Detecting and Resolving Process Model Differences in the Absence of a Change Log

Jochen M. Küster[1], Christian Gerth[1,2], Alexander Förster[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,alfo,engels}@upb.de

Abstract. Business-driven development favors the construction of process models at different abstraction levels and by different people. As a consequence, there is a demand for consolidating different versions of process models by detecting and resolving differences. Existing approaches rely on the existence of a change log which logs the changes when changing a process model. However, in several scenarios such a change log does not exist and differences must be identified by comparing process models before and after changes have been made. In this paper, we present our approach to detecting and resolving differences between process models, in the absence of a change log. It is based on computing differences and deriving change operations for resolving differences, thereby providing a foundation for variant and version management in these cases.

Keywords: process change management, process model differences.

1 Introduction

The field of business process modeling has a long standing tradition. Recently, new requirements and opportunities have been identified which allow the tighter coupling of business process models to its underlying IT implementation: In Business-Driven Development (BDD) [11], business process models are iteratively refined, from high-level business process models into models that can be directly executed. In such scenarios, a given business process model can be manipulated by several people and different versions of the original model can be created. At some point in time, these different versions need to be consolidated in order to integrate selected information found in different versions into a common process model. Technically, this consolidation involves inspecting differences and resolving differences by performing change operations on the original model in order to integrate information found in one of the versions.

Detection of differences introduced into a process model is straightforward in a process-aware information system [5,15] that provides change logs (see e.g. [21]). However, there exist scenarios where such a change log is not available: Either the tool does not provide one or process models are exchanged across tool boundaries. In such situations, detection of differences has to be performed by comparing process models before and after changes have been made. For each difference detected, appropriate change operations have to be derived which together can be considered as a reconstructed change log.

In addition, process modeling tools have to fulfill specific requirements concerning user-friendliness: The business user (usually not a computer scientist) should be able to inspect and resolve differences. As a consequence, differences must be displayed in a form that is understandable by the business user, by grouping related differences or those differences that can be resolved together. Similarly, resolution of differences should involve compound change operations (rather than primitive change operations) which enable the business user to deal with differences such as insertion or deletion of a task and automate the reconnection of control flow in the process model.

In this paper, we present our solution to the problem of computing differences, displaying differences and resolving differences between two process models in the situation that *no change log* is available. Detection of differences makes use of the concept of correspondences [13], well-known from model merging and model composition, but enriched with the technique of Single-Entry-Single Exit fragments (SESE fragments) [20]. Using SESE fragments we are able to associate each difference with a compound change operation that resolves the difference. Overall, our approach provides a foundation for variant and version management in cases where no change log is available which is a common situation in process modeling tools and in scenarios where process models are exchanged across tool boundaries.

The paper is structured as follows: Section 2 introduces our scenario for detection and resolution of differences and describes key requirements. Then, in Section 3, we discuss the foundations for our approach, correspondences and SESE fragments. In Section 4 and Section 5 we present our approach for difference detection and visualization. In Section 6, our approach to difference resolution is described and a prototype for initial validation is presented. We conclude with a discussion of related and future work.

2 A Scenario for the Detection and Resolution of Differences

In business-driven development, business process models are manipulated by several persons and multiple versions of a shared process model need to be consolidated at some point in time. A basic scenario is obtained when a process model V_1 is copied and then changed into a process model V_2, possibly by another person. After completion, only some of the changes shall be applied to the original model V_1 to create a consolidated process model. Figure 1 shows an example process model V_1 that has been changed into a process model V_2. In the following, we use process models in a notation similar to activity diagrams in UML [12].

Both models describe the handling of a claim request by an insurance company. V_1 starts with an *InitialNode* followed by the *actions "Check Claim"* and *"Record Claim"*. Then, in the *Decision*, it is checked whether the claim is covered by the insurance contract or not. In the case of a positive result the claim is settled. In the other case the claim is rejected and closed, represented by the *actions "Reject Claim"* and *"Close Claim"*. We now assume that V_2 is derived from V_1 by another business user who introduces and deletes elements, with the final result that is shown in Figure 1. A manual inspection of the process models V_1 and V_2 leads to the identification of the following differences:

- The *action "Check Claim"* is moved into a newly inserted cyclic structure. In addition a new *action "Retrieve additional Data"* is inserted into the cyclic structure.

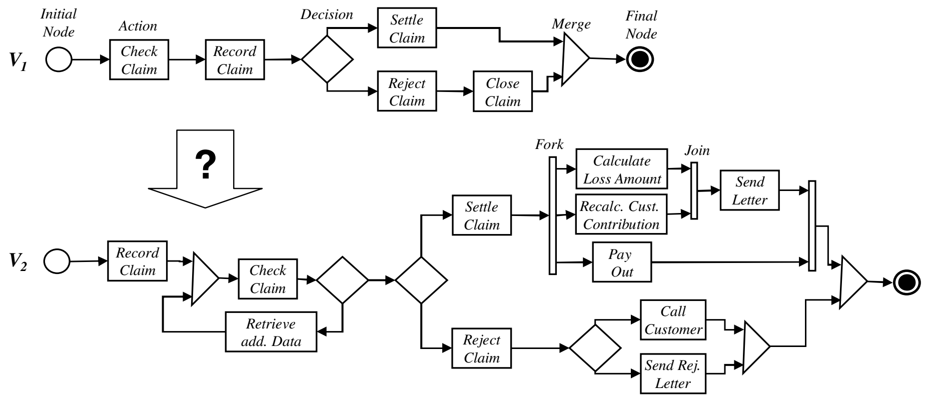


Fig. 1. Versions V_1 and V_2 of a business process model

- A parallel structure (*Fork* and two *Joins*) is introduced in V_2 containing four *actions "Calculate Loss Amount", "Recalculate Customer Contribution", "Pay Out", and "Send Letter"*.
- *Action "Close Claim"* has been deleted in V_2.
- A new alternative structure (*Decision* and *Merge*) is inserted in V_2 together with two *actions "Call Customer"* and *"Send Rejection Letter"*.

```
- deleteEdge(InitialNode, "Check Claim")
- deleteEdge("Check Claim", "Record Claim")
- addEdge(InitialNode, "Record Claim")
- deleteEdge("Record Claim", Decision)
- addControlNode(Merge)
- addEdge("Record Claim", Merge)
- addEdge(Merge, "Check Claim")
- addControlNode(Decision)
- ...
```

