contain explicitly specified preconditions in combination with a condition evaluation engine [12] to verify whether the preconditions of each operation in a sequence are fulfilled in a certain state of a model after the preceding operations in the sequence have been applied. If we determine an operation with invalid preconditions in a certain state of the model, we consider this operation to be disabled in the respective operation sequence.

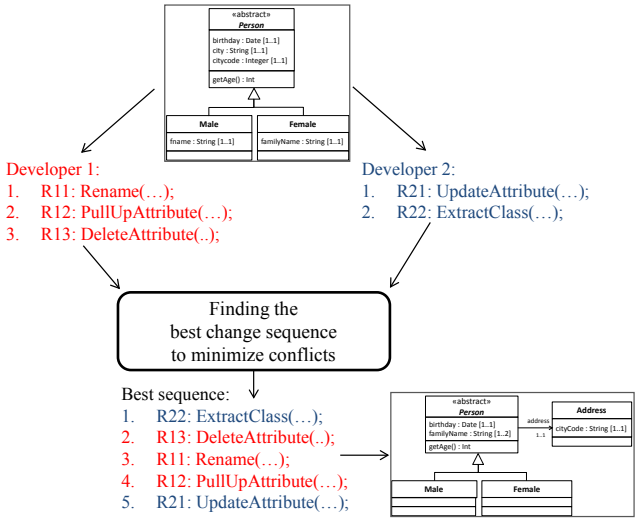

Figure 4: Finding the Best Operation Sequence.

The process of generating a solution can be viewed as the mechanism that finds the best order among all possible operation sequences that minimizes the number of disabled operations. The size of the search space is determined not only by the number of operations applied by the different developers on the same model, but also by the order in which they are applied. Due to the large number of possible refactoring sequences, we considered refactoring merging as an optimization problem. In the next subsection, we describe the adaptation of genetic algorithm to our problem domain.

3.2 Adaptation of the Genetic Algorithm

3.2.1 Genetic Algorithm

By using genetic algorithm (GA), the basic idea is to explore the search space by creating a population of candidate solutions, also called individuals, and evolve them towards an optimal solution for a specific problem.

In GA, a solution can be represented as a vector. Each dimension of this vector must contain symbols that are appropriate for the specific problem. Each individual of the population is evaluated by a fitness function that determines a quantitative measure of its ability to solve the specific problem.

The exploration of the search space is achieved by evolving candidate solutions using selection and genetic operators, such as crossover and mutation. The selection operator chooses the fittest individuals of the current population that are allowed to transmit

[1] In the following, we focus on refactorings, but the presented techniques apply for all kinds of changes in general.

[2] Please note that the length of the operation sequences of the different developers may vary as shown in Figure 4.

their features to new individuals by undergoing crossover and mutation. The crossover operator ensures generation of new children based on parent individuals and allows the transmission of the features of the best fitted parent individuals to new individuals. This is usually achieved by replacing randomly selected dimensions of one parent individual with randomly chosen dimensions from another parent individual to obtain one child. A second child is obtained by inverting parents. Finally, the mutation operator is applied, with a probability that is inversely proportional to its fitness value, to modify some randomly selected nodes in a single individual. The mutation operator introduces diversity into the population and allows escaping local optima found during the search.

Once selection, mutation, and crossover have been applied according to given probabilities, individuals of the newly created generation are evaluated using the fitness function. This process is repeated iteratively until a stopping criterion is met. This criterion usually corresponds to a fixed number of generations. The final result (the best solution found) is the fittest individual produced along all generations.

To adopt GA for solving the problem of finding an operation sequence that maximizes the combined effect of all concurrent operations, we start by constructing an initial GA population that represents possible operation sequences. Then, the main GA fragment explores the search space by constructing new individuals by reordering the operations using the aforementioned crossover and mutation operators, whereas promoting the individual with the best fitness; that is, the operation sequences with the lowest number of disabled operations. The process terminates when the termination criterion (maximum iteration number) is met, and returns the best sequence of operations.

3.2.2 Genetic Algorithm for Model Merging

In the following, we discuss how we adapted the genetic algorithm to solve the model merging problem. Therefore, we present the representation of operation sequences and the applied fitness function to evaluate the quality of operation sequences.

