RZ 3727 (# 99737) 02/06/2009
Computer Science 23 pages

Research Report

Dependent and Conflicting Change Operations of Process Models

Jochen M. Küster[1], Christian Gerth[1,2], and Gregor Engels[2]

[1]IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

E-mail: {jku,cge}@zurich.ibm.com

[2]Department of Computer Science
University of Paderborn
Germany

E-mail: {gerth,engels}@upb.de

LIMITED DISTRIBUTION NOTICE

This report will be distributed outside of IBM up to one year after the IBM publication date.
Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

IBM Research
Almaden · Austin · Beijing · Delhi · Haifa · T.J. Watson · Tokyo · Zurich

Dependent and Conflicting Change Operations of Process Models

Jochen M. Küster[1], Christian Gerth[1,2], and Gregor Engels[2]

[1] IBM Zurich Research Laboratory, Säumerstr. 4
8803 Rüschlikon, Switzerland {jku,cge}@zurich.ibm.com
[2] Department of Computer Science, University of Paderborn, Germany
{gerth,engels}@upb.de

Abstract. Version management of models is common for structural diagrams such as class diagrams but still challenging for behavioral models such as process models. For process models, conflicts of change operations are difficult to resolve because often dependencies to other change operations exist. As a consequence, conflicts and dependencies between change operations must be computed and shown to the user who can then take them into account while creating a consolidated version. In this paper, we introduce the concepts of dependencies and conflicts of change operations for process models and provide a method how to compute them. We then discuss different possibilities for resolving conflicts. Using our approach it is possible to enable version management of process models with minimal manual intervention of the user.

1 Introduction

Version management of models is a crucial technique for enabling modeling in distributed modeling scenarios and has recently been identified as one challenge in model management [8]. In general, it requires to compute and visualize changes that have been performed on a common source model while creating different versions. Based on this, version management capabilities then have to enable the user to create a consolidated model, by accepting or rejecting changes and thereby modifying the original source model.

A key requirement for consolidation of changed models is that it should impose minimal manual overhead on the user: Otherwise, a straightforward solution would be that the user remodels all changes manually. Nowadays, version management is a common functionality of mainstream modeling tools such as the IBM Rational Software Architect [15]. However, for behavioral models such as process models, inspecting and accepting or rejecting changes can involve quite some overhead if the changes to be dealt with are numerous. One reason for this is that the semantics of behavioral models is usually more complex than for structural models. A straightforward approach to compute all changes on model elements (called elementary changes) and display them is difficult to handle for the user: typically, elementary changes cannot be considered in isolation but must be aggregated to compound changes [14, 25].

To enable a high degree of automation within consolidation of changes, it is important to understand dependencies and conflicts of changes. Informally, if two changes are dependent, then the second one requires the application of the first one. If two changes

are in conflict, then only one of the two can be applied. Other than in structural models, in behavioral models changes are often dependent on one another. As a consequence, an approach for computing dependent and conflicting compound changes is required. Further, once conflicts have been computed, techniques for resolving conflicts are needed that take into account the characteristics of the modeling language.

In this paper, we study dependencies and conflicts of compound changes for process models. We first capture each of our compound change operations as a model transformation and then compute critical pairs [3, 9, 10] which can be used for detecting dependent and conflicting transformations. We then show how the results from critical pair analysis can be encoded as conditions which enable fast checks for dependencies and conflicts. We extend dependencies and conflicts to change sequences and provide a means of breaking up a change sequence into individual subsequences such that they can be dealt with separately in the conflict resolution process. For conflict resolution, we propose several resolution options that take into account characteristics of compound change operations. Using our approach, dependencies and conflicts of compound change operations in change logs can be computed and displayed to the user. In an evaluation we show that our approach leads to considerable less dependencies and conflicts and also to less user intervention for inspecting and resolving changes compared to an approach based on elementary changes.

The paper is structured as follows. First, in Section 2 we introduce our example scenarios that we obtain when performing process modeling in a distributed environment. In Section 3, we discuss how dependencies and conflicts of change operations can be defined and computed. In Section 4, we extend the notion of dependency and conflict to change sequences and in Section 5 we present our approach to conflict resolution. Section 6 reports on tool support and an evaluation of our approach. Finally, we discuss related work and future work.

2 Background

In this section, we introduce our case study motivated by process modeling in the IBM WebSphere Business Modeler (WBM) [1]. Figure 1 shows an example business process model V from the insurance domain. The language supported by WBM has similarities to UML 2.0 Activity Diagrams [20]: Nodes can be *Actions* or *ControlNodes* where *ControlNodes* contain Decision and Merge, Fork and Join, InitialNodes and FinalNodes. Nodes are connected by control flow as it is known from UML Activity Diagrams. In the example in Figure 1, an insurance claim is first checked, then it is recorded and then a decision is made whether to settle or reject it.

In a distributed modeling scenario, the process model V might have been created by the process model representative in an enterprise and then given to two colleagues for further elaboration. During this elaboration period, one colleague creates model V_1 and the other one model V_2. Afterwards, the process model representative is faced with the task of inspecting each change and then either accepting or rejecting it.

A common approach for version management of models is to capture possible operations performed on the model. For behavioral models such as process models, it is possible to design compound change operations that transform a model from one con-

sistent state into a new consistent state. Following this idea, we have previously proposed compound change operations for process models [14] as follows: *InsertAction*, *DeleteAction* or *MoveAction* operations allow to insert, delete or modify actions and always produce a connected process model as output. Each of the operations consists of several elementary changes such as creating a new action and redirecting source and targets of the edges. Similarly, *InsertFragment*, *DeleteFragment* and *MoveFragment* operations can be used for inserting, deleting or moving a complete fragment of the process model. Here, a fragment can either be an alternative fragment consisting of a Decision and a Merge node, a concurrent fragment consisting of a Fork and a Join node or further types of fragments including unstructured or complex fragments which allow to express all combinations of control nodes [14].

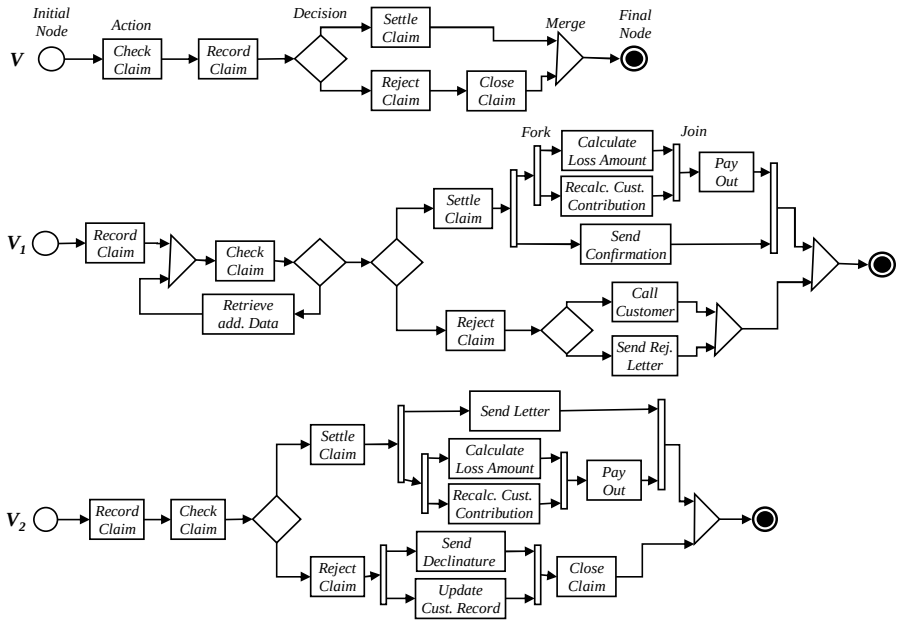

Fig. 1. Example

Action Mapping (V_1, V_2)
- "Retrieve add. Data" – ""
- "Calculate Loss Amount" – "Calculate Loss Amount"
- "Recalc. Cust. Contribution" – "Recalc. Cust. Contribution"
- "Pay Out" – "Pay Out"
- "Send Confirmation" – "Send Letter"
- "Call Customer" – ""
- "Send Rej. Letter" – "Send Declinature"
- "" – "Update Cust. Record"

Fig. 2. Action mapping between V_1 and V_2

For the following discussions, we assume knowledge about newly introduced action nodes that are supposed to be identical in different versions, captured by a mapping of identical actions shown in Figure 2.

