RZ 3763 (# 99773) 01/27/2010
Computer Science 18 pages

Research Report

Dynamic Computation of Change Operations in Version Management of Business Process Models

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

[1]IBM Research – Zurich
8803 Rüschlikon
Switzerland

[2]Department of Computer Science
University of Paderborn
Germany

Emails: jku@zurich.ibm.com, {gerth,engels}@upb.de

Dynamic Computation of Change Operations in Version Management of Business Process Models

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

[1] IBM Research - Zurich, 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 business process models requires that changes can be resolved by applying change operations. In order to avoid user intervention and enable the user to follow an arbitrary order when resolving changes, position parameters of change operations need to be computed dynamically. In such an approach, change operations with computed position parameters must be applicable on the model and dependencies of change operations must be taken into account because otherwise invalid models can be constructed. In this paper, we study the concept of partially specified change operations where parameters are computed dynamically. We provide a formalization for partially specified change operations using graph transformation and provide a concept for their applicability. Based on this, we study potential dependencies and conflicts of change operations and show how these can be taken into account within change resolution.

1 Introduction

Version management of models typically comprises change detection as well as change resolution. Change detection produces a list of change operations which can then be inspected by the user. Within change resolution, the user makes decisions which change operations should be applied in order to produce a consolidated model. Existing approaches to version management of models allow the computation of change operations (e.g. by using technology such as EMF Compare [6]) and provide a set of techniques for model matching under different circumstances (see e.g. [1, 10]).

Version management of process models poses specific requirements on change operations: Compound change operations [14, 23] are used which always produce a connected process model and abstract from individual edge changes. Position parameter of change operations specify the place where a change is applied, i.e. direct predecessor and successor of the element that is changed. Iterative application of change operations requires a concept of change operations where position parameters are dynamically computed [14] in order to give the user maximal flexibility in the selection of change operations to apply.

If position parameters of change operations are dynamically computed then it has to be ensured that the change operations obtained are applicable on the model and produce

C. Gerth is funded by the International Graduate School of Dynamic Intelligent Systems at the University of Paderborn

again a connected process model. In addition, potential dependencies and conflicts of change operations must be taken into account. Otherwise it can happen that a user applies a change operation which cannot be properly applied, leading to a potentially unconnected model and problems when applying following change operations.

Existing approaches to dependency and conflict computation of change operations rely on the computation of a dependency and conflict matrix which allows to determine whether two operations are dependent [16, 13]. These approaches require that change operations are fully specified and cannot be applied in the situation that parameters are dynamically computed.

In this paper, we distinguish between *partially specified* and *fully specified* change operations and study the transition from partially to fully specified operations. For this purpose, we formalize change operations using graph transformation. We introduce the concept of an *applicable* change operation which ensures that a change operation produces a connected model. We establish the concept of an *enabled* operation which does not have any dependencies on another operation. We show that an enabled operation is always applicable and use this result to ensure that a connected process model is produced. We show how dependencies can be efficiently computed even for partially specified change operations based on an underlying decomposition of the process model into a process structure tree [22]. Using this decomposition, also conflict detection between change operations in distributed scenarios can be improved by reducing the number of required operation comparisons.

Throughout the paper, we present the theory for our approach along process models. However, we believe that the fundamental techniques can also be applied for other behavioral models where a tree-based representation of the model can be computed (such as statecharts).

The paper is organized as follows: We first introduce an example scenario where version management of business process models is demonstrated with a set of change operations. We then provide a formal model for change operations in Section 3 and explain our approach for computing position parameters of change operations. This provides the basis for introducing dependencies in Section 4 and conflicts in Section 5. We briefly report about tool support in Section 6 and conclude with related work and conclusions.

2 Business Process Model Version Management

We use business process models as our domain for model version management. In the following, we first introduce an example and then explain our approach which relies on the process structure tree to compute changes.

Figure 1 shows an example business process model V from the insurance domain using Business Process Model Notation (BPMN) [18]: Nodes can be *Activities*, *Gateways*, or *Events* such as Start and End. *Gateways* contain Exclusive/Inclusive/Complex Decision and Merge, and Parallel Fork and Join. Nodes are connected by control flow edges. 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. Figure 1 also shows a decomposition of the models into fragments (e.g. f_Z, f_X,..). A fragment can either be an