3.2.2.1 Generation of a Population

To represent a candidate solution (individual), we use a vector containing all operations that have been applied by the developers in parallel, whereas each item in the vector represents a single operation (with links to the elements to which it is applied) and the order of operations in this vector represents the sequence in which the operations are applied. Please note that some operations can be eliminated in case they are equivalent; that is, two developers applied the same operation to the same model elements. Thus, we exclude duplicates. Consequently, all vectors, each representing one candidate solution, have the same number of dimensions that corresponds to the number of all parallel operations applied by all developers.

Figure 5 depicts a possible population of operation sequences for the running example introduced above, whereas *R*** refers to the label of the respective refactoring introduced in Figure 4. For instance, the solutions represented in Figure 5 are composed of five dimensions corresponding to five operations proposed by two different developers. All the solutions have the same length, but they constitute a different order.

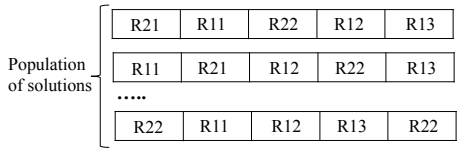

Figure 5: A Population of Operation Sequences.

The proposed algorithm first generates a population randomly from the list of all operations. Second, crossover and mutation operators are used to generate new populations in the next iterations as explained in the following.

3.2.2.2 Generating New Populations

We used the principle of the Roulette wheel [5] to select individuals for mutation and crossover. The probability to select an individual for crossover and mutation is directly proportional to its relative fitness in the population. In each iteration, we select *population_size / 2* individuals from the population pop_i to form population pop_{i+1}. These (*population_size / 2*) selected individuals will "give birth" to another (*population_size / 2*) new individuals using a crossover operator. Therefore, two parent individuals are selected, and a sub-vector is picked on each one. Then, the crossover operator swaps the dimensions and their relative sub-vector from one parent to the other. When applying the crossover, we ensure that the length of the vector remains the same. The crossover operator allows creating two offspring p'_1 and p'_2 from the two selected parents p_1 and p_2. It is defined as follows: a random position, k, is selected. The first k operations of p_1 become the first k elements of p'_1. Similarly, the first k operations of p_2 become the first k operations of p'_2.

The mutation operator can be applied to pairs of dimensions selected randomly. Given a selected solution, the mutation operator first randomly selects one or many pairs of dimensions of the vector. Then, for each selected pair, the dimensions are swapped.

3.2.2.3 Evaluating Generated Solutions

The quality of an individual is computed based on the quality of its represented operation sequence. In each operation sequence, different operations may be disabled after certain preceding operations have been applied. Thus, the quality of a solution is determined with respect to the number of disabled operations within a certain operation sequence. In other words, the best solution is the one that minimizes the number of disabled operations, obtained by applying the respective operation sequence to the initial model.

The evaluation of an individual is formalized in terms of a mathematical function called fitness function. The fitness function quantifies the quality of the proposed solution. The goal is to define an efficient and simple—in the sense that it is not computationally expensive—fitness function in order to reduce the computational complexity. This function should evaluate the number of disabled operations. Therefore, we use previously developed tools to specify composite operations including their preconditions in combination with an engine for evaluating the conditions in a certain model state (cf. [12] for details). As evaluating conditions can be rather expensive, we only compute whether one operation in a sequence disables another for each possible *pair-wise combination* of operations in advance, instead of checking the preconditions of each operation with the combined effect of every operation that precedes the operation in the sequence. Please note that this might miss to detect some

disabled operations in certain scenarios: the preconditions of an operation might be valid with each *single* preceding operation, but the preceding operations *in combination* might still invalidate the preconditions of the subsequent operations. As this is a rather rare case, we left this issue as a topic of future work.

The information on which operation in a sequence disables the other is represented in terms of a matrix $n \times n$ where n is the number of operations applied originally by the different developers in total (after eliminating duplicates). Each item in this matrix represents a combination of two operations and holds a value of either 0 or 1: if an operation i disables the operation j then, the item (i,j) in the matrix takes the value 1, otherwise it takes 0. Based on this matrix, we may determine easily the number of disabled operations for a specific operation sequence by summing up all values in the matrix.