Fig. 2. Primitive change operations applied to V_1 in order to obtain V_2

For larger process models, manual identification of differences represents a large overhead. As such, techniques for detecting and resolving differences between process models are required which typically depend on the modeling language as well as on constraints of the modeling environment. In our scenario, we assume that no change log is available. A simple approach would then compute all changed elements and express them using primitive change operations as displayed in Figure 2. For the business user, such a change log is difficult to handle because the relationship between change operations and the process model elements is difficult to determine. Furthermore, changes are not grouped into compound change operations [21] which package several related primitive change operations. For example, for inserting the new alternative structure with the *Decision* and *Merge*, the business user has to insert these two nodes by appropriate *addControlNode* operations, delete edges by *deleteEdge* operations and insert new edges by *insertEdge* operations. This is in contrast to a compound operation that comprises all these change primitives, being close to the conceptual understanding of the change. As a consequence, we define the following requirements a solution for detection and resolution of differences should fulfill:

- (Detection) The solution must provide a technique to re-construct one possible change log which represents the transformation steps for transforming one process model into the other process model.

- (Visualization) Differences should be grouped and associated to areas where they occur in order to improve usability by the business user.
- (Resolution) The solution should enable the business user to resolve differences using compound change operations rather than change primitives manipulating individual process model elements.
- (Resolution) The business user should have the opportunity to select only some of the changes and apply them in any order when possible.

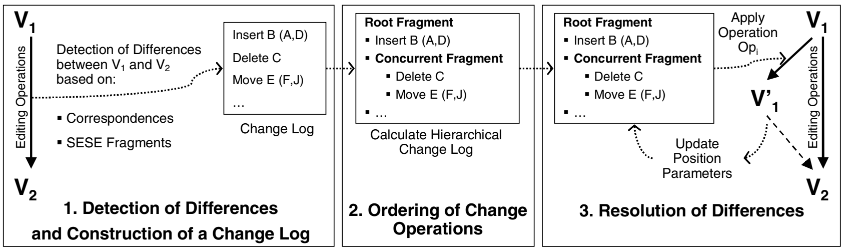

V1
Editing Operations
V2
Detection of Differences between V1 and V2 based on:
▪ Correspondences
▪ SESE Fragments
Insert B (A,D)
Delete C
Move E (F,J)
...
Change Log
1. Detection of Differences and Construction of a Change Log
Root Fragment
▪ Insert B (A,D)
▪ Concurrent Fragment
▪ Delete C
▪ Move E (F,J)
▪ ...
Calculate Hierarchical Change Log
2. Ordering of Change Operations
Apply Operation Opi
V'1
Update Position Parameters
3. Resolution of Differences

Fig. 3. Overview of our process merging approach

Figure 3 provides an overview of the approach that we have developed based on these requirements: The first step is to detect differences between the two process models. This detection makes use of correspondences and SESE fragments. For each difference, a change operation is generated which resolves the difference between the two models. In the second step, change operations are ordered according to the structure of the process models. The third step is then to resolve differences between the process models in an iterative way, based on the business user's preferences.

3 Correspondences and SESE Fragments

In this section, we first define business process models and provide a summary of the concepts of Single-Entry-Single-Exit fragments [20]. Then we introduce correspondences. Fragments and correspondences will be later used for detecting differences and also provide a basis for deriving change operations.

3.1 Process Models and SESE Fragments

For the following discussions, we assume a business process model $V = (N, E)$ consisting of a finite set N of nodes and a relation E representing control flow. N is partitioned into sets of *Actions* and *ControlNodes*. *ControlNodes* contain Decision and Merge, Fork and Join, InitialNodes and FinalNodes. In addition, we assume that the following constraints hold:

1. *Actions* have exactly one incoming and one outgoing edge.
2. Nodes are connected in such a way that each node is on a path from the InitialNode to the FinalNode.

3. Control flow splits and joins are modeled explicitly with the appropriate *ControlNode*, e.g. *Fork*, *Join*, *Decision*, or *Merge*. *ControlNodes* have either exactly one incoming and at least two outgoing edges (*Fork*, *Decision*) or at least two incoming and exactly one outgoing edge (*Join*, *Merge*).
4. An *InitialNode* has no incoming edge and exactly one outgoing edge and a *FinalNode* has exactly one incoming edge and no outgoing edge.
5. A process model contains exactly one *InitialNode* and exactly one *FinalNode*.

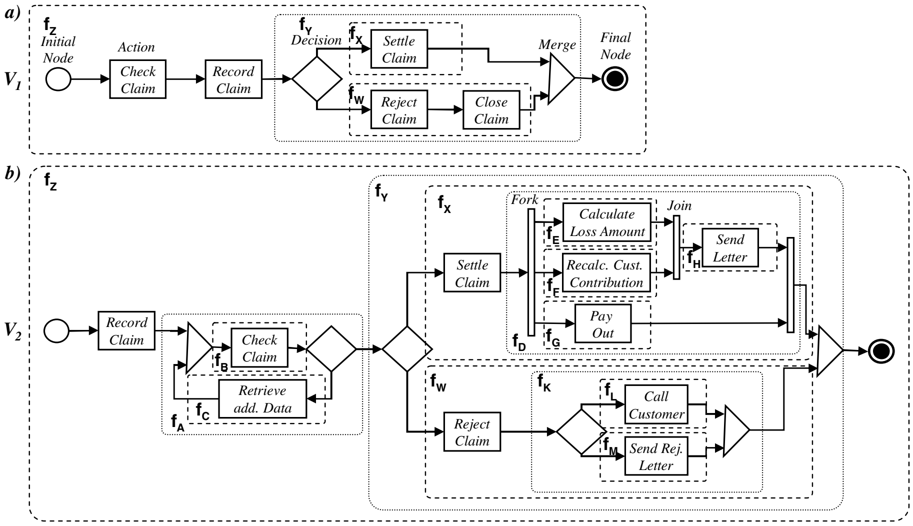


Fig. 4. Versions V_1 and V_2 decomposed into canonical SESE fragments

In general, process models can be decomposed into SESE fragments [20]. A SESE fragment is a non-empty subgraph in the process model with a single entry and a single exit edge. The fragment which surrounds the entire process model is also considered as a SESE fragment which we refer to as *root fragment*. Among all possible SESE fragments, we select so-called canonical fragments which are not overlapping on the same hierarchical level and denote them with $\mathcal{F}(V)$ for a given process model V. Figure 4 shows an example of a SESE decomposition into canonical fragments, with fragments visualized by a surrounding of dotted lines.