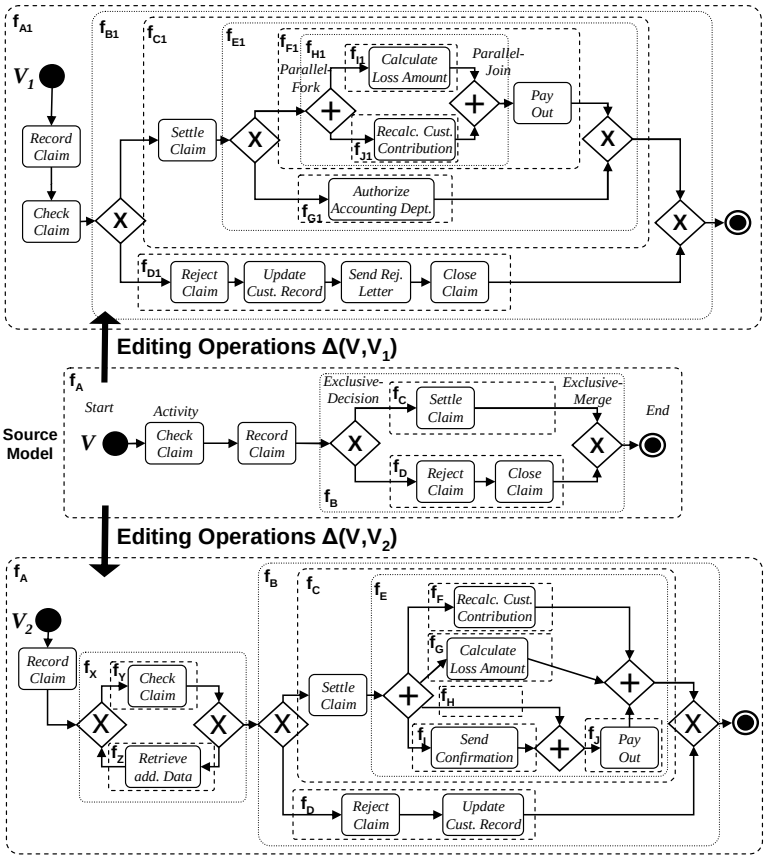


Fig. 1. An example scenario

alternative fragment consisting of an Exclusive Decision and an Exclusive Merge node, a concurrent fragment consisting of a Parallel Fork and a Parallel Join node or further types of fragments including unstructured or complex fragments which allow to express all combinations of gateways. Fragments can be organized into a Process Structure Tree (PST) of the process model [22].

In a distributed modeling scenario, the process model V (Figure 1) might have been created by the process model representative in an enterprise and then stored in a repository for further elaboration. During this elaboration period, two colleagues individually manipulate V to create new versions, e.g., V_1 and V_2. In our approach, changes performed by the colleagues to obtain V_1 and V_2 will be detected and collected in terms of change operations in a change log $\Delta(V, V_1)$ and $\Delta(V, V_2)$.

We have previously proposed change operations for process models [14] as follows: *InsertActivity*, *DeleteActivity* or *MoveActivity* operations allow to insert, delete or modify activities and always produce a connected process model as output. Similarly, *InsertFragment* and *DeleteFragment* operations can be used for inserting or deleting a complete

fragment of the process model. Figure 2 shows an overview of the change operations that are supported by our approach. Given an operation op we denote by $type(op)$ the type of the operation and assume the type as indicated in Figure 2. These change operations are computed by comparing the PSTs of two process models and identifying newly inserted, deleted and moved nodes in the PSTs (see [14] for a detailed introduction).

Change Operation op	Effects on Process Model V	type(op)
InsertActivity(x, a, b)	Insertion of a new activity x (by copying activity y) between two succeeding elements a and b in process model V and reconnection of control flow.	INSERTACT
DeleteActivity(x, c, d)	Deletion of activity x between c and d and reconnection of control flow.	DELETEACT
MoveActivity(x, c, d, a, b)	Movement of activity x from its old position between element c and d into its new position between two succeeding elements a and b in process model V and reconnection of control flow.	MOVEACT
InsertFragment(f_1, a, b)	Insertion of a new fragment f_1 between two succeeding elements a and b in process model V and reconnection of control flow.	INSERTFRAG
DeleteFragment(f_1, c, d)	Deletion of fragment f_1 between c and d from process model V and reconnection of control flow.	DELETEFRAG

Fig. 2. Change operations for process models [14]

In the case of the elaboration of V into V_1 in our example (Figure 1), we obtain the change operations given in the change log $\Delta(V, V1)$ in Figure 3. These change operations are initially partially specified. For example, for *c) InsertActivity("Pay Out",-,-)* the last two parameters are not fixed yet. These parameters which we denote as *position parameters* will be computed dynamically.

Δ(V, V1):
a) MoveActivity("Check Claim", -, -, -, -)
b) InsertFragment(f_{E1}, -, -)
c) InsertActivity("Pay Out", -, -)
d) InsertActivity("Authorize Accounting Dept.", -, -)
e) InsertFragment(f_{H1}, -, -)
f) InsertActivity("Calculate Loss Amount", -, -)
g) InsertActivity("Recalc. Cust. Contribution", -, -)
h) InsertActivity("Update Cust. Record", -, -)
i) InsertActivity("Send Rej. Letter", -, -)

Fig. 3. Change log $\Delta(V, V1)$

The operations in the change log can then be used to create a consolidated model V' out of V, V_1 and V_2. That means, the process model representative in the distributed modeling scenario, will inspect each change and decide which change to apply in order to construct a consolidated model V'. Thereby, he applies the changes in an iterative way and continues to do so until he is satisfied with the resulting model V'. With regards to Figure 1, the process model representative might first apply operation *i) InsertActivity("Send Rej. Letter",-,-)* and then operation *h) InsertActivity("Update Cust. Record",-,-)*. In this approach, the order of operation application leads to different position parameters of the change operations requiring dynamic computation of position parameters.

In contrast to that, fixing the position parameters in advance in terms of fully specified change operations restricts the order of application to one particular order and thereby restricts the user in creating a consolidated version. Figure 4 gives an example, assuming that operations $h)$ and $i)$ have not yet been applied. The fully specified operations *h) InsertActivity("Update Cust. Record", "Reject Claim", "Close Claim")* and *i) InsertActivity("Send Rej. Letter", "Update Cust. Record", "Close Claim")* restrict the application order to $h)$, $i)$. In addition, it is not possible to apply only the operation *i) InsertActivity("Send Rej. Letter", "Update Cust. Record", "Close Claim")*.