In addition, we assume that each sequence of change operations is recorded in a change log. This change log describes the change operations performed on the source model to obtain the target model and can either be logged during editing or reconstructed by comparing source and target model, proposed in [14]. We further assume that this change log is clean, i.e. it does not contain unnecessary

change operations that are later in the change log overridden [21]. In Figure 3, two change logs are given: $\Delta(V, V_1)$ describes the sequence of change operations for obtaining V_1 from V and $\Delta(V, V_2)$ describes the sequence of change operations for obtaining V_2 from V. For example, *InsertAlt.Fragment*$(F_A$, *"Reject Claim"*, *"Close Claim"*$)$ introduces a new alternative fragment called F_A between the nodes *"Reject Claim"* and *"Close Claim"*.

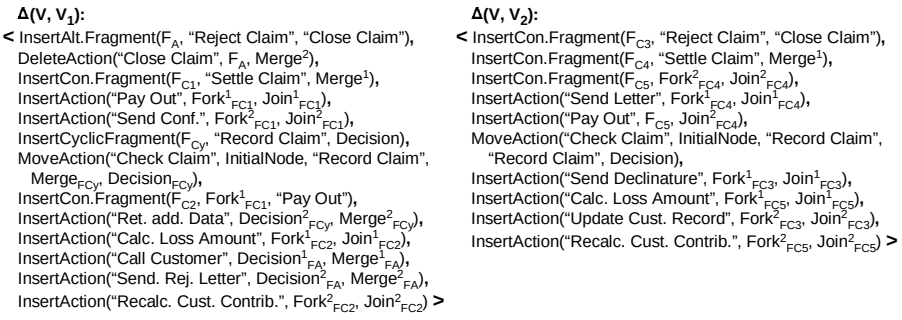

Fig. 3. Change logs $\Delta(V, V_1)$ and $\Delta(V, V_2)$

For the following discussion, we distinguish between two scenarios: In the *single user scenario*, a sequence of change operations is performed on a model V, obtaining model V_1. Afterwards this sequence of change operations needs to be displayed to the user and for each change the user either has to confirm or reject it. In the *multi-user scenario*, two sequences of change operations are performed concurrently on V, leading to V_1 and V_2. Afterwards, all change operations are reconsidered and either rejected or confirmed.

Requirements for both scenarios are that the application of changes should be automatic and involve minimal user interaction. This requires that the change operations can be executed automatically and requires the validity of their parameters: If the parameters are invalid then a change operation becomes non-applicable. For this purpose, it is important that dependencies between change operations in the change sequences are known: The rejection of one operation can turn other operations non-applicable. For example, the rejection of an *InsertAlt.Fragment* operation leads to the non-applicability of all operations operating on this fragment. In addition, in the multi-user scenario, conflicts need to be identified because it is impossible to apply both operations that are in conflict without adaptation. For example, *InsertAlt.Fragment*$(F_A$, *"Reject Claim"*, *"Close Claim"*$)$ and *InsertCon.Fragment*$(F_{C3}$, *"Reject Claim"*, *"Close Claim"*$)$ are in conflict because either an alternative or a concurrent fragment is inserted at the same position. This means that once one of the change operations has been chosen the other one becomes non-applicable. In the following, we first provide a concept for dependencies and conflicts of change operations and then proceed to conflict resolution.

3 Dependencies and Conflicts of Change Operations

In this section, we establish the notions of dependencies and conflicts of change operations and discuss how to compute them. We first formalize change operations using graph transformations and then compute potential dependencies and conflicts of change operations.

3.1 Metamodel and Change Operations

Change operations can be formalized over a process model metamodel as has been done previously for other model transformation rules. We assume a business process model defined by the simplified metamodel shown in Figure 4 consisting of nodes connected by edges. Nodes can be *Actions* or *ControlNodes* or *Fragments*. *ControlNodes* contain Decision and Merge, Fork and Join, InitialNodes and FinalNodes. We assume that the metamodel is restricted by constraints and in particular that for *Actions*, only at most one incoming and outgoing edge is allowed. Fragments are an extension that allow us to represent a decomposition of the process model which can be computed using existing algorithms [24]. Fragments can be used for various analysis purposes such as control and data flow analysis but are also beneficial in the context of version management because they allow to detect and specify compound changes [14].

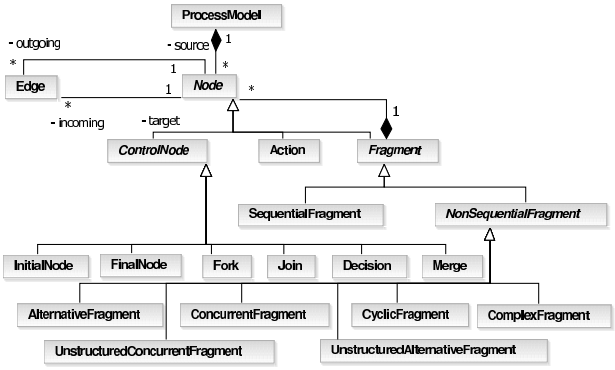

Fig. 4. Metamodel for process models

Each change operation c on a model V can be viewed as a model transformation rule which can be formalized as a typed attributed graph transformation rule [10, 13, 18] where the type graph represents the metamodel. A typed graph transformation rule $p : L \rightarrow R$ consists of a pair of typed instance graphs L, R such that the union is defined. A graph transformation step from a graph G to a graph H, denoted by $G \overset{p(o)}{\Longrightarrow} H$, is given by a graph homomorphism $o : L \cup R \rightarrow G \cup H$, called occurrence, such that the left hand side is embedded into G and the right hand side is embedded into H and precisely that part of G is deleted which is matched by elements of L not belonging to R, and, that part of H is added which is matched by elements new in R. For a rule p, an inverse rule p^{-1} can be constructed that inverts the transformation defined by p.

The change operations used in Figure 3 are specified as graph transformation rules in Figure 5. The *InsertAction* operation inserts a new *Action* between two existing nodes and also reconnects the process model such that it stays connected. For this purpose, the left hand side of the rule matches a fragment f and two nodes a and b connected by an

edge e. It then creates a new *Action* x and a new edge $e2$ and redirects the target of the edge $e1$ to be the new *Action*. In a similar way, *DeleteAction* and *MoveAction* delete an action or move an action, respectively. Fragment operations are used for inserting, deleting or moving a fragment of the process model. Note that fragments can be concurrent fragments or alternative fragments or of a further type. Details about the fragment structure are left out here for simplification.

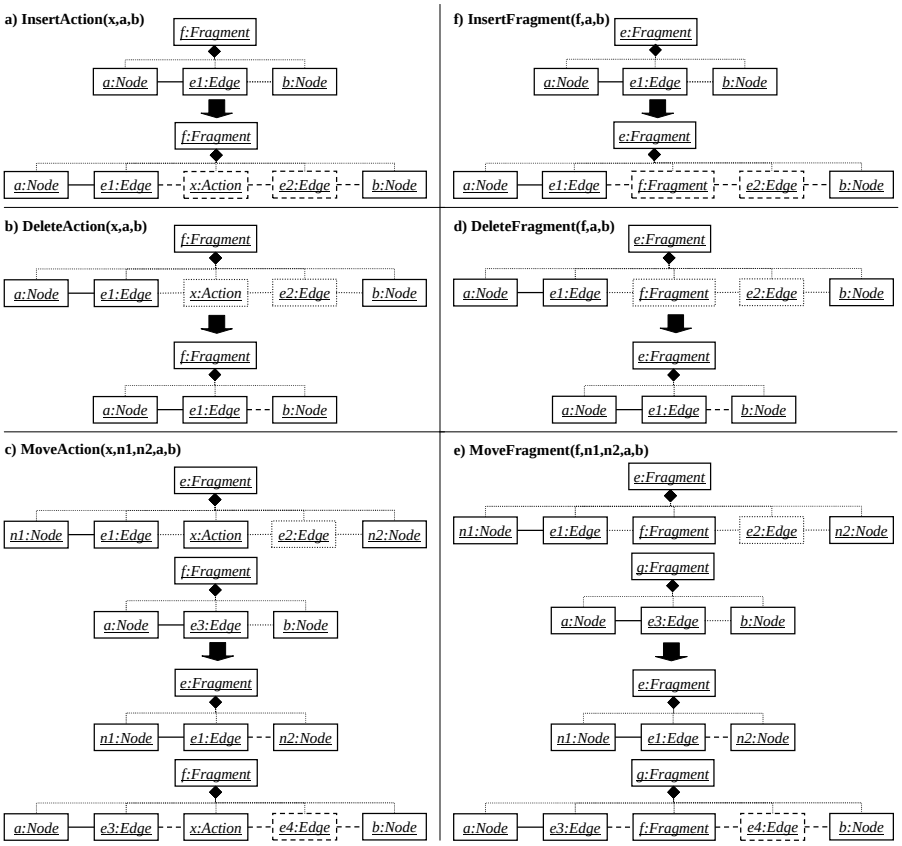

Fig. 5. Specification of operations dealing with Actions and Fragments

3.2 Dependencies and Conflicts of Changes

For graph transformation, dependencies and conflicts have been defined [5, 9, 18]: Formally, given a sequence of graph transformations $G \overset{p_1(o_1)}{\Longrightarrow} H_1 \overset{p_2(o_2)}{\Longrightarrow} X$, $H_1 \overset{p_2(o_2)}{\Longrightarrow} X$ is *(weakly sequential) independent* of $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ if the occurrence $o_2(L_2)$ is already present before the application of p_1. This is the case if $o_2(L_2)$ does not overlap with objects created by p_1. If in addition p_2 does not delete objects that are needed for the

application of p_1, then p_1 and p_2 can be exchanged and are called sequentially independent.