To illustrate the fitness function, we consider one solution (cf. Figure 6) from our running example to evaluate. Since 5 operations have been applied in parallel, the computed matrix has a size of 5 × 5. To determine the number of disabled operations for the given sequence, we iterate over the operations of the sequence and sum the values of the matrix at the items representing current operation (row) and all operations that are after that operation in the sequence (column). For the example, the fitness function is $f_i = 2$.

Possible solution:

	R11	R12	R13	R21	R22
R11	-	0	0	0	0
R12	0	-	0	0	0
R13	0	0	-	0	1
R21	0	1	0	-	0
R22	0	0	0	0	-

Figure 6: An Illustration of the Fitness Function.

4. EVALUATION

To evaluate our proposal, we conducted experiments on six real-world models. We present first the objectives of this exploratory study and then we describe and discuss the obtained results. For replication purposes, the experimentation material can be downloaded from [7].

4.1 Research Questions and Experimental Setting

The study was conducted to quantitatively assess the performance of our approach when applied to real-world scenarios. Thereby, we aimed at answering the following research questions (RQ).

- ***RQ1.*** *To what extent can the proposed approach reduce the number of disabled operations?*
- ***RQ2.*** *To what extent the best merging solutions make sense semantically?*

To answer RQ1 and RQ2, we used a corpus containing an extensive evolution of multiple open source systems. Furthermore, we compared our results to those produced without taking into consideration the order of operations. We also investigated manually the semantics of merged models after applying the best solution generated by our algorithm. In the following, we describe details about our experiments' setting.

We chose to analyze the extensive evolution of three Ecore metamodels coming from the Graphical Modeling Framework (GMF) [8], an open source project for generating graphical modeling editors. We considered the evolution from GMF's release 1.0 over 2.0 to release 2.1 covering a period of two years. For achieving a broader data basis, we analyzed the revisions of three models, namely the Graphical Definition Metamodel (GMF Graph), the Generator Metamodel (GMF Gen), and the Mappings Metamodel (GMF Map). Therefore, the respective metamodel versions had to be extracted from GMF's version control system and, subsequently, manually analyzed to determine the actually applied operations between successive metamodel versions. In addition to GMF, we used model fragments extracted from three open source projects: GanttProject (Gantt for short) [9], JHotDraw [10], and Xerces-J [11]. We considered the evolution across three versions of Gantt (v1.7, v1.8, and v1.9.10), three versions of JHotDraw (v5.1, 5.2, and 5.3) and four versions of Xerxes-J (v1.4.4, v2.5.0, v2.6.0, and v2.6.1). Table 1 summarizes for each model evolution scenario the number of applied refactorings, as well as the number of model elements for the smallest and largest model version.

Additionally, we had to specify all operation types (i.e., their comprised atomic operations and preconditions) that have been applied across all versions leading to 38 different types of operations. The evolution of the analyzed models provides a relatively large set of revisions containing overall 401 different applications of the operation types as shown in Table 1.

Due to the lack of existing parallel revision histories that we could have used for evaluating our approach, we *emulate parallel evolution* by dividing the applied operations from the single revisions into parallel sequences of operations manually and asked five graduate students to additionally modify different model fragments of these open source systems in order to cause disabled operations in the considered evolutions.

To assess the accuracy of our approach, we evaluate manually the number of conflicts (NC), i.e., number of disabled operations of the best operation sequence found by our approach, as well as the correctness of the found operation sequences.

Automatic correctness (AC) consists of comparing the suggested operation order to the expected one, operation by operation. AC method has the advantage of being automatic and objective. However, since different possibilities exist to avoid disabled operations, AC could reject a good solution because it yields different operation sequences from the original one. To account for those situations, we also use manual correctness (MC) for evaluating the correctness of the proposed sequence, again operation by operation. When calculating MC, we verify whether the proposed operation sequence preserves the semantics of the design (and not only avoids disabled operations).

Table 1: Number of Refactoring Operations.