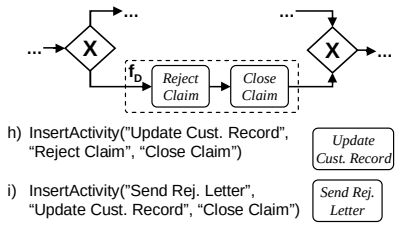


Fig. 4. Fully specified Change Operations restrict the Execution Order

One requirement for iterative change resolution is that when computing position parameters of change operations it must be ensured that they yield a change operation that can be applied on the process model. Further, a dependency concept for change operations is needed to ensure that only operations that do not depend on other operations can be applied. Otherwise it can happen that applying a change operation leads to a potentially unconnected model and problems when applying following change operations. For example, inserting an activity into a fragment that does not exist yet leads to problems when later inserting the fragment. Furthermore, in a concurrent modeling scenario as described above, an approach for computing conflicts is required as well. In the following sections, we will present our approach for addressing these problems.

3 Partially and Fully Specified Change Operations

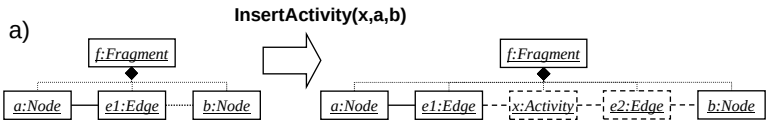
In this section, we formalize change operations using typed attributed graph transformation, distinguishing between fully specified and partially specified ones. Using this formalization, we define whether a fully specified change operation obtained from a partially specified one is applicable on the model.

3.1 Formalization of Change Operations

Each change operation op for a model V can be viewed as a model transformation rule on the model V transforming it to a model V'. A model transformation rule can be formalized as a typed attributed graph transformation rule [12, 5, 17]. We distinguish between change operation type and a concrete change operation: A change operation type (such as *InsertActivity(x,a,b)*) describes a set of concrete change operations. By replacing the parameters of a change operation type with model elements of the model V and V', a concrete change operation is obtained. Figure 5 b) shows a sequence of concrete change operations.

The behavior (or semantics) of a change operation type op is specified using a typed attributed graph transformation rule op_r. A typed graph transformation rule $op_r : L \rightarrow R$ 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 $G \overset{op_r(o)}{\Longrightarrow} H$, 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

a)



b)

Δ(V, V1):

1. MoveActivity(”Check Claim”, “Start”, “Record Claim”, “Record Claim”, “Exclusive Decision”)
2. InsertFragment(f_D, ”Settle Claim”, “Exclusive Merge”)
3. InsertActivity(”Pay Out”, “Parallel Fork”, “Parallel Join”)
4. InsertActivity(”Authorize Accounting Dept.”, “Parallel Fork”, “Parallel Join”)
5. InsertFragment(f_I, “Parallel Fork”, “Pay Out”)
6. InsertActivity(”Calculate Loss Amount”, “Parallel Fork:2”, “Parallel Join:2”)
7. InsertActivity(”Recalc. Cust. Contribution”, “Parallel Fork:2”, “Parallel Join:2”)
8. InsertActivity(”Update Cust. Record”, “Reject Claim”, “Close Claim”)
9. InsertActivity(”Send Rej. Letter”, “Update Cust. Record”, “Close Claim”)

Fig. 5. Change operation type and concrete change operations

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. Figure 5 a) shows the typed attributed graph transformation rule for *InsertActivity(x,a,b)*.

The theory of graph transformation provides the basis for defining the semantics of a change operation type as follows: Given a change operation type op together with its rule op_r, a concrete change operation on a model V leading to a model V' conforming to the type op is modelled by a change operation application of the rule op_r to V transforming it to V'. Formally, this is represented by a graph transformation $G \overset{op_r(o)}{\Longrightarrow} H$ where op_r is applied at an occurrence o to the graph G leading to a new graph H (where G and H are the type graphs obtained from the models V and V'). We also write $V \overset{op}{\Longrightarrow} V'$ or $V \overset{op(o)}{\Longrightarrow} V'$. To represent a concrete change operation, we write $op(o)$.

Formally, the occurrence morphism o represents a binding between the change operation type and the models V and V'. It maps nodes and edges of L and R to G and H. An occurrence morphism can be specified by a set of parameters $x_1, ..., x_n$ in the change operation type op and their instantiation in the change operation $op(o)$. We also write $op(x_1, .., x_n)$ for a change operation type and $op(X_1, ..., X_n)$ for a concrete change operation $op(o)$. For each rule op_r, we distinguish between parameters that are preserved, deleted or newly created, so $x_i \in pres(op_r) \cup del(op_r) \cup new(op_r)$.

As an example, consider the change operation type *InsertActivity(x,a,b)* and the change operation *InsertActivity(X,A,B)*. This implies an occurrence morphism mapping x to X, a to A and b to B where X, A, B are model elements in model V and/or V', and x is newly created whereas a and b are preserved elements. When designing a set of change operation types, we use a shorthand which only includes those elements of the occurrence morphism such that the morphism is uniquely determined. For example, we write *InsertActivity(x,a,b)* instead of *InsertActivity(f,a,e1,b,x,e2)* where $f, a, e1, b, x, e2$ refers to the elements defined in the transformation rule (see Figure 5 a)).