Formally, given two graph transformations $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ and $G \overset{p_2(o_2)}{\Longrightarrow} H_2$, $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ is *(weakly parallel) independent* of $G \overset{p_2(o_2)}{\Longrightarrow} H_2$ if the occurrence $o_1(L_1)$ of the left-hand side of p_1 is preserved by the application of p_2. This is the case if $o_1(L_1)$ does not overlap with objects that are deleted by p_2. If the two transformations are mutually independent, they can be applied in any order yielding the same result. In this case we speak of *parallel independence*. Otherwise, if one of two alternative transformations is not independent of the second, the second will disable the first. In this case, the two steps are *in conflict*. According to the Local Church Rosser Theorem [5] [3], parallel independence of two transformation steps induces their sequential independence and vice versa (with adapted occurrences).

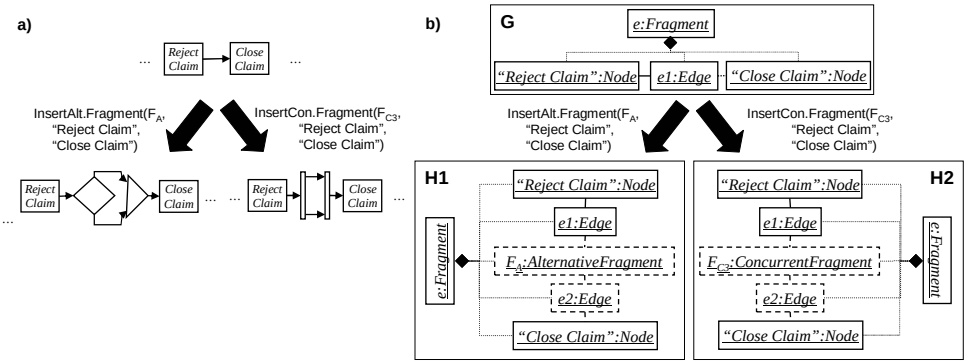

Fig. 6. A conflict between two changes

Often, we are not only interested to know whether two particular transformation steps are parallel or sequentially independent but also whether two transformation rules are parallel or sequentially independent. Related work (e.g. [9]) already discusses the notion of *potential* conflicts and dependencies. Given two rules p_1, p_2, a potential conflict or dependency occurs if there exist transformation steps such that a conflict or sequential dependency occurs. Given two rules p_1 and p_2, the computation of potential conflicts and dependencies can be done using critical pairs. A critical pair is a pair of transformation steps $H_1 \overset{p_1(o_1)}{\Longleftarrow} G \overset{p_2(o_2)}{\Longrightarrow} H_2$ which are in conflict and with the property that G is minimal.

Critical pairs of two rules p_1 and p_2 can be computed by overlapping the left hand sides of p_1 and p_2 in all possible ways such that there exists at least one object that is deleted by one of the rules and both rules are applicable. Figure 6 a) shows two conflicting changes in concrete syntax from our example and Figure 6 b) shows the critical pair for this situation. Here, both changes insert a fragment at the same position in G. If one of the changes is applied, the other one will not be applicable anymore.

[3] The Local Church Rosser Theorem has been proven for typed attributed graph transformation in [7].

In the following, we discuss the results of critical pair analysis obtained for the different scenarios. In the single-user scenario it is important to know which changes can be independently rejected/confirmed. This can be achieved by studying compound operations for sequential independence. The idea is here to determine when compound operations are sequentially independent based on the parameters they have. As an example, an *InsertFragment* operation followed by an *InsertAction* operation into the fragment leads to a dependency. This means that if the *InsertFragment* operation is rejected, also the (dependent) *InsertAction* operation needs to be rejected.

In order to compute the sequential dependencies between compound changes, given two rules p_1 and p_2, we compute critical pairs of p_1 and p_2^{-1} and p_1^{-1} and p_2 [9].

The critical pairs obtained are then encoded by specifying conditions on the parameters of the operations and captured in a dependency matrix, shown in Figure 19 for *InsertAction*, *DeleteAction*, *MoveAction* and *InsertFragment* in the Appendix A [4]. We assume here that $entry(F)$ and $exit(F)$ are used to denote the entry or exit nodes of a fragment which can be either a Decision or Fork or Merge or Join node. For each combination of operations, there exist configurations of the parameters such that the two operations are sequentially dependent. For example, $InsertAction(X1, A, B)$ and $InsertFragment(F2, C, D)$ are sequentially dependent if $C = X1 \vee D = X1$.

In the multi-user scenario, given two rules p_1 and p_2, we compute the critical pairs of p_1 and p_2 for all combinations of change operations. Critical pairs obtained are then encoded by specifying conditions on the parameters of p_1 and p_2, shown in a conflict matrix in Figure 20 for our compound changes *InsertAction*, *DeleteAction*, *MoveAction* and *InsertFragment*. Similarly to the sequential dependencies, all of the change operations are potentially in conflict.

4 Dependencies and Conflicts of Change Sequences

Until now we have studied dependencies and conflicts of change operations in isolation. We now extend our concept of dependencies and conflicts to change sequences in order to deal with change logs as introduced above.

4.1 Dependencies of Change Sequences

For the following discussion, we assume that a change sequence $\Delta = \langle t_1(o_1), .., t_n(o_n) \rangle$ consists of a sequence of transformation steps t_i at an occurrence o_i such that the transformation $G = S_0 \overset{t_1(o_1)}{\Longrightarrow} S_1..S_{n-1} \overset{t_n(o_n)}{\Longrightarrow} S_n = H$ exists. Informally, a change sequence Δ can be considered as a concatenation of model transformations and represents a change log as introduced before. As a shorthand, we also write $\Delta = \langle t_1, .., t_n \rangle$.

Given a change sequence $\Delta = \langle t_1, .., t_n \rangle$, we are interested in sequential dependencies because these are the changes that cannot be resolved in any order. All potential dependencies that can occur between two changes are shown in Figure 19. Based on

[4] We used the AGG tool [23] to partially compute and validate the entries of the matrices. However, AGG does currently not support inheritance in the type graph which required a simplification of rules.

this, a given change sequence $\Delta = \langle t_1, .., t_n \rangle$ can be broken up into subsequences c_i such that the following holds:

- each subsequence c_i consists of a sequence of change operations $t_i \in \Delta$, i.e. $c_k = \langle t_l, .., t_r \rangle$ with the property that t_i is not dependent of any change operation not contained in c_k, and
- for two subsequences c_k and c_l, the change operations contained are disjoint.

These subsequences can be computed as follows: Given a Δ, we compute for each pair of compound changes t_i and t_j sequential dependencies. Thereby we check whether operations t_i and t_j with their concrete parameters form a critical pair according to the dependency matrix shown in Figure 19. If t_i and t_j are dependent, they belong to the same subsequence.

The dependency matrix can only indicate a sequential dependency between two operations whose signatures overlap. There are cases where a sequential dependency exists and signatures do not overlap. These dependencies will be detected in a transitive way. For instance, the sequential dependency of *InsertAction*(*"Calc. Loss Amount"*, $Fork^1_{FC5}$, $Join^1_{FC5}$) on *InsertConcurrentFragment*(F_{C4}, *"Settle Claim"*, $Merge^1$) (fragment F_{C4}) will be detected transitively since the insertion of the action *"Calc. Loss Amount"* is dependent on *InsertConcurrentFragment*(F_{C5}, $Fork^2_{FC4}$, $Join^2_{FC4}$) (fragment F_{C5}) which is in turn dependent on the insertion of fragment F_{C4}.

In the end, each t_i belongs to exactly one subsequence and the operations in different subsequences are sequentially independent. According to the Local Church Rosser Theorem this induces parallel independence for operations in disjoint subsequences as well.