The canonical fragments of a process model V can be organized into a process structure tree (PST) [20], denoted by $PST(V)$, according to the composition hierarchy of the fragments (see Figure 5 for the tree obtained for V_1). If a fragment f_1 contains another fragment f_2 (respectively node n), then f_1 will be the parent of fragment f_2 (node n) in this tree and fragment f_2 (node n) will be one of its children. Further, the root of the tree is the root fragment. We distinguish between different types of fragments as follows [20]:

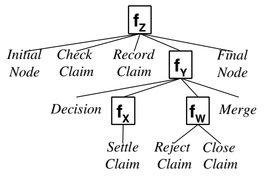


Fig. 5. Process structure tree of V_1

– a well-structured fragment f is either a sequential, a sequential branching, a cyclic, or a concurrent branching fragment. For example, in Figure 4, f_Y is a well-structured sequential branching fragment and f_W is a well-structured sequential fragment.
– an unstructured concurrent fragment f is not well-structured and contains no cycles and has no decisions and no merges as children. In Figure 4, f_D is an unstructured concurrent fragment.
– an unstructured sequential fragment f is not well-structured and has no forks and no joins as children.
– a complex fragment f is any other fragment that is none of the above.

Given a fragment $f \in \mathcal{F}(V)$, we denote by $type(f)$ the type of the fragment and by $frag(f)$ the parent fragment. Similarly, given a node $x \in N$, we denote by $type(x)$ the type of the node and $frag(x)$ the parent fragment of x. For example, $type(x) = Action$ means that x is an *Action* node.

SESE fragments have been used successfully for checking soundness [18,14] of process models but they are also beneficial for detection of differences between process models, discussed later in this paper.

3.2 Correspondences

Correspondences are useful for the detection of differences because they provide the link between elements in different process models. We assume process models $V_1 = (N_1, E_1)$ and $V_2 = (N_2, E_2)$ and $x \in N_1$ and $y \in N_2$ as given. A correspondence is used to express that a model element x has a counterpart y with the same functionality in the other version. In such a case, we introduce a 1-to-1 correspondence between them. In the case that a model element x does not have a counterpart with the same functionality, we speak of a 1-to-0 correspondence. In case that y does not have a counterpart, we speak of a 0-to-1 correspondence. In addition, refinement of an element into a set of elements would give rise to a 1-to-many correspondence and abstraction of a set of elements into one element would give rise to a many-to-1 correspondence. These last two types are not needed in our scenario.

We express a 1-to-1 correspondence by inserting the tuple (x, y) into the set of correspondences $\mathcal{C}(V_1, V_2) \subseteq N_1 \times N_2$. We further introduce the set of elements in V_1 which do not have a counterpart and denote this set by $\mathcal{C}_{1-0}(V_1, V_2)$. Similarly, we denote the set of elements in V_2 without counterparts as $\mathcal{C}_{0-1}(V_1, V_2)$. Similarly, we introduce correspondences for SESE fragments, expressed in the sets $\mathcal{C}^F(\mathcal{F}(V_1), \mathcal{F}(V_2))$, $\mathcal{C}^F_{1-0}(\mathcal{F}(V_1), \mathcal{F}(V_2))$, and $\mathcal{C}^F_{0-1}(\mathcal{F}(V_1), \mathcal{F}(V_2))$.

In our scenario, we assume that the functionality of a node remains the same if it is copied. Correspondences can then be computed in a straightforward way by first establishing 1-to-1 correspondences between all nodes [1] (respectively fragments) of a process model when copying process model V_1 to create an initial V_2. After obtaining the final V_2 by editing operations, all 1-to-1 correspondences have to be inspected and 1-to-0 or 0-to-1 correspondences are created if nodes (respectively fragments) have

[1] Correspondences are internally based on the unique identifiers of elements.

been deleted or added. In addition, for new nodes (respectively fragments) in V_2, additional 0-to-1 correspondences have to be created. In other scenarios across tool boundaries, other means of correspondence computation are required which might involve semantic matching techniques.

Fig. 6 shows correspondences between versions V_1 and V_2 of the process model introduced earlier in this paper. A dotted line represents 1-to-1 correspondences and connects model elements with the same functionality between V_1 and V_2. 1-to-0 correspondences are visualized by dotted elements in V_1 and 0-to-1 correspondences are visualized by dotted elements in V_2.