In model version management, an important concept is change operation applicability: Given a change operation $op(X_1, .., X_n)$, we want to reason whether this change operation is applicable to a model V or not:

Definition 1 (Applicable Change Operations) *Let a change operation $op(X_1, .., X_n)$ of a change operation type $op(x_1, .., x_n)$, its rule op_r and a model V be given, with $X_i \in V$ if $x_i \in pres(op_r) \cup del(op_r)$. Then we say that $op(X_1, .., X_n)$ is applicable to V if there exists a model V' with $V \overset{op_r(o)}{\Longrightarrow} V'$ for an occurrence o such that $x_i \mapsto X_i \in V$ if $x_i \in pres(op_r) \cup del(op_r)$ and $x_i \mapsto X_i \in V'$ if $x_i \in new(op_r)$. Otherwise, we say that $op(X_1, .., X_n)$ is not applicable on V.*

Figure 5 b) shows examples of applicable and non-applicable change operations. For example, *InsertActivity("Pay Out", "Parallel Fork", "Parallel Join")* is not applicable if $ParallelFork$ and $ParallelJoin$ do not exist in the model or are not connected by an edge as required by op_r. In other words, the applicability of the change operation *InsertActivity("Pay Out", "Parallel Fork", "Parallel Join")* depends on the chosen position parameters and might also be dependent on the application of another operation. Choosing correct position parameters is discussed in the following.

3.2 Correct Specification and Computation of Position Parameters

If all parameters of a concrete change operation are specified, we call the change operation *fully specified*. Otherwise, it is called *partially specified*. For example, *InsertActivity(X,-,-)* is a partially specified change operation because only x has been specified and a, b are not specified yet.

Fully specified change operations are obtained from partially specified ones by computing position parameters. In the following, we explain how the PSTs can be used for computing position parameters of change operations. Given two process structure trees $PST(V)$, $PST(V_1)$ and correspondences between their nodes, a joint PST, denoted as $J - PST(V, V_1)$, can be constructed which contains both process structure trees where corresponding nodes have been identified [14]. A J-PST can be annotated with change operations where each change operation is associated to the fragment node in the J-PST in which it occurs. In addition, for *InsertFragment* or *DeleteFragment* operations, we denote with $fragment(op)$ the newly inserted or deleted fragment. Figure 6 shows the annotated J-PST of the example.

When transitioning from a partially specified to a fully specified change operation, we require that parameters are chosen inside the parent fragment of an operation as follows:

Definition 2 (Correct specification) *Given a partially specified change operation op in a J-PST, a full specification of op is said to be correct, if the position parameters are chosen inside the parent fragment.*

With regards to Figure 6, a correct full specification of the *InsertActivity* operation ensures that the position parameters are chosen inside fragment f_I and not outside.

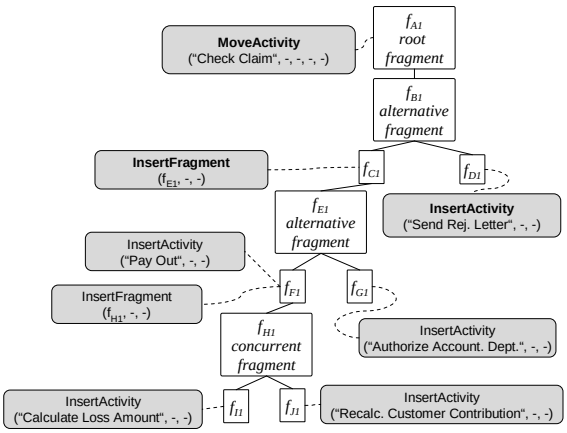


Fig. 6. The J-PST of the example

For applying a partially specified change operation, Algorithm 1 shows how to compute the position parameters. This algorithm starts at the element which is affected by the change operation and then searches backward and forward until a node is reached that exists in both process models. Note that this algorithm always returns a correct specification.

Although Algorithm 1 always returns a fully specified change operation, this does not ensure that the operation obtained is also correct according to Def. 2. For ensuring their correctness, dependencies between change operations must be taken into account which we will discuss in the next section.

4 Dependencies of Change Operations

In this section, we introduce concepts for dependencies of change operations. We first review dependencies for fully specified change operations and then elaborate on partially specified change operations.

As a fully specified change operation op is formally defined by a graph transformation rule op_r, we can directly apply the dependency concept from graph transformation (see e.g. [3, 17, 9, 13]): Informally, if two changes are dependent, then the second one requires the application of the first one. This is usually the case if the first change creates model structures that are required by the second change. Formally, we define:

Definition 3 (TR-Dependent Change Operations) *Let two fully specified change operations op_1 and op_2 be given such that $V \overset{op_1}{\Longrightarrow} V'$ and $V' \overset{op_2}{\Longrightarrow} V''$. Then we call op_2 transformation rule dependent (TR-dependent) on op_1 if op_2 is not applicable on V and op_2 is applicable on V'.*

Dependencies can be computed for change operations by applying existing theory for establishing a so-called dependency matrix (see [13] for an overview). An entry in

Algorithm 1 Computation of position parameters of a change operation op in model V and V_1

Procedure computePositionParameter(op,V,V_1):
$x = op.element$;
{Old Position Parameters of x in Model V}
if op is $DeleteActivity/Fragment$ or $MoveActivity$ **then**
  c = direct predecessor of $x \in V$; d = direct successor of $x \in V$;
{New Position Parameters of x in Model V}
if op is $InsertActivity/Fragment$ or $MoveActivity$ **then**
  a = getPredecessor(x,V,V_1); b = getSuccessor(x,V,V_1);
  if $a, b \neq$ null **then**
    if a is not directly connected to b **then**
      select an edge i between a and b; $a = i.source$; $b = i.target$;
  else
    select an edge i in V in the parent fragment of op; $a = i.source$; $B = i.target$;
return a, b, c, d;

{Computation of Predecessor}
Procedure getPredecessor(x,V,V_1):
determine predecessor p of x in V_1
if p exists in $V \wedge p$ is not affected by a Move operation **then**
  return p;
else
  return getPredecessor(p,V,V_1)
return null;

{Computation of Successor}
Procedure getSuccessor(x,V,V_1):
determine successor s of x in V_1
if s exists in $V \wedge s$ is not affected by a Move operation **then**
  return s
else
  return getSuccessor(s,V,V_1)
return null

this matrix then states the conditions under which two fully specified change operations are dependent.