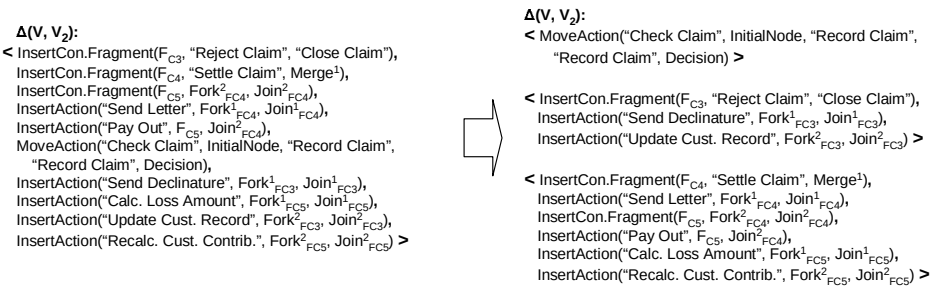

Fig. 7. Independent subsequences in $\Delta(V, V_2)$

Figure 7 shows the sequence of changes applied on our example model V in order to obtain V_2 and the decomposition of these changes into parallel independent subsequences. The parallel independent subsequences for change sequences are important for several reasons: firstly, they show which changes are dependent which is important for the single-user scenario. Secondly, for the multi-user scenario, the parallel independent subsequences will be used for computing conflicts.

4.2 Conflicts of Change Sequences

Given two change sequences $\Delta_1 = \langle t_1, .., t_n \rangle$ and $\Delta_2 = \langle s_1, .., s_m \rangle$, we first compute the parallel independent subsequences of each change sequence as described previously. Given two subsequences $c_k = \langle t_i, ..t_j \rangle \in \Delta_1$ and $d_l = \langle s_m, .., s_n \rangle \in \Delta_2$ we are then interested in conflicts because these must be taken into account when rejecting or accepting changes in the multi-user scenario.

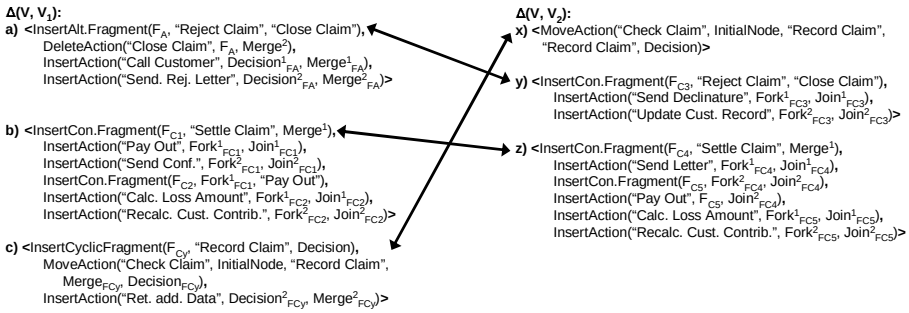

Fig. 8. Computation of conflicts between subsequences of change sequences

Conflicts can be computed based on the results of critical pair analysis which determines potential conflicts, displayed in Figure 20. For computation of conflicts, the operations in two change sequences are analyzed pairwise for conflicts. Figure 8 shows the result of this computation for our example. Here, three conflicts occur, indicated by the arrows. Once conflicts have been determined, conflicts need to be resolved. For this, different options exist that will be discussed in the next section.

After resolving a conflict between c_k and d_l, new conflicts between Δ_1 and Δ_2 can occur. For identifying these, we recompute conflicts after each conflict resolution. Optimizations of this procedure where recomputation of conflicts is restricted to certain subsequences is left for future work. Resolving a conflict can also lead to less conflicts if an operation together with its subsequence is rejected and its dependent operations also become non-applicable. In the following section, we will elaborate on conflict resolution.

5 Conflict Resolution

In this section, we discuss the different options for conflict resolution. For the following discussion, we assume that two change sequences Δ_1 and Δ_2 exist that have been divided into parallel independent subsequences as previously explained. For a given conflict, conflict resolution can consist of (at least) the following choices:

- *selection of the subsequence to adopt*, meaning that the complete other subsequence is discarded and not considered further,
- performing a *combination of the two operations or unifying the two operations*. The operations in conflict have a similar type or are structurally very similar. In such a

case, the conflict can be resolved by performing one operation and establishing a mapping between the elements used. If the operations cannot be unified directly, i.e. one operation inserts a fragment with six branches, the other one with only two branches, then a common superset or subset can be chosen.

– *both operations are performed* by modifying one or both operations, leading e.g. to a sequential or parallel insertion of fragments or actions.

The choice which type of conflict resolution to adopt is made by the user, usually based on his or her domain knowledge of the models, and cannot be automated.

In many cases, the decision about conflict resolution influences the change operations that are dependent on the conflicting operations. In the case of combination using unification, the parameters of the dependent operations have to be recomputed by replacing the unified parameters of the conflicting operations. In the case of a combination by introducing a new operation, this also yields to recomputation of parameters.

In the case that one of the two subsequences is adopted and the other one is discarded, it is important to know about possible conflicts that occur within the adopted subsequence. By adoption, all the operations inside the subsequence will also be adopted, meaning that in case of a conflict this type of conflict resolution will be chosen for contained operations as well.

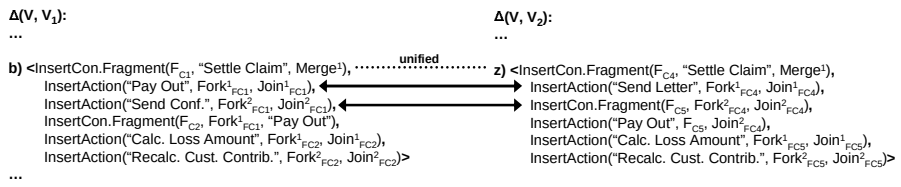

Fig. 9. Unification of two conflicts

In our example, a conflict which is likely to be resolved by a unification is the conflict between subsequence b) and z) shown in Figure 9. After the unification of *InsertCon.Fragment*$(F_{C1}, ...)$ and *InsertCon.Fragment*$(F_{C4}, ...)$ the parameters and conflicts for the changes that are dependent on the unified changes are recomputed. In this case, F_{C1} and F_{C4} are unified as well as the nodes $Fork_{FC1}$ and $Fork_{FC4}$ and $Join_{FC1}$ and $Join_{FC4}$. This leads to two additional conflicts between *InsertAction*$(Pay\ Out, ...)$ and *InsertAction*$(Send\ Letter, ...)$ as well as *InsertAction*$(Send\ Conf., ...)$ and *InsertCon.Fragment*$(F_{C5}, ...)$, because due to the unification the dependent changes are now applied in the same concurrent fragment and their parameters overlap.

In all cases of conflict resolution, conflict resolution entails the application of one or both conflicting, possibly modified or adapted, change operations. After this, conflicts between the following operations are recomputed and displayed to the modeler, leading to an iterative resolution process.

Figure 10 illustrates one possible resulting process model V_{merged} based on the modifications made in V_1 and V_2. In order to visualize the conflict resolution process, applied compound changes are printed in bold letters and rejected changes in italic letters. We start the conflict resolution by unifying the conflict between subsequence a)

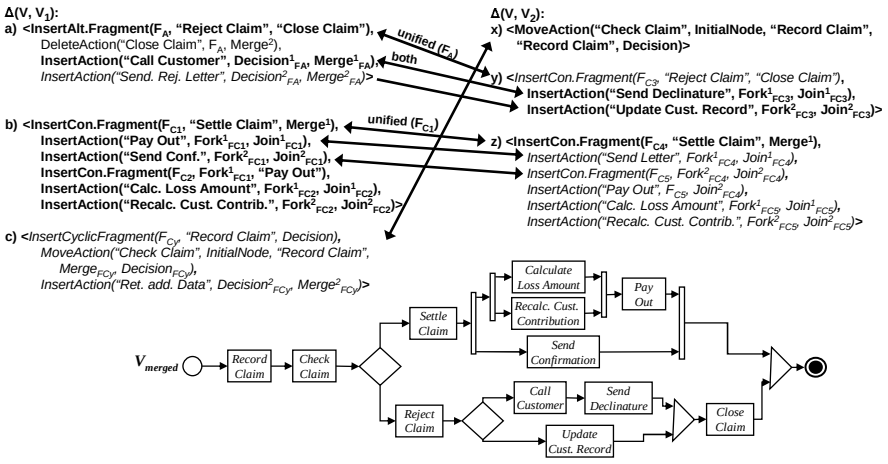

Fig. 10. A possible merged process model based on V and the modifications made in V_1 and V_2

and y) and applied $\mathit{InsertAlt.Fragment}(F_A, ...)$ on model V. By the unification, all occurrences of Fork^*_{FC3} and Join^*_{FC3} in the signatures of the remaining operations are substituted by $\mathit{Decision}^*_{FA}$ and Merge^*_{FA}. Thereby, two new conflicts between the *InsertAction* operations in subsequence a) and y) arise. In case of the conflict between $\mathit{InsertAction}(\mathit{"Call\ Customer"}, ...)$ and $\mathit{InsertAction}(\mathit{"Send\ Declinature"}, ...)$, we select both operations for application, leading to a sequential insertion of the two actions. In the other case, we apply $\mathit{InsertAction}(\mathit{"Update\ Cust.\ Record"}, ...)$ and reject $\mathit{InsertAction}(\mathit{"Send\ Rej.\ Letter"}, ...)$. Finally, we apply $\mathit{DeleteAction}(\mathit{"Close\ Claim"}, ...)$. Further, we resolve the conflict between b) and z) by unification as described previously and then apply only operations in b). For the resolution of the conflict between subsequence c) and x), we decide to adopt only subsequence x) and rejected all operations contained in subsequence c). This example shows that using our approach it is possible to resolve conflicts between change sequences in an iterative way with minimal manual intervention such that a consolidated process model is constructed.