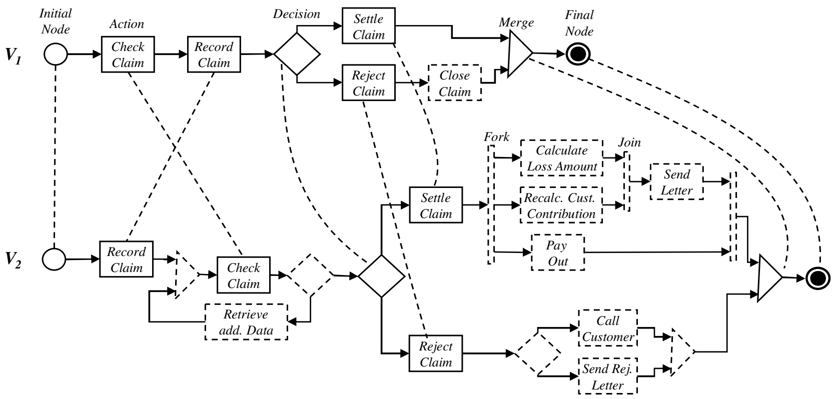


Fig. 6. Correspondences between nodes of V_1 and V_2

We further assume a partial ordering relation on actions and fragments of a process model, restricted to a partial ordering within each fragment. The partial orders will be later used for detecting moved elements. Given an element x (fragment or action), we denote by $order_f$ the partial order of elements in fragment f derived by the control flow order of elements within f and we write $x <_f y$ for x smaller than y according to this order[2]. Let a tuple of actions or fragments $(x, y) \in \mathcal{C}(V_1, V_2) \cup \mathcal{C}^F(\mathcal{F}(V_1), \mathcal{F}(V_2))$ and $frag(x) = f_1$ and $frag(y) = f_2$ be given. Then we write $order_{f_1}(x) \neq order_{f_2}(y)$ if and only if there exists an element $(z_1, z_2) \in \mathcal{C}(V_1, V_2) \cup \mathcal{C}^F(\mathcal{F}(V_1), \mathcal{F}(V_2))$ with $frag(z_1) = f_1$ and $frag(z_2) = f_2$ such that $z_1 <_{f_1} x$ and $z_2 >_{f_2} y$, or $z_1 >_{f_1} x$ and $z_2 <_{f_2} y$, or z_1 and x are unordered and z_2 and y are ordered, or z_1 and x are ordered and z_2 and y are unordered.

4 Detection of Differences

In this section, we describe an approach to detect differences between process models, based on the existence of correspondences and SESE fragments.

[2] For cyclic fragments, we assume an order obtained by a depth-first search of the fragment along the control flow edges.

4.1 Action and Fragment Differences and Change Operations

The correspondences between two process models can be used to identify differences. One form of differences that can occur are those that result from adding, deleting or moving actions, as defined in the following:

Definition 1 (Action Differences). *Given two business process models V_1, V_2 and sets of correspondences $\mathcal{C}_{1-0}(V_1, V_2)$, $\mathcal{C}_{0-1}(V_1, V_2)$ and $\mathcal{C}(V_1, V_2)$, we define the following action differences:*

- *an InsertAction difference is defined as an element $y \in \mathcal{C}_{0-1}(V_1, V_2)$ and type$(y) =$ Action,*
- *a DeleteAction difference is defined as an element $x \in \mathcal{C}_{1-0}(V_1, V_2)$ and type$(x) =$ Action,*
- *a MoveAction difference is defined as a tuple of actions $(x, y) \in \mathcal{C}(V_1, V_2)$ and either $(frag(x), frag(y)) \notin \mathcal{C}^F(\mathcal{F}(V_1), \mathcal{F}(V_2))$ or $((frag(x), frag(y)) \in \mathcal{C}^F(\mathcal{F}(V_1), \mathcal{F}(V_2))$ and $order_{frag(x)}(x) \neq order_{frag(y)}(y))$.*

The identification of *InsertAction* and *DeleteAction* differences is straightforward. With regards to *MoveAction* differences, we distinguish between *intra-fragment* differences where the action has been moved within corresponding SESE fragments and *inter-fragment* differences where actions have been moved between SESE fragments. The detection of inter-fragment differences can be done by iterating over all 1-to-1 correspondences and checking whether the surrounding SESE fragments are also in a 1-to-1 correspondence. If this is not the case, then the element has been moved and is considered as an inter-fragment difference. The detection of intra-fragment differences has to compare all elements within a fragment with the elements in the corresponding fragment and identify changes in the order of elements. Each action difference can be directly converted into a suitable *InsertAction*, *DeleteAction* or *MoveAction* operation which resolves the difference, shown in Figure 7. The position parameters *a* and *b* specify the position where action *x* is inserted or moved to in process model V_1.

Compound Change Operation applied on V	Effects on Process Model V
InsertAction(V,x,a,b)	Insertion of a new action x (by copying action y) between two succeeding elements a and b in process model V and reconnection of control flow.
DeleteAction(V,x)	Deletion of action x and reconnection of control flow.
MoveAction(V,x,a,b)	Movement of action x between two succeeding elements a and b in process model V and reconnection of control flow.

Fig. 7. Overview of compound change operations for actions

In addition to action differences, different versions of process models can also be constructed by introducing or removing control nodes, as well as deleting or changing edge connections involving such control nodes. These changes give rise to differences concerning the fragment structure of the process models and are defined as follows:

Definition 2 (Fragment Differences). *Given two business process models V_1 and V_2 and sets of correspondences between SESE fragments $\mathcal{C}^F(\mathcal{F}(V_1), \mathcal{F}(V_2))$, $\mathcal{C}^F_{1-0}(\mathcal{F}(V_1), \mathcal{F}(V_2))$, and $\mathcal{C}^F_{0-1}(\mathcal{F}(V_1), \mathcal{F}(V_2))$, we define the following fragment differences:*

- *an InsertFragment difference is defined as a SESE fragment $f_2 \in \mathcal{C}^F_{0-1}(\mathcal{F}(V_1), \mathcal{F}(V_2))$.*
- *a DeleteFragment difference is defined as a SESE fragment $f_1 \in \mathcal{C}^F_{1-0}(\mathcal{F}(V_1), \mathcal{F}(V_2))$.*
- *a MoveFragment difference is defined as a tuple of fragments $(f_1,f_2) \in \mathcal{C}^F(\mathcal{F}(V_1), \mathcal{F}(V_2))$ and either $(frag(f_1),frag(f_2)) \notin \mathcal{C}^F(\mathcal{F}(V_1), \mathcal{F}(V_2))$ or $((frag(f_1),frag(f_2)) \in \mathcal{C}^F(\mathcal{F}(V_1), \mathcal{F}(V_2))$ and $order_{frag(f_1)}(f_1) \neq order_{frag(f_2)}(f_2))$.*
- *a ConvertFragment difference occurs if the type of the fragment has changed or f_1 has a control node as child that has no counterpart in f_2 or f_2 has a control node as child that has no counterpart in f_1.*

The identification of *InsertFragment*, *DeleteFragment* and *MoveFragment* differences is analogous to action differences. Identification of *ConvertFragment* differences involves iteration over all tuples $(f_1,f_2) \in \mathcal{C}^F(\mathcal{F}(V_1), \mathcal{F}(V_2))$ and examining whether the type of f_1 or f_2 has changed or whether one of the fragments has a control node as child that has no counterpart in the other fragment.