Dependencies of partially specified change operations cannot be computed using the dependency matrix because parameters are missing. One possibility would be to apply dependency computation only to fully specified change operations, leading to the situation that dependencies are only detected late in the change resolution phase. Another approach is to use the annotated J-PST (see Figure 6) for defining dependencies of change operations as follows:

Definition 4 (J-PST Dependencies and Enabled Change Operations) *Let a J-PST annotated with change operations OPS be given. For each $op \in OPS$, we denote*

with $depops(op)$ all operations that are dependent on op. We define dependencies on the change operations as follows:

- *Let a change operation op be given with $type(op) =$ INSERTFRAG and let OPC be the set of all operations associated to a child of $fragment(op)$. Then every $op_i \in OPC$ is dependent on op and therefore $depops(op) = \{op_i \in OPC\}$.*
- *Let a change operation op be given with $type(op) =$ DELETEFRAG and let OPC be the set of all operations associated to a child of $fragment(op)$. Then every $op_i \in OPC$ is a prerequisite of op and therefore $op \in depops(op_i)$.*

We call a partially specified change operation op enabled if $depops(op) = \emptyset$.

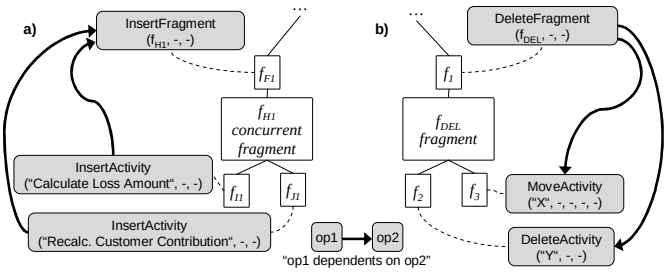


Fig. 7. Dependencies in the J-PST

The idea behind these dependencies is that a change operation that is dependent on an *InsertFragment* operation can only be applied if the *InsertFragment* operation has previously been applied to create the fragment. Similarly, a *DeleteFragment* operation is dependent on all other operations that affect the fragment deleted. This ensures that first all operations within the deleted fragment are applied before the fragment is deleted. Figure 7 a) shows an extract of the J-PST introduced earlier and the two *InsertActivity* change operations which are both dependent on the *InsertFragment* change operation. Figure 7 b) shows an example for *DeleteFragment* dependencies.

J-PST dependencies yield a dependency concept for change operations that are partially (or fully) specified. J-PST dependencies can be easily computed by traversing the J-PST and, for each fragment, computing dependencies between all operations associated to the fragment and operations associated to the grandfather fragment. J-PST dependencies have an important property by definition that we show in the following:

Lemma 1 (Acyclic J-PST dependencies). *Let a J-PST annotated with change operations be given. Then the J-PST dependencies are acyclic.*

In our approach, J-PST dependencies together with the concept of a correct specification can be used to show that an enabled operation is also applicable. The following theorem shows this:

Theorem 1 (Applicability of Enabled Operations). *Let a fragment f in the J-PST together with a change operation op be given. If op is enabled then it is also applicable if its position parameters are computed by Algorithm 1.*

For our application in model version management, it is important to know whether we can run into a dependency when applying fully specified change operations. The following theorem shows that it is sufficient to compute J-PST dependencies if operations are associated to different fragments in the J-PST:

Theorem 2 (TR-independence). *Let a J-PST annotated with change operations be given and let op_i and op_j be two operations. Assume further that op_i and op_j are attached to different fragments. If op_i and op_j are not J-PST dependent then op_i and op_j are not TR-dependent on each other.*

The previous theorem enables us to apply operations in different fragments without running into dependency problems. For change operations associated to the same fragment, the order of change operation application can lead to different results. However, by Theorem 1, each enabled operation is always applicable. This ensures that after applying an operation re-computation of position parameters of other operations leads to applicable operations again.

The previous theorems have the following consequences for iterative change resolution: All enabled operations are applicable which means that there exists a model V' obtained from V when applying an enabled change operation. As all transformation rules of our change operations produce connected process models, V' is always connected. Further, the order in which enabled operations are applied which are contained in different fragments does not matter.

In the next section, we will elaborate on conflicts that can arise when a process model has been changed concurrently by two persons.

5 Conflicts of Change Operations

In this section, we consider conflicts between change operations. Similar to the previous section about dependencies, we first introduce conflicts between fully specified change operations and then show how the J-PST can be used to ease up conflict computation.

Conflicts between change operations arise in scenarios where changes are applied independently on different versions of a process model. Our running example (see Figure 1) illustrates such a scenario where an original process model V is manipulated independently into two new versions V_1 and V_2.

In general, two changes are in conflict if only one of the two can be applied. This is the case if the two changes involve the same model structure and manipulate it in a different way. Typical conflicting pairs of change operations include the movement of an element in one model (e.g., *MoveActivity("Check Claim",-,-,-,-)*) and its deletion (e.g., *DeleteActivity("Check Claim",-,-)*) in the other model. Formally, we define:

Definition 5 (TR-Conflicting Change Operations) *Let two fully specified change operations op_1 and op_2 be given such that $V \overset{op_1}{\Longrightarrow} V'$ and $V \overset{op_2}{\Longrightarrow} V''$. Then we call op_1*

and op_2 transformation rule conflicting (TR-conflicting) if op_2 is not applicable on V' and op_2 is applicable on V.