6 Tool Support and Evaluation

In this section, we report on tool support and evaluation of our approach. The dependency and conflict detection approach has been implemented as a prototype for IBM WebSphere Business Modeler. Figure 11 shows a screenshot of the extension with the example and computed conflicts.

As an initial evaluation of our approach we provide in the remainder of this section a case study comparing our approach using compound change operations to an approach relying on elementary change operations. In particular, we concentrate on the effects on dependency and conflict computation and on a comparison of required user intervention for merging two process models. For that purpose, we first describe the setting of our case study and introduce a set of elementary change operations together with configurations of the operations that lead to sequential dependencies and conflicts. Then we use

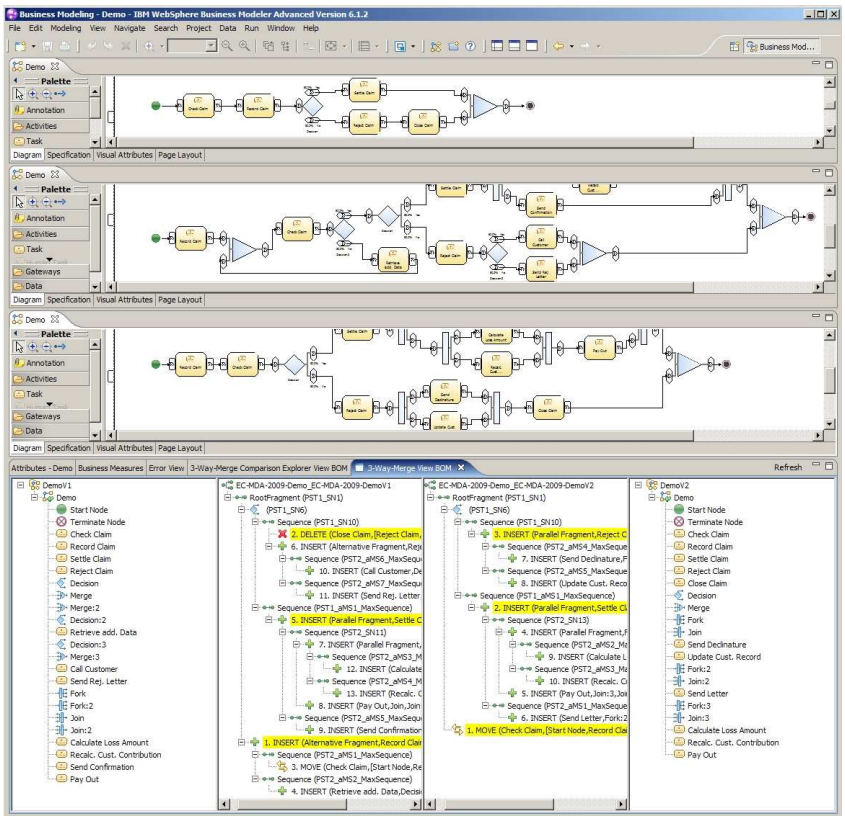

Fig. 11. 3-Way Merge View in the IBM WebSphere Business Modeler

the elementary change operations to express the differences of our example. Finally, we compare the required user intervention to resolve selected differences and conflicts of the example using compound changes and elementary changes.

6.1 Setting of the Case Study

For our case study we assume that the set of elementary change operations consists of the following four operations: $\mathit{InsertElement}(A)$, $\mathit{DeleteElement}(B)$, $\mathit{InsertLink}(L,A,B)$, and $\mathit{DeleteLink}(L,A,B)$. Analogously to our set of compound operations, we have formulated the elementary change operations in terms of graph transformations and have computed sequential dependencies and conflicts.

Figure 21 in the Appendix B shows all configurations of elementary operations that lead to sequential dependencies. For instance, there is a sequential dependency between $\mathit{InsertElement}(X1)$ and $\mathit{InsertLink}(L1,C,D)$ if either $C = X1$ or $D = X1$ with the result, that $X1$ needs to be inserted before the insertion of the Link $L1$ becomes applicable.

Conflicts between elementary change operations, which can occur in multi-user scenarios, are shown in Figure 22 in the Appendix B. We computed critical pairs be-

tween two rule sets of elementary change operations and encoded conflicting situations in terms of change operation parameters, e.g. two independently applied operations *DeleteElement*$(X1)$ and *DeleteElement*$(X2)$ are in conflict if $X2 = X1$. The entries of the matrices shown in Figure 21 and Figure 22 were computed and validated using the AGG tool [23].

Using the elementary change operations to express the differences between the process models V, V_1 and V, V_2 of our example (Figure 1) results in the change logs given in Figure 12. For convenience and readability, both change logs $\Delta(V, V1)$ and $\Delta(V, V2)$ are ordered in such a way that change sequences containing dependent change operations are clearly visible (separated by black lines).

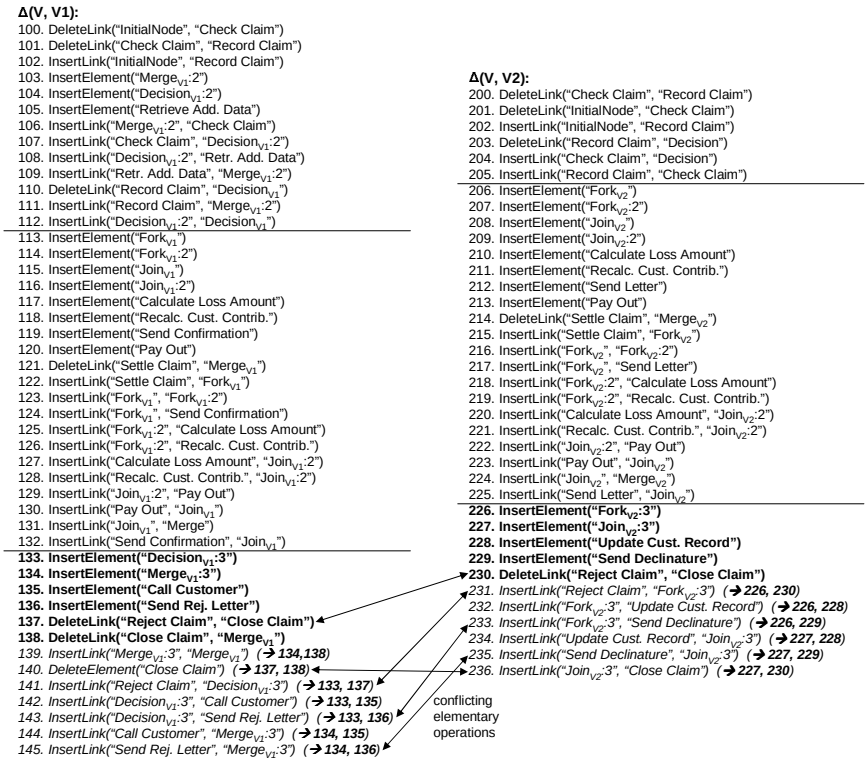

Fig. 12. Elementary change operations for the example with dependencies and conflicts for selected operations

In case of our example, semantically related elementary changes (e.g. the insertion of a fork and the insertion of a join forming a parallel fragment) are within the same change sequence because in our example the related changes are either directly or transitively dependent on each other. This is not always the case as it can happen that semantically related changes are neither directly nor transitively dependent. In practice this makes the resolution of differences even more difficult because related changes are then located in different change sequences.