Model	*Number of refactorings*	*Number of elements (min, max)*
GMF Map	14	367, 428
GMF Graph	36	277,310
GMF Gen	112	883,1295
GanttProject	72	451, 572
Xerces-J	86	1698,1732
JHotDraw	81	985, 1457

We compared our results with: *(i)* a randomly generated order (random search), *(ii)* a traditional technique where internal sequence is not considered (we apply operations originally applied by developer 1 then developer 2, etc.), and *(iii)* another local search algorithm (Simulated Annealing) [6].

We used the AC and NC scores for the comparison. The different scores (AC, MC, and NC) are calculated as an average of 30 runs to ensure the stability of the results. In addition, the comparison between SA, random search and GA is also based on an average of 30 runs.

4.2 Results and Discussions

Table 2 summarizes our findings over an average of 30 runs. In general, using our search-based approach, we were able to reduce the number of conflicts. For instance, in JHotDraw only 11 disabled operations are detected after executing the optimal operation sequence. However, 37 disabled operations are detected by executing the operations as they appear without altering the internal order provided by different developers. Thus, we reduced the number of disabled operations by 70% (1 - 11/37). Similar results are obtained on remaining open-source systems where the number of conflicts is reduced by 63% (1 - 3/8), 69% (1 - 5/16), 68% (1 - 16/49), 43% (1 - 19/33), 59% (1 - 12/29) for GMF Map, GMF Graph, GMF Gen, GanttProject, and Xerces-J, respectively. Thus, we conclude that our proposal reduces significantly the number of disabled operations.

Table 2: Number of Disabled Operations (NC).

Systems	Number of disabled operations (with heuristic search)	Number of disabled operations (without heuristic search)
GMF Map	3	8
GMF Graph	5	16
GMF Gen	16	49
GanttProject	19	33
Xerces-J	12	29
JHotDraw	11	37

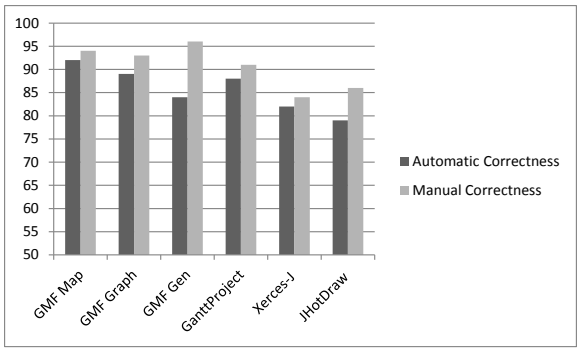

Figure 7: AC and MC Scores for the Case Studies.

Figure 7 illustrates the AC and MC scores on the different models. For the automatic precision, we compared the generated operation order to the expected (original) one provided manually by a group of five developers. The AC was more than 80% for all the various models. The lower AC score was found in JHotDraw. JHotDraw was extensively modified within only one large revision comprising 27 refactoring operations that will potentially lead to disabled operations when divided and applied in parallel. Thus, the evolution of this model is a very representative mixture of different scenarios for the application of operations leading to many disabled operations (as described in Table 2). However, our approach finds approximately the best order of operations to avoid disabled ones with an acceptable score (more than 80%) even if the number of all operations and disabled ones is large.

Similar to JHotDraw, Xerces-J versions were extensively revised leading to a huge list of operations. This can explain also the AC obtained by our approach (82%). The remaining models contain less model elements and revisions, thus AC scores were higher with more than 85%. For instance, the evolution of GMF Map contained four revisions, having in each revision a maximum number of three refactoring operations. Using our approach, we could find almost a very similar order to the reference one. We noticed that our technique does not have a bias towards the types of operations since most of them were used in the suggested solutions.

With regards to MC, the score for all the six models were improved since we found interesting operation order alternatives that deviate from the reference ones proposed by the experts: for all the six models, we obtained approximately more than 85% as MC, e.g., for GMF Graph 94% and for GMF Gen 96%. When we manually analyzed the results, we found that many disabled operations can be avoided in different manners and sometimes changing the order of some operations does not affect the number of disabled operations. In the context of this experiment, we conclude that our technique was able to find the best order of merging operations that reduces the disabled ones.