Each fragment difference described can be resolved by an appropriate change operation. An overview of the operations and their effect on a process model is given in Figure 8. Note that here for the *InsertFragment* difference a number of different change operations is given, inserting the fragment of suitable type into the process model. The type can be determined by inspecting the fragment in V_2.

Figure 9 shows one possible set of compound change operations obtained for the example earlier in this paper. Here, *InsertCyclicFragment*$(V_1, \_, \_, f_A)$ will insert a new cycle into V_1, *MoveAction*$(V_1,$ *"Check Claim"*$, ...)$ moves *"Check Claim"* to its

Compound Change Operation applied on V	Effects on Process Model V
InsertFragment(V,a,b,f_2) *The generic operation InsertFragment is realized by:* • *InsertParallelFragment(V,a,b,f_2)* • *InsertAlternativeFragment(V,a,b,f_2)* • *InsertSequentialFragment(V,a,b,f_2)* • *InsertCyclicFragment(V,a,b,f_2)* • *InsertUnstructuredConcurrentFragment(V,a,b,f_2)* • *InsertUnstructuredSequentialFragment(V,a,b,f_2)* • *InsertComplexFragment(V,a,b,f_2)*	Insertion of a new fragment f_1 between two succeeding elements a and b in process model V, copying the structure of f_2, and reconnection of control flow.
DeleteFragment(V,f_1)	Deletion of fragment f_1 from process model V and reconnection of control flow.
MoveFragment(V,f_1,a,b)	Move of fragment f_1 between two succeeding elements a and b in process model V and reconnection of control flow.
ConvertFragment(V,f_1,f_2)	Conversion of a fragment f_1 into the fragment type of f_2, replacing the structure from f_1 with the structure of f_2, and reconnection of control flow.

Fig. 8. Overview of compound change operations for fragments

- InsertCyclicFragment(V_1, _, _, f_A)
- MoveAction(V_1,"Check Claim", _, _)
- InsertAction(V_1,"Retrieve add. Data", _, _)
- InsertUnstr.Conc.Fragment(V_1, _, _, f_D)
- InsertAction(V_1,"Calc. Loss Amount", _, _)
- InsertAction(V_1,"Recalc. Cust. Contr.", _, _)
- InsertAction(V_1,"Pay Out", _, _)
- InsertAction(V_1,"Send Letter", _, _)
- InsertAlternativeFragment(V_1, _, _, f_K)
- InsertAction(V_1,"Call Customer", _, _)
- InsertAction(V_1,"Send Rej. Letter", _, _)
- DeleteAction(V_1,"Close Claim")

Fig. 9. Compound change operations that transfer V_1 into V_2

- **InsertCyclicFragment**(V_1,"Record Claim", Decision, f_A)
- MoveAction(V_1,"Check Claim", _, _)
- InsertAction(V_1,"Retrieve add. Data", _, _)
- **InsertUnstr.Conc.Fragment**(V_1,"Settle Claim", Merge, f_D)
- InsertAction(V_1,"Calc. Loss Amount", _, _)
- InsertAction(V_1,"Recalc. Cust. Contr.", _, _)
- InsertAction(V_1,"Pay Out", _, _)
- InsertAction(V_1,"Send Letter", _, _)
- **InsertAlternativeFragment**(V_1,"Reject Claim", Merge, f_K)
- InsertAction(V_1,"Call Customer", _, _)
- InsertAction(V_1,"Send Rej. Letter", _, _)
- **DeleteAction**(V_1,"Close Claim")

Fig. 10. Compound change operations with position parameters

new position, and *InsertAction*(V_1, *"Retrieve add. Data"*, ...) will insert another action. Note that *Insert* and *Move* operations are still incomplete because the position parameters have not been specified. In general, if the position parameters of an operation are determined, we call this operation applicable. The computation of position parameters will be discussed in the following subsection.

4.2 Computation of Position Parameters

According to our requirements, differences between two versions of a process model should be resolvable in an arbitrary way which depends on the position parameters.

Figure 11 shows a simple example where two actions *"A3"* and *"A4"* have been inserted, leading to *InsertAction(V_1,"A3", _, _)* and *InsertAction(V_1,"A4", _, _)*. In order to ensure that the business user can choose both operations, we compute position parameters to be *InsertAction(V_1,"A3","A1","A2")* and *InsertAction(V_1,"A4","A1","A2")*. If either *"A3"* or *"A4"* were position parameters, this would induce a dependency between them, requiring that one of them is applied before the other one.

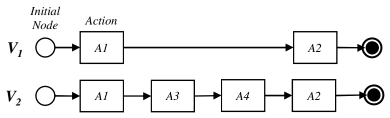


Fig. 11. Simple example

In order to avoid such situations, we express position parameters in terms of *fixpoints* for given process models V_1 and V_2 and a set of correspondences $\mathcal{C}(V_1, V_2)$. A *fixpoint pair* is a pair of nodes $(n_1, n_2) \in \mathcal{C}(V_1, V_2)$ such that n_1 and n_2 are not moved in the process models by any change operation that has been derived from the differences between V_1 and V_2. Given a fixpoint pair (n_1, n_2), both n_1 and n_2 are called fixpoints. For example, in Figure 11, both *("A1","A1")* and *("A2","A2")* are fixpoint pairs. Using fixpoints as position parameters also ensures that the insert and move operations can always produce a model that is connected because the newly inserted or moved element or fragment can be connected to the fixpoints automatically.

Figure 10 shows the operations with position parameters computed (bold printed operations are applicable, others are not yet applicable and need position parameters). After applying an operation, the set of fixpoints increases and the position parameters are recomputed, making more operations applicable.

In the following, we first reason about the completeness of change operations derived using our approach.

4.3 Completeness of Change Operations

The set of change operations containing all compound change operations for two business process models V_1 and V_2 derived according to our approach is denoted by $Changes_{Compound}(V_1, V_2)$. After defining action and fragment differences, one question to ask is whether these are all differences that can occur when changing a process model V_1 into a process model V_2. For this, we assume an ideal minimal change log consisting of change primitives [17] inserting or deleting nodes (Action or ControlNodes) and edges, namely *addActionNode*, *addControlNode*, *addEdge*, *deleteActionNode*, *deleteControlNode*, *deleteEdge*.