Conflicts between fully specified change operations can be computed by applying existing theory, e.g., by establishing a conflict matrix (see [16, 13]) which specifies conditions under which two fully specified change operations are conflicting.

In contrast to dependencies, conflicts cannot be computed on partially specified change operations. However, using the J-PST still simplifies the conflict detection by decreasing the number of change operations, which need to be compared. Using the J-PST, conflicts can be computed by inspecting the set of enabled change operations between $\Delta(V, V_1)$ and $\Delta(V, V_2)$, instead of comparing all fully specified operations.

Given two joint process structure trees $J - PST(V, V_1)$, $J - PST(V, V_2)$, we first compute position parameters for enabled operations in the J-PSTs and then use the conflict matrix to identify conflicting operations. In the case that a conflicting operations is assigned to a fragment, all children are also marked as conflicting. Formally, we define conflicts of change operations in the J-PST:

Definition 6 (J-PST Conflicts) *Let two joint process structure trees $J - PST(V, V_1)$ and $J - PST(V, V_2)$, both annotated with operations, be given. Then we define conflicts on the change operations as follows:*

- *Two enabled change operations $op_{V1} \in \Delta(V, V_1)$ and $op_{V2} \in \Delta(V, V_2)$ are conflicting if they are TR-conflicting according to Definition 5.*
- *For two conflicting change operation op_{V1} and op_{V2} of the $type(op_{V1}) =$ INSERTFRAG, DELETEFRAG, let OPC_{V1} be the set of all operations associated to a child of $fragment(op_{V1})$. Then every $op_i \in OPC_{V1}$ is dependent on op_{V2}.*
- *For two conflicting change operation op_{V1} and op_{V2} of the $type(op_{V2}) =$ INSERTFRAG, DELETEFRAG, let OPC_{V2} be the set of all operations associated to a child of $fragment(op_{V2})$. Then every $op_i \in OPC_{V2}$ is dependent on op_{V1}.*

Using the J-PSTs for conflict detection reduces the number of required comparisons to the set of enabled operations. There is no need to compare all operations with each other. Figure 8 gives an example. The $J - PST(V, V_1)$ illustrates the alternative fragment f_{E1} which was inserted into process model V_1 (see our running example in Figure 1) and $J - PST(V, V_2)$ depicts the inserted concurrent fragment f_{E2} in V_2. The change operations *InsertFragment(f_{E1},-,-)* and *InsertFragment(f_{E2},-,-)* that insert these fragments into V_1 and V_2 are enabled and conflicting according to Definition 6, since only one of the operations can be applied in the merged version. Depending on the resolution of this conflict, the child operations contained in these fragments may also be conflicting. Thus, they are marked preventively as conflicting, as required by Definition 6.

In the following theorem we show that the number of conflicts in the J-PST constitute an upper bound for conflicting transformations, i.e. if two operations are not conflicting in the J-PST then they cannot be transformation conflicting.

Theorem 3 (TR-Conflicts are limited by J-PST Conflicts). *Let a J-PST annotated with operations be given and let op_i be an operation. Then no J-PST conflict between op_i and any other op_j induces that op_i is not transformation conflicting.*

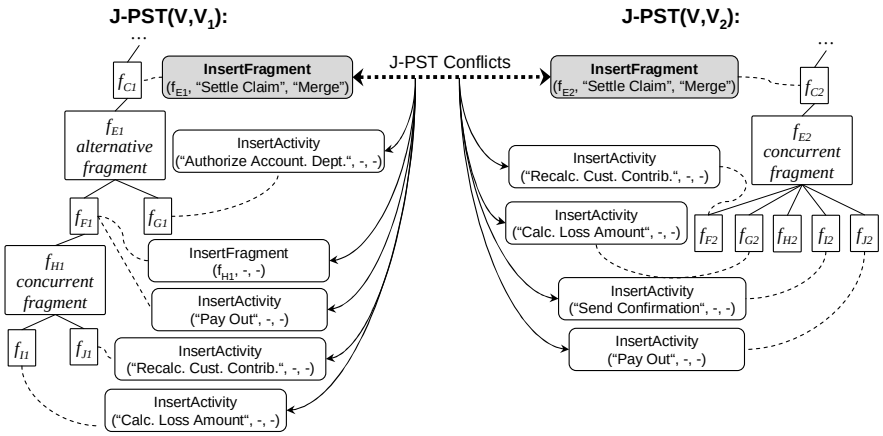


Fig. 8. Conflicts between Change Operations in the $J - PST(V, V_1)$ and $J - PST(V, V_2)$

Other than for dependencies, conflicts in the J-PST are not an abstraction but only an approximation of conflicts on the transformation rules and provide an upper bound of the overall number of conflicting transformation rules. This means that if two operations are conflicting in the J-PST then they may be giving rise to conflicting transformations but this is not always the case. For instance in Figure 8, the resolution of the conflict between the two enabled *InsertFragment(*f_E1*,-,-) InsertFragment(*f_E2*,-,-)* operations, which is not known prior to resolving the conflict (see [13] for several options of conflict resolution), determines possible conflicts between child operations. To avoid problems, we make all the child operations conflicting and recompute conflicts after resolving the conflict between enabled operations.

6 Tool Support

Our approach has been implemented in a prototype for process model version management in the IBM WebSphere Business Modeler and the IBM WebSphere Integration Developer. For an architectural overview we refer to [8]. In Figure 9 iterative change resolution is illustrated. In this view, only enabled change operations can be selected and applied, whereas all other operations are grayed-out. After the application of an enabled operation, the set of enabled operations is recomputed. Internally, the prototype uses the algorithms described above for computing position parameters and dependencies of partially specified change operations.