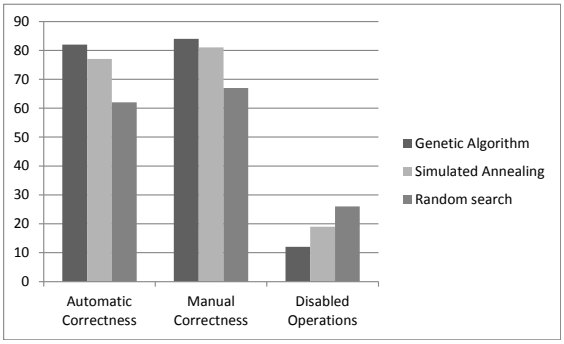

Figure 8: Comparison: Genetic Algorithm, Simulated Annealing, and Random Search for XercesJ.

In Figure 8, we compared our genetic algorithm (GA) results with a random search (without the use of change operators) and a local search algorithm, namely simulated annealing (SA). The local search starts from only one solution (instead of a population of solutions) generated randomly and then refined using mutation in the next iterations. The AC, MC, and NC scores for SA were also acceptable. However, GA performs better than SA in the case of Xerces-J, because Xerces-J exhibits a larger model and a larger list of applied refactorings, and GA usually provides better results than SA in scenarios having a larger search space.

The correctness results might vary depending on the operation sequence which is randomly generated, though guided by a meta-heuristic. To ensure that our results are relatively stable, we compared the results of multiple executions of GA as shown in Figure 9. We consequently believe that our technique is stable, since the AC and MC precision scores are approximately the same for several executions.

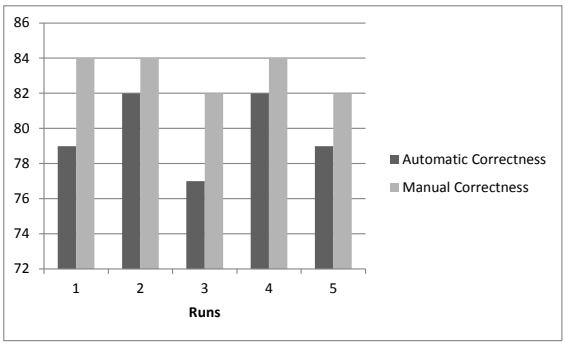

Figure 9: AC and MC Scores for Multiple Executions (5 runs).

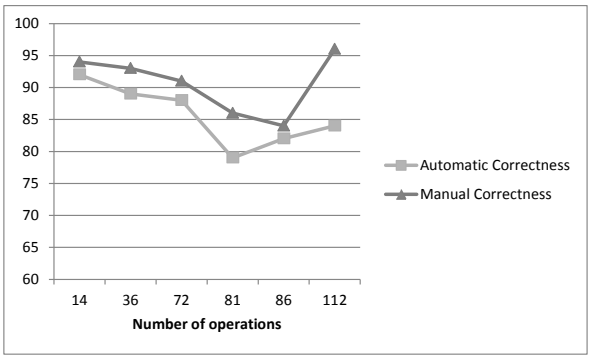

Figure 10: AC and MC in Comparison to Number of Operations.

Figure 10 analyzes the correlation between the number of operations and the correctness values. More precisely, we sort AC and MC based on the number of refactorings for each open source system described in Table 1. From this data, we conclude that AC and MC are not necessarily affected negatively by a larger number of refactorings. For example, MC even increases from 85% to 96% when the number of refactorings increases from 86 to 112. Thus, we can conclude that our proposal shows a good scalability and is not affected negatively by the number of refactorings. However, when the number of operations increases, it does not necessarily mean that the number of disabled operations does.