Given process models V_1 and V_2, a minimal sequence of primitive change operations op_i converting V_1 into V_2, denoted as $ChangeLog_{min}(V_1, V_2)$, is called a minimal change log. Given such a minimal change log, we have to show that each entry in this change log gives rise to a compound change operation involving actions or fragments, so no entry will be ignored. The following theorem establishes a relationship between the minimal change log and our compound change operations:

Theorem 1 (Completeness of Differences). *Given two business process models V_1, V_2, $ChangeLog_{min}(V_1, V_2)$ and $Changes_{Compound}(V_1, V_2)$, for each $op \in ChangeLog_{min}(V_1, V_2)$ there exists $c \in Changes_{Compound}(V_1, V_2)$ such that c comprises op.*

Proof sketch: Given $op \in ChangeLog_{min}(V_1, V_2)$:

- If $op = addActionNode$, then there exists $y \in \mathcal{C}_{0-1}(V_1, V_2)$ which gives rise to an InsertAction difference which means that we derive an InsertAction operation c comprising *op*.
- If $op = addControlNode$, then the control node either creates a new fragment or is inserted into an existing fragment. The first case induces an InsertFragment difference, the second case a ConvertFragment difference. In both cases, c (InsertFragment or ConvertFragment) comprises *op*.
- If $op = addEdge$, then this involves integrating new nodes (ControlNodes or Actions) into the process model, reordering of existing fragments or nodes, or reconnection of existing nodes in case of deletions. In the first case, there must be a suitable addActionNode or addControlNode operation, leading to appropriate *Insert* operations comprising *op*. In the second case, the addEdge operation (possibly together with other addEdge operations) gives rise to MoveAction or MoveFragment differences comprising *op*. In the third case, there must be a suitable DeleteAction or DeleteFragment operation comprising *op*.
- The cases $op = deleteActionNode$, $op = deleteControlNode$ and *deleteEdge* can be treated analogously.

This result shows that it is possible to detect differences based on actions and fragments. Note that each fragment difference usually involves more than one control node or edge difference and gives rise to the possibility to abstract from several individual differences relating to edge reconnection or control node changes.

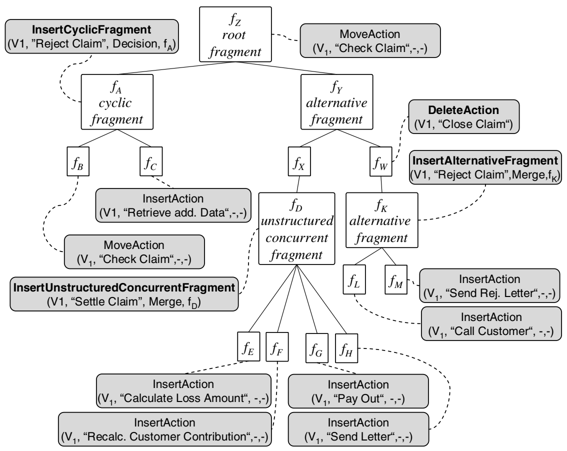
5 Computation of Hierarchical Change Log

In order to enable user-friendly resolution of changes, change operations can be visualized according to the structure of the two process models which is obtained by their SESE decomposition: Given such a fragment decomposition, each operation can be associated to the fragment in which it occurs. In the following, we first introduce a joint process structure tree as a basis of such a hierarchical change log. Given two process structure trees $PST(V_1)$, $PST(V_2)$ and correspondences between their nodes, then the joint PST is denoted as $PST(V_1,V_2)$. The joint PST can be constructed as follows:

- for a pair $(v_1,v_2) \in \mathcal{C}^F(V_1,V_2)$, a new node v_3 is inserted into $PST(V_1,V_2)$ with $\mathit{fragment}(v_3) = \mathit{fragment}(v_1)$.
- for a node $v_1 \in \mathcal{C}^F_{1-0}(V_1,V_2)$, a new node v_3 is inserted into $PST(V_1,V_2)$ with $\mathit{fragment}(v_3) = \mathit{fragment}(v_1)$,
- for a node $v_2 \in \mathcal{C}^F_{0-1}(V_1,V_2)$, a new node v_3 is inserted into $PST(V_1,V_2)$ with $\mathit{fragment}(v_3) = \mathit{fragment}(v_2)$.

Based on the joint PST, we can define a hierarchical change log. The idea of the hierarchical change log is to arrange change operations according to the structure of the process model by associating to each SESE fragment the change operations that affect it (for a formal definition see [10]).

Figure 12 shows a hierarchical change log for the two versions V_1 and V_2 of a process model introduced earlier in this paper. For example, the *InsertCyclicFragment* operation takes place within the root fragment f_Z. The *InsertUnstructuredConcurrentFragment* occurs in the branch f_X of the alternative fragment f_Y. Within the newly inserted unstructured parallel fragment f_D, there are several *InsertAction* operations. Further operations such as the *InsertAlternativeFragment* or *DeleteAction* operations are also associated to their fragments.



f_Z root fragment
InsertCyclicFragment (V1, "Reject Claim", Decision, f_A)
MoveAction (V_1, "Check Claim",-,-)
f_A cyclic fragment
f_Y alternative fragment
DeleteAction (V1, "Close Claim")
f_B
f_C
f_X
f_W
InsertAlternativeFragment (V1, "Reject Claim",Merge,f_K)
InsertAction (V1, "Retrieve add. Data",-,-)
f_D unstructured concurrent fragment
f_K alternative fragment
MoveAction (V_1, "Check Claim",-,-)
InsertUnstructuredConcurrentFragment (V1, "Settle Claim", Merge, f_D)
f_L
f_M
InsertAction (V_1, "Send Rej. Letter",-,-)
InsertAction (V_1, "Call Customer", -,-)
f_E
f_F
f_G
f_H
InsertAction (V_1, "Calculate Loss Amount", -,-)
InsertAction (V_1, "Pay Out", -,-)
InsertAction (V_1, "Recalc. Customer Contribution",-,-)
InsertAction (V_1, "Send Letter",-,-)

Fig. 12. Hierarchical change log of example

Using the hierarchical change log, one can easily identify the areas of the process model that have been manipulated. This enables the business user to concentrate on those changes that are relevant to a certain area in the process model and increases thereby usability. The hierarchical change log can also be beneficial for identifying dependencies between change operations and for identifying groups of change operations that can be applied as a change transaction. In the next section, we elaborate on the application of change operations and tool support.