7 Related Work

One area of related work is concerned with model composition and model versioning. Alanen and Porres [1] describe an algorithm how to compute elementary change operations. Kolovos et al. [11] describe the Epsilon merging language which can be used to

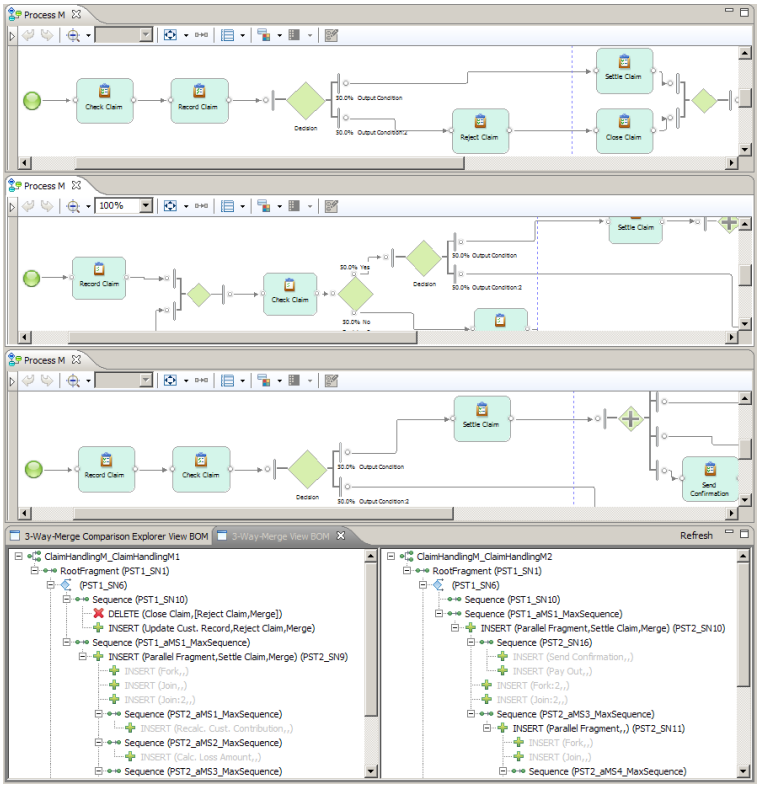


Fig. 9. Screenshot of our prototype in WebSphere Business Modeler

specify how models should be merged. Kelter et al. [10] present a generic model differencing algorithm. In the IBM Rational Software Architect [15] or using the EMF Compare technology [6], dependencies and conflicts between versions are computed based on elementary changes. These approaches to model versioning deal with the problem of model merging but they do not make use of a model stucture tree (such as our process structure tree). This enables us to compute position parameters for partially specified change operations and thereby realize iterative change resolution.

In the area of software evolution, Fluri et al. [7] describe how to use abstract syntax trees and tree differencing for extracting changes and better understanding change types. Our work uses the process structure trees on the modeling level. One difference can be seen in the way change resolution is performed because on the modeling level more flexibility is needed which is incorporated by the concept of partially specified change operations.

Dependencies of transformation rules have been studied in the literature: Mens et al. [17] analyze refactorings for dependencies using critical pair analysis. They first express refactorings as graph transformations and then detect dependencies using the AGG tool [21]. Graph transformations have also been used extensively for defining and parsing visual languages [19] where rules are used as parsing rules. Further, graph

transformation rules have been used in various model transformation approaches (see e.g. [4, 2]), as a formal foundation as well as in transformation engines executing a model transformation. In these approaches, a transformation rule is matched and applied along existing theory of graph transformation [3]. In contrast to these approaches, we study dependencies of change operations that are partially specified using J-PST dependencies and establish a relationship to TR-dependencies. In our earlier work [13], we assumed that all change operations are fully specified.

Within the process modeling community, Rinderle et al. [20] have studied disjoint and overlapping process model changes in the context of the problem of migrating process instances but have not considered dependencies between changes and different forms of change resolution.

8 Conclusion

Version management of behavioral models such as process models or statecharts poses specific requirements on change operations: Typically, a user cannot be forced to resolve changes in a certain order and further he/she is supposed to select those changes that are to be resolved. This requires the identification of dependencies of change operations and dynamic computation of position parameters of change operations.

In this paper, we have introduced the concept of a partially specified change operation where position parameters are dynamically computed on demand. We have established a formal model for partially specified and fully specified change operations, based on the theory of graph transformation. We have then introduced an approach for computing dependencies and conflicts of change operations based on an underlying tree-based decomposition of the model which greatly reduces the number of comparisons needed. We have shown how to use these dependencies within iterative change resolution.

There are several directions of future work: An interesting question is how our approach can be extended to other models such as statecharts. Further, as our change operations are essentially model transformations with changing parameters it arises the question whether dynamic computation of model transformation parameters are also required in other application scenarios.

Acknowledgements: The authors wish to thank Jana Koehler for valuable feedback on an earlier version of this paper.

References

1. 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.
2. E. Biermann, C. Ermel, and G. Taentzer. Precise semantics of emf model transformations by graph transformation. In K. Czarnecki, I. Ober, J. Bruel, A. Uhl, and M. Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 2008.
3. 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.