5. RELATED WORK

With respect to the contribution of this paper, namely to integrate two parallel operation histories into one operation sequence that maximizes the number of successfully applied operations, related work dates back to the early 1990ies. Before that time, merging has been mostly achieved based on the states of the artifacts under version control [23]. The origin work on operation-based merging has been published by Lippe [14]. He pointed out several advantages of operation-based merging over purely state-based merging and contributed the important notion of frontier set. The frontier set, including frontier points, is an indicator how far one can merge two sequences of changes. One goal is to shift the frontier points as far away from the original model version as possible to maximize the applicability of the performed operations. One way to shift the frontier points is to reorder the atomic operations, i.e., to apply all non-conflicting atomic operations before the conflicting ones. What we have contributed with our search-based approach is a mechanism to minimize the critical points in the merge process where users have to be involved, even when composite operations, such as refactorings, have been applied.

Operation-based merging has been heavily applied in asynchronous collaborative graphical editing. Edwards [13] has defined several strategies for combining two operation sequences into one sequence. The strategies range from fully automatic merging by computing each possible sequence of non-conflicting changes to interactive merging allowing the user to decide how each change of a conflicting change pair should be incorporated in the finally merged model. Ignat and Norrie [15] have compared an operation-based approach and a state-based approach for merging change logs of collaborative graphical editors. They distinguish "real" conflicts from resolvable conflicts. The latter may be resolved by finding an appropriate order to incorporate the changes to the finally merged model. For finding the appropriate order, priority lists for change types have to be defined.

With the advent of MDE, the research topic collaborative modeling is gaining momentum. Several state-based approaches for model versioning have been proposed (cf. [16, 23] for an overview), as well as a few operation-based approaches. Koegel et al. [18] record changes in modeling editors and provide conflict detection for two sequences of recorded changes. They also support composite changes, but only consider how these operations are build up from atomic changes while explicit preconditions are disregarded. If they detect that a composite change is in conflict with an atomic change, they let the user decide which one to take. Similarly, Barret et al. [19] discuss pushing the frontier points as far as possible by incorporating all non-conflicting changes to produce a merged model and then let again the user decide which change of a conflict pair to prioritize. Other operation-based approaches for models have been presented in [20, 22], but no dedicated reordering strategies have been discussed. In [21], the authors mention that finding an appropriate sequence for unifying the changes of two parallel change sets is an optimization problem, but they based their approach on manual conflict resolution during the merge process.

In summary, reasoning on arbitrary application orders of the operations to unify (including composite changes) to find the order that is maximizing the successful application of the operations is not considered by existing operation-based merge approaches. State-of-the-art approaches mostly reside on a two-phase process: first, they apply the non-conflicting changes and then let the user select the change to be prioritized out of two conflicting changes. In contrast, our approach explores arbitrary sequences and the result is the most applicable sequence of operations found by the genetic algorithm. Thus, we are able to minimize the critical and labor-intensive tasks involving user interaction in the merge process going beyond existing state-of-the-art approaches.

Our proposal is part of the search-based software engineering (SBSE) contributions [29]. SBSE uses search-based approaches to solve optimization problems in software engineering. Based on surveys proposed by Harman et al. [29, 30], our work represents the first attempt to treat the problem of model merging as a combinatorial optimization problem.

6. CONCLUSION

This paper proposes a novel approach for merging parallel versions of models by finding the best operation sequence. Such a sequence is very useful in model versioning to find a tentative merge, as a basis for subsequently resolving the remaining

conflicts manually. Therefore, a merged model is necessary that maximizes the combined effect of all operations that have been applied by multiple developers in parallel to the same model. This is achieved by finding an optimal (potentially intermingled) order of operations that minimizes the number of disabled operations. As the search space in terms of all possible sequences of operations is potentially huge, we consider the merging process as an optimization problem.

We evaluated our proposal with six real-world models extracted from different open source systems. The experiment results indicate clearly that the number of disabled operations is reduced significantly in comparison to the number of disabled operations without taking into consideration the different possible operation orders. We further evaluated successfully that the computed operation sequences lead to correct models in terms of their semantics in most of the considered cases.

Although our approach has been evaluated with real-world models with a reasonable number of applied operations, we are working now on larger models and with larger list of operations applied in parallel. This is necessary to investigate more deeply the applicability of the approach in practice, but also to study the performance of approach when dealing with very large models. Moreover, we plan to investigate an empirical study to compare with other search-based algorithms. More generally, we plan to extend this work by fixing detected conflicts since this work focus only on minimizing the number of disabled operations.