6 Application of Operations and Tool Support

The operations in the change log with position parameters are ready for application. Figure 13 shows V_1 and the application of the *InsertUnstructuredConcurrentFragment* operation, leading to the insertion of the *Fork* and two *Joins* and the automated reconnection of control flow. Alternatively, the user could have chosen any other operation of the change log that is applicable.

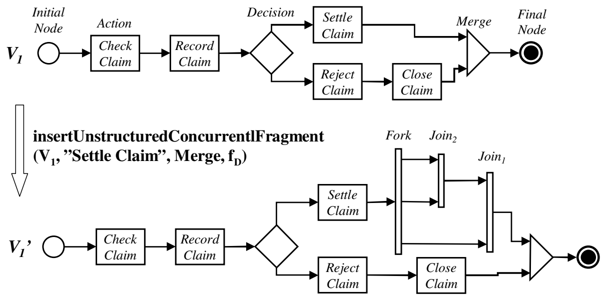


Fig. 13. Applying a compound change operation

By recomputing the position parameters of the remaining operations we can increase the number of applicable operations and refine existing position parameters. In the example, the position parameters of the *InsertAction(V_1,"Calculate Loss Amount", _, _)*, *InsertAction(V_1,"Recalc. Customer Contribution", _, _)*, *InsertAction(V_1,"Pay Out", _, _)*, and *InsertAction(V_1,"Send Letter", _, _)* operations can be computed after inserting the unstructured concurrent fragment. This leads to a new change log which is shown in Figure 14.

As proof of concept, we have implemented a prototype as an extension to the IBM WebSphere Business Modeler [1] (see Fig. 15), including functionality for creation of correspondences when copying a process model, decomposition of process models into SESE fragments and detection and resolution of differences.

Root Fragment
- **InsertCyclicFragment**(V_1,"Record Claim", Decision, f_A)
  - ***Move***(V_1,"Check Claim", _, _)
  - ***InsertAction***(V_1,"Retrieve add. Data", _, _)
- Alternative Fragment
  - Unstructured Concurrent Fragment
    - **InsertAction**(V_1,"Calculate Loss Amount", Fork, Join_2)
    - **InsertAction**(V_1,"Recalc. Cust. Contrib.", Fork, Join_2)
    - **InsertAction**(V_1,"Pay Out", Fork, Join_1)
    - **InsertAction**(V_1,"Send Letter", Join_2, Join_1)
  - **Delete**(V_1,"Close Claim")
  - **InsertAlternativeFragment**(V_1,"Reject Claim", Merge, f_K)
    - ***InsertAction***(V_1,"Call Customer", _, _)
    - ***InsertAction***(V_1,"Send Rej. Letter", _, _)

Fig. 14. Recomputed change log after applying a compound operation

Fig. 15 shows versions V_1 and V_2 of the business process model introduced earlier in this paper. The lower third of Fig. 15 illustrates the Difference View, which is divided into three columns. The left and right hand columns show versions V_1 and V_2 of the process model decomposed into SESE fragments. The middle column of the difference view displays the hierarchical change log as introduced previously. Using our prototype, differences between the two versions can be iteratively resolved using the change operations introduced in this paper.

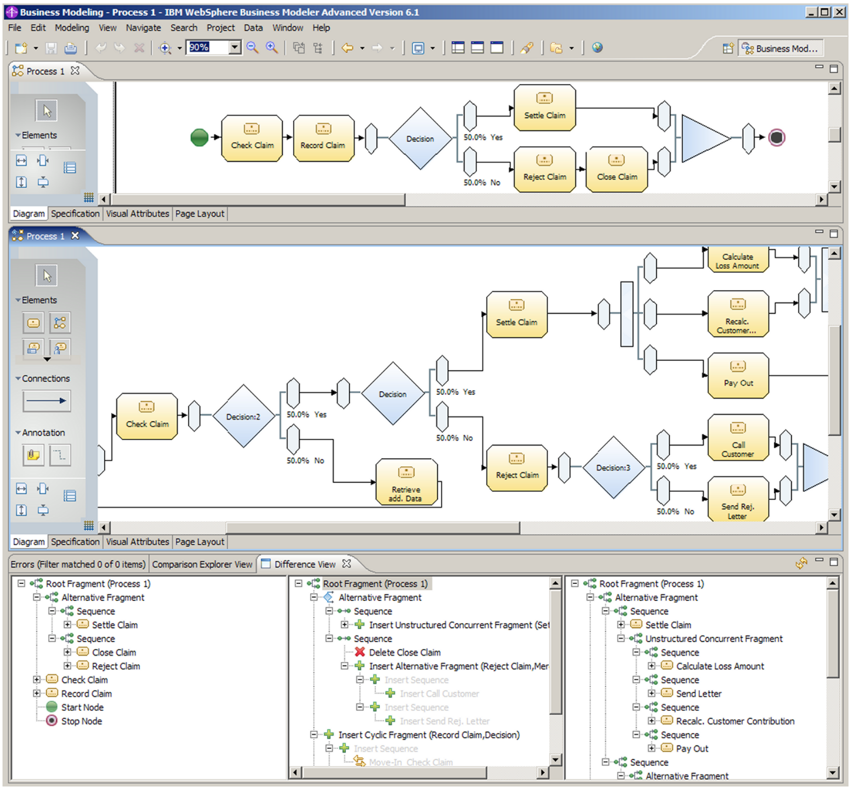


Fig. 15. Business Process Merging Prototype in the IBM WebSphere Business Modeler

As an initial validation, we applied our approach to the IBM Insurance Application Architecture (IAA) [9]. We focused on claim handling, a key process in insurance industry, which is modeled as a composition of several other processes in IAA and is far to large to identify differences without tool support. We introduced a set of differences in the claim handling process and its subprocesses. Using our prototype, users who were unaware of the differences, were able to identify the differences easily and additionally were able to resolve selected differences in order to create a consolidated version of the process model.