4. G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models . In *Proceedings ASE'02*, pages 267–270, September 2002.
5. J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.*, 376(3):139–163, 2007.
6. Eclipse Foundation. EMF Compare. http://www.eclipse.org/modeling/emft/?project=compare.
7. B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.
8. C. Gerth, J. M. Küster, and G. Engels. Language-Independent Change Management of Process Models. In A. Schürr and B. Selic, editors, *MODELS'09*, volume 5795 of *LNCS*, pages 152–166. Springer, 2009.
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. 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.
11. 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.
12. J. M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 5(3):233–259, 2006.
13. J. M. Küster, C. Gerth, and G. Engels. Dependent and Conflicting Change Operations of Process Models. In R. Paige, A. Hartman, and A. Rensink, editors, *ECMDA-FA'09*, volume 5562 of *LNCS*, pages 158–173. Springer-Verlag, 2009.
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.
16. T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Trans. Software Eng.*, 28(5):449–462, 2002.
17. T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 6(3):269–285, 2007.
18. Object Management Group (OMG). Business Process Modeling Notation (BPMN). http://www.omg.org/spec/BPMN/1.2.
19. J. Rekers and Andy Schürr. Defining and Parsing Visual Languages with Layered Graph Grammars. *J. Vis. Lang. Comput.*, 8(1):27–55, 1997.
20. 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.
21. G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *AGTIVE*, volume 3062 of *LNCS*, pages 446–453, 2003.
22. 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.
23. 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 Proofs

Proof Sketch of Lemma 1: *Proof by contradiction. According to the definition, there are two types of J-PST dependency edges, either INSERTFRAG or DELETEFRAG. Assume that there exists a cycle in the dependencies i.e.* $\langle op_1, .., op_k \rangle$ *such that* $op_{j+1} \in depops(op_j)$ *for* $j = 1, ..k-1$ *and* $op_1 \in depops(op_k)$*. Label each dependency edge with the type. The cycle cannot consist of DELETEFRAG or INSERTFRAG edges only because these edges either go downward in the J-PST or upward. Therefore there must be one* op_i *in the cycle that has an incoming DELETEFRAG and an outgoing INSERTFRAG edge or vice versa. The first case cannot occur because this implies that* op_i *has both types which is a contradiction. In the second case, this leads to the situation that at some point in the cycle the first case must occur which again is a contradiction.*

Proof Sketch of Theorem 1: *If an operation* op *is enabled, we compute its full specification by Algorithm 1. This full specification is a correct specification, because the parent fragment of an enabled operation exists in the process model. Thus, Algorithm 1 will return position parameters inside the parent fragment. For showing that* op *is applicable, we consider its transformation rule. For applicability we have to ensure that* $op_r : L \rightarrow R$ *can be applied and that the properties of the occurrence* o *are fulfilled, see Definition 1. This has to be shown for each rule* op_r *separately. In the following, we consider InsertActivity(x,a,b). Due to the algorithm, the computed position parameters are model elements in* V*.* x *has been computed by the change operation detection algorithm and is a model element in* V_1*. We can therefore define an occurrence* o *which conforms to Definition 1. What remains to be shown is that there is indeed a match in* V *for* o *and* op_r*. For InsertActivity(x,a,b),* L *requires that* f *contains* a *and* b *directly connected by an edge (see Figure 5 a)). As this is the postcondition of the algorithm 1 this is fulfilled. For other change operation types, the proof follows the same line of argumentation.*

Proof Sketch of Theorem 2: *Proof by Contradiction. Assume that two operations in different fragments exist and that no J-PST dependency exists between them. Let us now assume that they are TR-dependent on each other when their position parameters are fixed. We consider the dependency matrix in [13] which shows the TR-dependencies of two operations* op_i *and* op_j*. The entries of the matrix specify when two operations are TR-dependent. Each entry requires that the two operations have at least one common parameter. This ensures for all combinations where* op_i *and* op_j *are Insert/Delete/MoveActivity operations that the operations are associated to the same fragment node in the J-PST which is a contradiction. For all other combinations involving InsertFragment or DeleteFragment, the common parameter ensures that the two operations are associated to the same fragment node or it is the case that an* $entry(F)$ *or* $exit(F)$ *occurs in the parameters where* F *is the fragment manipulated by InsertFragment or DeleteFragment. In this case, by Definition 4, a J-PST dependency must exist between the two operations which is again a contradiction.*

Proof Sketch of Theorem 3: *Proof by contradiction. We can distinguish two cases: Operation* op_i *is either enabled (is not dependent on any other* op_j*) or* op_i *is not enabled and requires the application of another* op_j *before it becomes enabled.*

Case 1. If op_i is enabled, let us assume that op_i has no J-PST conflict, although its transformation is conflicting to another transformation of another enabled operation op_x. Since op_i is not J-PST conflicting, we can deduce that the position parameters of op_i do not overlap with the position parameters of any other enabled operation according to the conflict matrix in [13] by Definition 6. This means that no other enabled operation modifies the process model in that particular place, thus op_i cannot be transformation conflicting with op_x which is a contradiction.

Case 2. If op_i is not enabled, let us assume that op_i has no conflict in the J-PST, although its transformation is conflicting. Since op_i is not enabled, there exists at least one operation op_j which has to applied before op_i becomes applicable, i.e. op_i is dependent on op_j. Hence, the type of op_j is $type(op_j) = "INSERTFRAG"$, otherwise op_i would not depend on op_j according to Definition 4. Without loss of generalization, we assume that op_j is enabled. According to Definition 6 op_j cannot be conflicting in the J-PST or on the transformation level (Case 1.), otherwise op_i as a child of $fragment(op_j)$ would be conflicting too. Since the position parameter of op_i must be inside its parent fragment $fragment(op_j)$, op_i cannot be transformation conflicting which is a contradiction.