Dependencies and conflicts for selected changes are visualized in Figure 12. Dependencies are indicated directly behind the elementary operations, e.g. the application of operation 142 requires that operations 133 and 135 were applied earlier. Conflicts are represented by black arrows connecting operations in the change logs. For instance, the two *DeleteLink* operations 137 and 230 are in conflict, because they delete the same link $(L2 = L1)$.

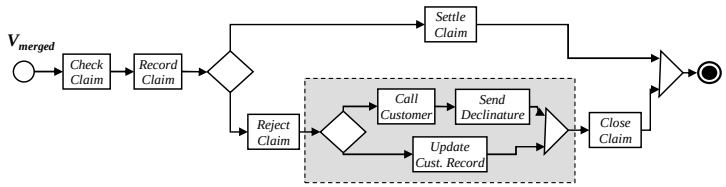

Fig. 13. Merged Process Model for the Evaluation Case Study

Figure 13 visualizes a merged process model V_{merged} that can be obtained by applying a combination of operations contained in the change logs $\Delta(V, V1)$ and $\Delta(V, V2)$. Changes to the original source process model *V* (introduced in Figure 1) are highlighted by the gray rectangle.

In the following two subsections we describe two alternative resolutions, first by compound and then by elementary change operations of the change logs in order to obtain the merged process model V_{merged}. Along the resoluions, we measure the required user intervention in terms of work units for the inspection of conflicts and the application of change operations. For simplicity, we assume that each inspection and application of an operation, dependency or conflict requires one work unit.

6.2 Difference and Conflict Resolution using Compound Change Operations

The compound change operations which need to be applied to the source process model *V* in order to create the merged process model V_{merged} are represented in Figure 15. In *Step 1* the conflict between the insertions of the two fragments F_A and F_{C3} is inspected (one work unit). We assume that a user decides to resolve the conflict by a unification of the compound changes and inserts the fragment F_A (done in *Step 2*, one work unit). Thereby, the position parameters of both *InsertAction* operations in change sequence **y** are adapted to reflect entry (*Decision*) and exit (*Merge*) of fragment F_A. Further, the conflict between the insertions of *Call Customer* and

Case Study: Difference and Conflict Resolution using Compound Change Operations		Work Units
Step 1	Inspect conflict between the insertions of the fragments F_A and F_{C3}	1
Step 2	Resolve conflict through unification	1
Step 2	Inspect conflict between “Call Customer” and “Send Declinature”	1
Step 3	Resolve conflict by applying both operations	2
Step 3	Inspect conflict between “Send Rej. Letter” and “Upd. Cust. Record”	1
Step 4	Resolve conflict by applying only right hand side	1
		7

Fig. 14. Required user intervention using compound change operations

Send Declinature is inspected and it is decided to apply both operations in *Step 3*. Inspection and the application of both operations increases the work unit counter by three.

Finally, the remaining conflict between the insertions of *Send Rej. Letter* and *Update Cust. Record* is inspected (one work unit) and it is resolved by applying only the right hand operation (one work unit). Using the compound operations and the conflicts guided the user through the resolution process and in total seven work units of user intervention were necessary to obtain the merged process model V_{merged}.

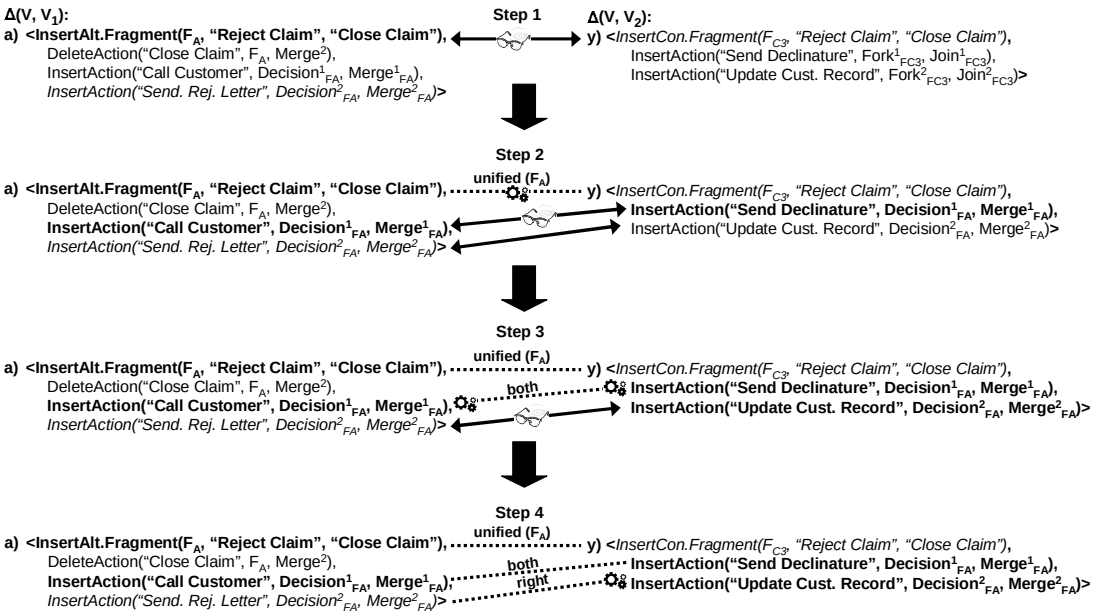

Fig. 15. Work units to apply selected changes using compound operations

6.3 Difference and Conflict Resolution using Elementary Change Operations

In contrast to compound change operation, a conflict-driven approach for the resolution of differences using elementary change operations is not applicable since most of the conflicting elementary operations require other operations to be applied before. As a consequence, we suggest an element-driven approach for elementary change operations that starts with the decision which elements shall be inserted or deleted. According to our set of elementary change operations, we call this approach *Insert−/DeleteElement*-driven.

Figure 17 visualizes the parts of the elementary change logs that need to be considered in order to create the merged process model V_{merged}. According to the $Insert-/DeleteElement$-driven strategy, we start with an inspection of all *InsertElement* and *DeleteElement* operations in *Step 1* (resulting in nine work units). Then, in *Step 2* we insert the five required elements (five work units). Afterwards, the remaining link operations need to be inspected in order to identify the operations that are applicable with respect to the recently inserted and deleted elements. Since this step is straight-forward computation, we assume that applicable link operations are determined automatically, which produces no costs.

Case Study: Difference and Conflict Resolution using Elementary Change Operations, Insert-/DeleteElement-Driven		Work Units
Step 1	Inspect all Insert- and DeleteElement operations	9
Step 2	Apply five InsertElement operations (Decision, Merge, Call Customer, Send Decl., Update Cust. Record)	5
	Inspect Insert/DeleteLink operations (link operations that turn non-applicable due to applied element operations can be computed easily – inspections are not counted)	(15)
Step 3	Apply working link Operations 137, 141, 142	3
Step 4	Manually connect the process model with five links	5
		22

Fig. 16. Required user intervention using elementary change operations

In *Step 3* the applicable link operations are applied. In our example, these are the operations 137, 141, and 142, increasing the work unit counter by three. Finally, the process model needs to be connected manually in *Step 4*, since not all elements are connected so far: A user has to insert five edges by hand in order to complete the alternative structure in the merged process model V_{merged}.

6.4 Evaluation Results

One goal of our evaluation was to show that our approach leads to less conflicts and dependencies than an approach relying on elementary change operations, illustrated in Table 18 for our example (Figure 1). Another goal was to show that our approach then also leads to less required user intervention than an approach based on elementary operations.

We can distinguish between application of operations, conflict examination and conflict resolution. On average, the number of elementary operations is three times the number of compound operations which makes the resolution more complex. For application of operations this means that the user intervention triples (unless further optimizations are implemented for the elementary operations).

The relation of conflicts for the elementary and compound operations cannot be estimated. In our example, we obtain the number of conflicts as indicated in the table in Figure 18, leading to a higher number of required conflict examinations if elementary change operations are used.