7 Related Work

Within the workflow community, the problem of migrating existing workflow instances to a new schema [3,15] has received considerable attention: Given a process schema change, it can be distinguished between process instances that can be migrated and those that cannot be migrated [16]. Rinderle et al. [16] describe migration policies for the situation that both the process instance as well as the process type has been changed. They introduce a selection of change operations and examine when two changes are commutative, disjoint or overlapping. Recent work by Weber et al. [21] provides a comprehensive overview of possible change patterns that can occur when process models or process instances are modified. These change patterns are used for evaluating different process-aware information systems [5] with respect to change support. Zhao and Liu [22] address the problem of versioning for process models and present a means of storing all versions in a version preserving graph, also relying on the existence of a change log. Grossmann et al. [7] show how two business processes can be integrated using model transformations after relationships have been established. Both the change operations by Rinderle et al. [16] and the change patterns for inserting, deleting and moving a process fragment are similar to our change operations. In contrast to the existing work, we describe an approach how to identify change operations in the case that no change log is available and how to use SESE fragments for ordering discovered change operations.

In the context of process integration where models do not originate from a common source model, Dijkman [4] has categorized differences of process models and van Dongen et al. [19] have developed techniques for measuring the similarity of process models. This work is complementary to our work where we assume that correspondences can be automatically derived. In process integration, correspondences must first be established using semantic matching techniques (see e.g. [6]) before our techniques for detecting differences can be applied.

In model-driven engineering, generic approaches for detecting model differences and resolving them have been studied [2,8]. Alanen and Porres [2] present an algorithm that calculates differences of two models based on the assumption of unique identifiers. The computation of differences has similarities to our reconstruction of change operations, however, in our work, we aim at difference resolution for process models with minimal user interaction.

8 Conclusion and Future Work

Detecting and resolving process model differences represents a standard functionality in process-aware information systems that provide a change log. In this paper, we have presented an approach for the detection and resolution of differences in the absence of a change log that is based on correspondences between process models and also makes use of the concept of a SESE fragment decomposition of process models. This SESE decomposition enables the detection of compound changes and the visualization of differences according to the structure of process models. The resolution of differences is performed in an iterative way, by applying change operations that automatically reconnect the control flow.

There are several directions for future work. The change operations are intended to preserve well-formedness and soundness of the process model which needs to be formally proven. Further work is needed to support all features of process models such as data flow. Afterwards, a detailed validation of our approach can be performed to show that dealing with compound changes saves time during modeling. Future work will also include the elaboration of our approach for merging process models in a distributed environment. In those scenarios, the concept of a conflict becomes important because one resolution can turn the other resolution non-applicable.

Acknowledgements. We would like to thank Jana Koehler, Ksenia Ryndina, and Olaf Zimmermann for their valuable feedback on an earlier version of this paper.

References

1. IBM WebSphere Business Modeler, `http://www.ibm.com/software/integration/wbimodeler/`
2. Alanen, M., Porres, I.: Difference and Union of Models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
3. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. Data Knowl. Eng. 24(3), 211–238 (1998)
4. Dijkman, R.: A Classification of Differences between Similar Business Processes. In: EDOC 2007, pp. 37–50. IEEE Computer Society, Los Alamitos (2007)
5. Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Process-Aware Information Systems. Wiley, Chichester (2005)
6. Grigori, D., Corrales, J., Bouzeghoub, M.: Behavioral matchmaking for service retrieval. In: ICWS 2006, pp. 145–152. IEEE Computer Society, Los Alamitos (2006)
7. Grossmann, G., Ren, Y., Schrefl, M., Stumptner, M.: Behavior Based Integration of Composite Business Processes. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 186–204. Springer, Heidelberg (2005)
8. Herrmann, C., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: An Algebraic View on the Semantics of Model Composition. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA 2007. LNCS, vol. 4530. Springer, Heidelberg (2007)
9. IBM Insurance Application Architecture, `http://www.ibm.com/industries/financialservices/iaa`
10. Küster, J.M., Gerth, C., Förster, A., Engels, G.: Process Merging in Business-Driven Development. IBM Research Report RZ 3703, IBM Zurich Research Laboratory (2008)
11. Mitra, T.: Business-driven development. IBM developerWorks article, IBM (2005), `http://www.ibm.com/developerworks/webservices/library/ws-bdd`
12. Object Management Group (OMG). The Unified Modeling Language 2.0 (2005)
13. Pottinger, R., Bernstein, P.A.: Merging Models Based on Given Correspondences. In: VLDB, pp. 826–873 (2003)
14. Puhlmann, F., Weske, M.: Investigations on Soundness Regarding Lazy Activities. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 145–160. Springer, Heidelberg (2006)
15. Reichert, M., Dadam, P.: ADEPT$_{\text{flex}}$-Supporting Dynamic Changes of Workflows Without Losing Control. J. Intell. Inf. Syst. 10(2), 93–129 (1998)
16. Rinderle, S., Reichert, M., Dadam, P.: Disjoint and Overlapping Process Changes: Challenges, Solutions, Applications. In: Meersman, R., Tari, Z. (eds.) OTM 2004. LNCS, vol. 3290, pp. 101–120. Springer, Heidelberg (2004)
17. Rinderle, S., Reichert, M., Jurisch, M., Kreher, U.: On Representing, Purging, and Utilizing Change Logs in Process Management Systems. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 241–256. Springer, Heidelberg (2006)
18. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. Journal of Circuits, Systems, and Computers 8(1), 21–66 (1998)

19. van Dongen, B., Dijkman, R., Mendling, J.: Measuring Similarity between Business Process Models. In: CAiSE 2008, pp. 450–464. Springer, Heidelberg (2008)
20. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 43–55. Springer, Heidelberg (2007)
21. Weber, B., Rinderle, S., Reichert, M.: Change Patterns and Change Support Features in Process-Aware Information Systems. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 574–588. Springer, Heidelberg (2007)
22. Zhao, X., Liu, C.: Version Management in the Business Process Change Context. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 198–213. Springer, Heidelberg (2007)