7. REFERENCES

[1] D. Dig, K. Manzoor, R. E. Johnson, T. N. Nguyen. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Transactions on Software Engineering,* 34(3):321-335, 2008.

[2] T. Ekman, U. Asklund. Refactoring-aware Versioning in Eclipse. *Electronic Notes in Theoretical Computer Science* 107:57-69, 2004.

[3] M. Koegel, M. Herrmannsdoerfer, Y. Li, J. Helming, D. Joern. Comparing State- and Operation-based Change Tracking on Models. In *Proceedings of EDOC,* 2010.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring – Improving the Design of Existing Code. 1st ed. Addison-Wesley, 1999.

[5] J. R. Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection. *MIT Press*, Cambridge, 1992.

[6] S. Kirkpatrick, C. D. Jr. Gelatt, M. P. Vecchi. Optimization by simulated annealing. *Sciences,* 220(4598):671–680, 1983.

[7] http://web.mst.edu/~marouanek/gecco13/merging

[8] www.eclipse.org/gmf

[9] http://www.ganttproject.biz

[10] http://www.jhotdraw.org

[11] http://xerces.apache.org/xerces-j

[12] P. Langer. Adaptable Model Versioning based on Model Transformation By Demonstration. *PhD Thesis, Vienna University of Technology*, 2011.

[13] W. K. Edwards: Flexible Conflict Detection and Management in Collaborative Applications. In *Proceedings of Symposium on User Interface Software and Technology*, pages 139-148, 1997.

[14] E. Lippe, N. van Oosterom. Operation-based merging. In *Proceedings of SDE*, pages 78-87, 1992.

[15] C. Ignat and M. C. Norrie. Operation-based versus State-based Merging in Asynchronous Graphical Collaborative Editing. In *Workshop on Collaborative Editing,* 2004.

[16] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, M. Wimmer: An Introduction to Model Versioning. In *Proceedings of SFM,* 2012.

[17] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel. The Past, Present, and Future of Model Versioning; *Emerging Technologies for the Evolution and Maintenance of Software Models*, IGI Global, 2011.

[18] M. Koegel, H. Naughton, J. Helming, M. Herrmannsdoerfer. Collaborative model merging. In *OOPSLA Companion,* 2010.

[19] S. Barrett, P. Chalin, G. Butler. Table-Driven Detection and Resolution of Operation-Based Merge Conflicts with Mirador. In *Proceedings of ECMFA,* 2011.

[20] C. Schneider, A. Zündorf, J. Niere. CoObRA - a small step for development tools to collaborative environments. In *Workshop on Directions in Software Engineering Environments,* 2004.

[21] M. Schmidt, S. Wenzel, T. Kehrer, U. Kelter. History-based Merging of Models. In *Workshop on Comparison and Versioning of Software Models*, 2009.

[22] A. Mougenot, X. Blanc, M. Gervais. D-Praxis: A Peer-to-Peer Collaborative Model Editing Framework. In Proceedings of *DAIS,* 2009.

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

[24] J. Bézivin. On the Unification Power of Models. *Software and Systems Modeling,* 4(2):171-188, (2005).

[25] A. ben Fadhel, M. Kessentini, P. Langer, M. Wimmer. Search-based Detection of High-level Model Changes. In *Proceedings of ICSM*, 2012.

[26] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdoerfer, M. Seidl, K. Wieland, G. Kappel. A Posteriori Operation Detection in Evolving Software Models. *Journal of Systems and Software*, 86(2):551-566, 2013.

[27] K. Wieland, P. Langer, M. Seidl, M. Wimmer, G. Kappel. Turning Conflicts into Collaboration - Concurrent Modeling in the Early Phases of Software Development. *Computer Supported Cooperative Work*, 22(2-3):181-240, 2013.

[28] G. Sunyé, D. Pollet, Y. Le Traon, J. M. Jézéquel. Refactoring UML Models. In *Proceedings of UML*, 2001.

[29] M. Harman. The current state and future of search based software engineering. In *Proceedings of ICSE*, 2007.

[30] M. Harman, S. A. Mansouri, Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys* 45(1):Article11, 2012.