The user intervention required for conflict resolution depends on the support given by the modeling tool. Our example illustrates that computed dependencies and conflicts between elementary change operations do not really help a user to merge different versions of a process model: Most of the conflicts are between operations that modify links, and these operations require other operations to be applied before. Our sample resolution shows that the user intervention required (work units in Figure 18) when

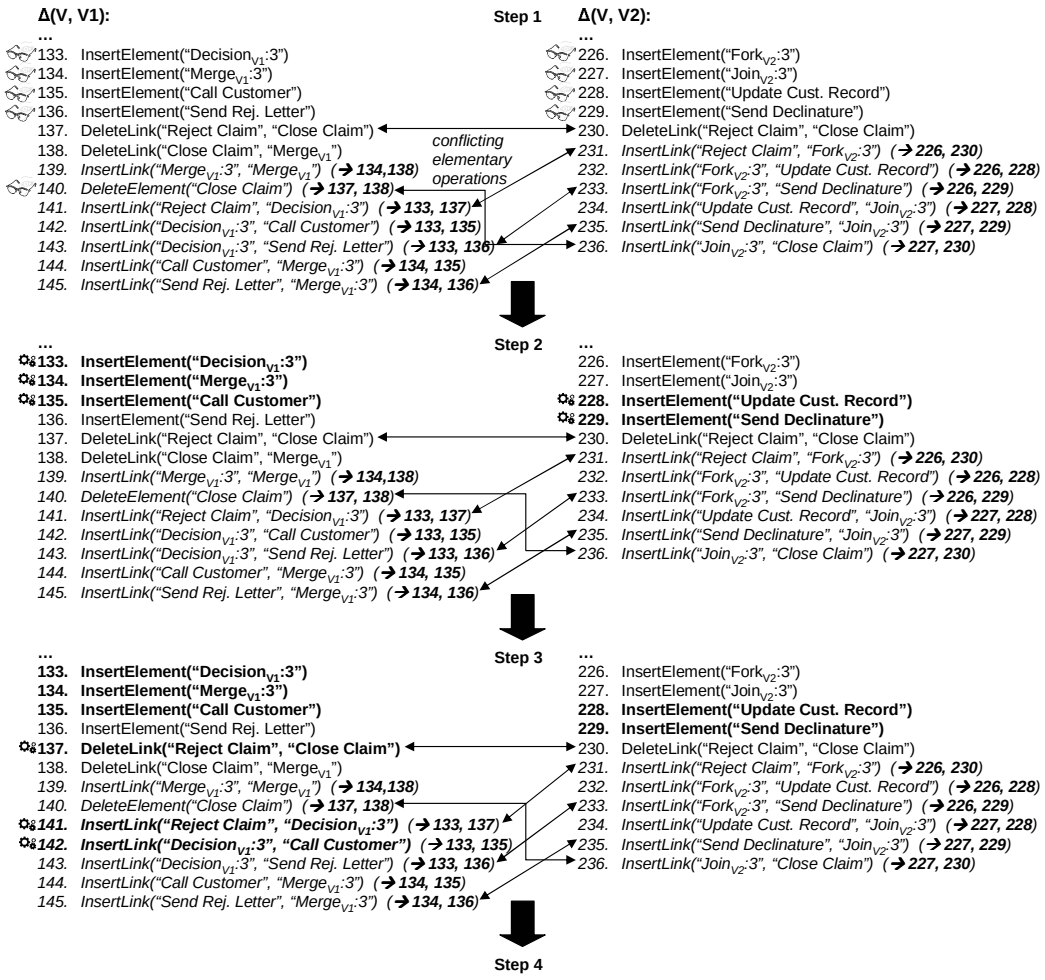

Fig. 17. Work units to apply selected changes using elementary operations

compound change operations are used is much less compared to the situation when using elementary change operations.

In addition, our case study also shows that compound operations can be used to realize advanced functionality such as change operation unification which is difficult to realize for elementary operations, unless they are grouped again to compound operations. Compound operations also enable to always create a connected and well-formed model during conflict resolution whereas using elementary operations elements often have to be reconnected manually.

	Elementary Changes		Compound Changes	
	Δ(V, V1)	Δ(V, V2)	Δ(V, V1)	Δ(V, V2)
# of Change Operations	42	33	13	10
# of Dependencies	45	36	10	7
# of Conflicts	23		3	
# of Work Units for Sample Resolution	21		7	

Fig. 18. Evaluation results for approaches based on elementary and compound operations

7 Related Work

Mens et al. [18] analyze refactorings for structural conflicts using critical pair analysis. They first express refactorings as graph transformations and then detect conflicts using the AGG tool [23]. Hausmann et al. [9] analyze functional requirements in a use-case driven software development approach for conflicts and dependencies. Further approaches including critical pair analysis include work by Mens et al. [17] for transformation dependency analysis. All of these approaches are similar to ours with regards to the analysis of syntactic conflicts and dependencies and the formalization using graph transformation. However, there are also differences: Firstly, we analyze process model refactorings and elaborate on change sequences which have not been analyzed for conflicts before. Further, our analysis is performed after the changes have been made for resolving conflicts whereas in their work conflicts should be avoided up front.

Another area of related work is concerned with model composition and model versioning. Alanen and Porres [2] describe an algorithm how to compute elementary change operations in a similar setting as ours. Kolovos et al. [12] describe the Epsilon merging language which can be used to specify how models should be merged. Kelter et al. [11] present a generic model differencing algorithm. All these approaches aim at providing generic support for merging different models but do not focus on dependencies and conflicts of change operations. Both in the IBM Rational Software Architect [15] as well as in the software configuration management infrastructure for UML models Odyssey-SCM [19], conflicts between versions are computed based on elementary changes. In contrast to these approaches, we focus on identifying dependent and conflicting compound change operations based on critical pairs and provide a selection of conflict resolution techniques which is language-specific to process models, showing that there is a need for these domain-specific approach to dependency and conflict detection. As such, our approach can be categorized as an operation-based, tree-based and syntactic approach to software merging [16].

Cicchetti et al. [4] have recently proposed a metamodel for representing conflicts which can be used for specifying both syntactic as well as semantic conflicts. One key difference to our work is that we do not specify conflicts for compound operations but we compute them by using the critical pair approach. Finally, within the process modeling community, Dijkman [6] has categorized differences of process models in the context of process integration where models do not originate from a common source model. Rinderle et al. [22] have studied disjoint and overlapping process model changes in the context of the problem of migrating process instances. In their formal framework, two change sequences are commutative if their application leads to trace equivalent

process models. If they in addition do not affect overlapping node sets, then they are called disjoint. Based on this formal framework, they study conditions under which process instances can be migrated to a new process model. Our work can be considered as complementary to their work because we provide a structural approach to detecting disjoint and overlapping changes.

8 Conclusion and Future Work

When modeling in a distributed environment, changes performed on models can be conflicting or sequentially dependent. In order to consolidate different models, conflicting changes must be computed and manually resolved. In this paper, we have shown how change operations can be analyzed for conflicts and dependencies. Based on this, we presented an approach for breaking up a sequence of change operations into subsequences such that change operations from different subsequences are independent. Our approach allows to make dependencies explicit and resolve conflicts in a versioning scenario with special language-specific conflict resolution choices. Our evaluation has shown that our approach leads to less user interaction than using elementary change operations.

There are several directions for future work: Firstly, we would like to validate our approach also for other behavioral models such as statecharts where compound change operations need to be designed and then analyzed for conflicts and dependencies in a similar way. Another area of future work is to take into account the semantics of process models in order to be able to identify those syntactic conflicts which do not represent a semantic conflict.

References

1. IBM WebSphere Business Modeler. http://www.ibm.com/software/integration/wbimodeler/.
2. M. Alanen and I. Porres. Difference and Union of Models. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003*, volume 2863 of *LNCS*, pages 2–17. Springer, 2003.
3. P. Bottoni, A. Schürr, and G. Taentzer. Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In *VL 2000*, pages 59–60. IEEE Computer Society, September 2000.
4. A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2008.
5. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1997.
6. R. Dijkman. A Classification of Differences between Similar Business Processes. In *EDOC'07*, pages 37–50. IEEE Computer Society, 2007.
7. H. Ehrig, U. Prange, and G. Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *ICGT*, volume 3256 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2004.
8. R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In L. C. Briand and A. L. Wolf, editors, *International Conference on Software*

Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA, pages 37–54, 2007.
9. J. H. Hausmann, R. Heckel, and G. Taentzer. Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In *Proceedings ICSE'02*, pages 105–115. ACM, 2002.
10. R. Heckel, J. M. Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *ICGT'02*, volume 2505 of *LNCS*, pages 161–176. Springer-Verlag, 2002.
11. U. Kelter, J. Wehren, and J. Niere. A Generic Difference Algorithm for UML Models. In P. Liggesmeyer, K. Pohl, and M. Goedicke, editors, *Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005 in Essen*, volume 64 of *LNI*, pages 105–116. GI, 2005.
12. D. S. Kolovos, R. Paige, and F. Polack. Merging Models with the Epsilon Merging Language (EML). In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS 2006*, volume 4199 of *LNCS*, pages 215–229. Springer, 2006.
13. J. M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 5(3):233–259, 2006.
14. J. M. Küster, C. Gerth, A. Förster, and G. Engels. Detecting and Resolving Process Model Differences in the Absence of a Change Log. In M. Dumas and M. Reichert, editors, *BPM'08*, volume 5240 of *LNCS*, pages 244–260. Springer-Verlag, 2008.
15. K. Letkeman. Comparing and merging UML models in IBM Rational Software Architect : Part 3. A deeper understanding of model merging. *IBM Developerworks*, 2005. http://www.ibm.com/developerworks/rational/library/05/802_comp3/.
16. T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Trans. Software Eng.*, 28(5):449–462, 2002.
17. T. Mens, R. Van Der Straeten, and M. D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS 2006*, volume 4199 of *LNCS*, pages 200–214. Springer, 2006.
18. T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 6(3):269–285, 2007.
19. L. Murta, H. Oliveira, C. Dantas, L. Lopes, and C. Werner. Odyssey-SCM: An integrated software configuration management infrastructure for UML models. *Science of Computer Programming*, 65(3):249–274, 2007.
20. Object Management Group (OMG). *The Unified Modeling Language 2.0*, 2005.
21. S. Rinderle, M. Jurisch, and M. Reichert. On Deriving Net Change Information From Change Logs - The DELTALAYER-Algorithm. In A. Kemper et al., editor, *BTW'07*, volume 103 of *LNI*, pages 364–381. GI, 2007.
22. S. Rinderle, M. Reichert, and P. Dadam. Disjoint and Overlapping Process Changes: Challenges, Solutions, Applications. In R. Meersman and Z. Tari, editors, *CoopIS'04*, volume 3290 of *LNCS*, pages 101–120. Springer, 2004.
23. G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *AGTIVE*, volume 3062 of *LNCS*, pages 446–453, 2003.
24. J. Vanhatalo, H. Völzer, and F. Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In *ICSOC 2007*, volume 4749 of *LNCS*, pages 43–55. Springer, 2007.
25. B. Weber, S. Rinderle, and M. Reichert. Change Patterns and Change Support Features in Process-Aware Information Systems. In J. Krogstie, A. L. Opdahl, and G. Sindre, editors, *CAiSE'07*, volume 4495 of *LNCS*, pages 574–588. Springer, 2007.

A Dependency and Conflict Matrix for Compound Operations

Figure 19 shows configurations of compound operations that lead to sequential dependencies. Analogously, configurations of compound operations that lead to conflicts are given in Figure 20. We assume here that $entry(F)$ and $exit(F)$ are used to denote the entry or exit nodes of a fragment which can be either a Decision or Fork or Merge or Join node. The entries of the matrices in Figure 19 and Figure 20 are computed and validated using the critical pair analysis of the AGG tool [23].

	InsertAction (X2,C,D)	**DeleteAction (X2,C,D)**	**MoveAction (X2,oQ,oT,nQ,nT)**	**InsertFragment (F2,C,D)**
InsertAction (X1,A,B)	[IA(X1), IA(X2)]: C = X1 v D = X1	[IA(X1), DA(X2)]: X2 = X1 v (D = X1 & X2 = A) v (C = X1 & X2 = B)	[IA(X1), MA(X2)]: (nQ = A & nT = X1) v (nQ = X1 & nT = B) v (X2 = A & oT = B) v (oQ = X1 & X2 = B) v (oQ=A & X2=X1 & oT=B)	[IA(X1), IF(F2)]: C = X1 v D = X1
DeleteAction (X1,A,B)	[DA(X1), IA(X2)]: (C = A & D = B)	[DA(X1), DA(X2)]: (C = A & X2 = B) v (X2 = A & D = B)	[DA(X1), MA(X2)]: (nQ = A & nT = B) v (oQ = A & X2 = B) v (X2 = A & oT = B)	[DA(X1), IF(F2)]: (C = A & D = B)
MoveAction (X1,oP,oS, nP,nS)	[MA(X1), IA(X2)]: (C = oP & D = oS) v (C = X1 & D = nS) v (C = nP & D = X1)	[MA(X1), DA(X2)]: (C = oP & X2 = oS) v (X2 = oP & D = oS) v (D = X1 & X2 = nP) v (C = X1 & X2 = nS) v (C = nP & X2 = X1 & D = nS)	[MA(X1), MA(X2)]: (nQ = oP & nT = oS) v (oQ = oP & X2 = oS) v (X2 = oP & oT = oS) v (X2 = X1 & oT = nS) v (oQ = X1 & X2 = nS) v (nQ = X1 & nT = nS) v (X2 = nP & oT = X1) v (nQ = nP & nT = X1) v (oQ = nP & X2 = X1)	[MA(X1), IF(F2)]: (C = oP & D = oS) v (C = X1 & D = nS) v (C = nP & D = X1)
Insert Fragment (F1,A,B)	[IF(F1), IA(X2)]: C = F1 v D = F1 v C = entry(F1) v D = exit(F1)	[IF(F1), DA(X2)]: (X2 = A & D = F1) v (C = F1 & X2 = B) v C = entry(F1) v D = exit(F1)	[IF(F1), MA(X1)]: (nQ = A & nT = F1) v (nQ = F1 & nT = B) v (X2 = A & oT = B) v (oQ = F1 & X2 = B) v oQ=entry(F1) v nQ=entry(F1) v oT=exit(F1) v nT=exit(F1)	[IF(F1), IF(F2)]: C = F1 v D = F1 v C = entry(F1) v D = exit(F1)

Fig. 19. Sequential dependencies of compound change operations

	InsertAction (X2,C,D)	**DeleteAction (X2,C,D)**	**MoveAction (X2,oQ,oT,nQ,nT)**	**InsertFragment (F2,C,D)**
InsertAction (X1,A,B)	(C = A & D = B)	(C = A & X2 = B) v (X2 = A & D = B)	(nQ = A & nT = B) v (X2 = A & oT = B) v (oQ = A & X2 = B)	(C = A & D = B)
DeleteAction (X1,A,B)	(C = A & D = X1) v (C = X1 & D = B)	(X2 = A & D = X1) v (C = X1 & X2 = D) v (C = A & X2 = X1 & D = B) v	(X2 = A & oT = X1) v (oQ = X1 & X2 = B) v (nQ = A & nT = X1) v (nQ = X1 & nT = B) v (oQ = A & X2 = X1 & oT = B)	(C = A & D = X1) v (C = X1 & D = B)
MoveAction (X1,oP,oS, nP,nS)	(C = nP & D = oP) v (C = X1 & D = oS) v (C = oP & D = X1)	(X2 = oP & D = X1) v (C = X1 & X2 = oS) v (C = nP & X2 = nS) v (X2 = nP & D = nS) v (C = oP & X2 = X1 & D = oS)	(nQ=nP & nT=nS) v (nQ=X1 & nT=oS) v (nQ=oP & nT=X1) v (oQ=nP & X2=nS) v (oQ=X1 & X2=oS) v (oQ=oP & X2=X1) v (X2=nP & oT=nS) v (X2=X1 & oT=oS) v (X2=oP & oT=X1)	(C = nP & D = oP) v (C = X1 & D = oS) v (C = oP & D = X1)
Insert Fragment (F1,A,B)	(C = A & D = B)	(C = A & X2 = B) v (X2 = A & D = B)	(nQ = A & nT = B) v (X2 = A & oT = B) v (oQ = A & X2 = B)	(C = A & D = B)

Fig. 20. Conflicts between compound change operations

B Dependency and Conflict Matrix for Elementary Operations

	InsertElement (X2)	**DeleteElement (X2)**	**InsertLink (L2,C,D)**	**DeleteLink (L2,C,D)**
Insert Element(X1)	[IE(X1), IE(X2)]:	[IE(X1), DE(X2)]:	[IE(X1), IL(L2)]: C = X1 v D = X1	[IE(X1), DL(L2)]:
Delete Element(X1)	[DE(X1), IE(X2)]:	[DE(X1), DE(X2)]:	[DE(X1), IL(L2)]:	[DE(X1), DL(L2)]:
InsertLink (L1,A,B)	[IL(L1), IE(X2)]:	[IL(L1), DE(X2)]:	[IL(L1), IL(L2)]:	[IL(L1), DL(L2)]:
DeleteLink (L1,A,B)	[DL(L1), IE(X2)]:	[DL(L1), DE(X2)]: X2 = A v X2 = B	[DL(L1), IL(L2)]: C = A v D = B	[DL(L1), DL(L2)]:

Fig. 21. Sequential dependencies of elementary change operations

	InsertElement (X2)	**DeleteElement (X2)**	**InsertLink (L2,C,D)**	**DeleteLink (L2,C,D)**
Insert Element(X1)				
Delete Element(X1)		X2 = X1	C = X1 v D = X1	
InsertLink (L1,A,B)		X2 = A v X2 = B	(C = A && D != B) v (C != A && D = B)	
DeleteLink (L1,A,B)				L2 = L1 & C = A & D = B

Fig. 22. Conflicts between elementary change operations