Structural Matching of Process Models for Change Detection

Masud Fazal-Baqae

Masterarbeit für den Masterstudiengang Informatik

betreut durch:
Dr. Jochen M. Küster, Business Integration Technologies, IBM Research – Zurich
Prof. Dr. Gregor Engels, Fachgebiet Datenbank- und Informationssysteme

vorgelegt bei:
Prof. Dr. Gregor Engels, Fachgebiet Datenbank- und Informationssysteme
Prof. Dr. Wilhelm Schäfer, Fachgebiet Softwaretechnik

Paderborn, 02. November 2009
Sperrvermerk


Erklärung


Rüschlikon (ZH)/Paderborn, 02. November 2009

(Masud Fazal-Baqeie)
Acknowledgements

This master thesis was written at the University of Paderborn in the research group “Datenbank- und Informationssysteme” of Prof. Dr. Gregor Engels in cooperation with IBM Research – Zurich. I would like to express my gratitude for the opportunity to write my thesis on an industrial research topic in the field of Business Process Modeling. I especially thank my supervisor Prof. Dr. Gregor Engels for making me aware of this opportunity and his flexible guidance and support.

At the laboratory of IBM Research – Zurich I was supervised by Dr. Jochen M. Küster. I would like to express my appreciation for his advice and continuous mentoring. I benefited a lot from his great experience and valuable insight and thank him for the constructive discussions and productive feedback.

I enjoyed being part of the Business Integration Technologies group and would like to thank all members for making my stay at Zurich so pleasant and successful. Let me especially thank my manager Dr. Jana Koehler for making my stay possible, Dr. Hagen Völzer for our professional discussions, Viktor Müller for his company in the office and on the driving range and for proofreading this thesis.

I am grateful for the support by my friends. Allow me to mention Darko Jus and Jörg Clobes, whom I want to thank for proofreading and valuable feedback; Anja Speer and Matthias Becker, who kept me focused and motivated by constant encouragement. I am thankful for them being the extraordinary people they are.

Last but foremost, I would like to thank my family for their faith in me and their continuous love, patience, and understanding.
Abstract

In order to keep their competitive advantage, companies are forced to enhance the capabilities and the flexibility of their IT by using Business-driven Development (BDD), a methodology based on the modeling and continuous refinement of business processes. One of the problem areas of BDD is the change detection and comparison of business process models, which is necessary to effectively consolidate and synchronize different versions of models. Differences between process models can be computed upon a matching of corresponding elements and process fragments. Creating such a matching is simple if all corresponding elements carry the same unique identifiers (IDs). However, in practice this is often not the case.

In this thesis a matching algorithm is presented that uses similarity-based matching heuristics to compute a matching between process model elements and fragments in cases of incomplete or missing ID information. The matching heuristics are based on element attributes, connections and containment in process fragments.

The algorithm was implemented and an existing process comparison component was extended with it. An evaluation based on industrial reference process models shows that the approach taken here is capable of matching process models in absence of IDs using solely similarity-based matching heuristics.
Contents

1 Introduction ........................................... 1

2 Foundations ........................................ 5
   2.1 Business-driven Development .................. 5
   2.2 Business Process Modeling Languages .......... 6
       2.2.1 BPMN ........................................ 6
       2.2.2 Workflow Graph .............................. 8
   2.3 Process Decomposition .......................... 12
       2.3.1 Single-Entry-Single-Exit Fragments .......... 14
       2.3.2 Process Structure Tree ...................... 14
       2.3.3 Fragment Properties ......................... 15
   2.4 Matching Terminology ........................... 16
       2.4.1 Correspondence ................................ 16
       2.4.2 Matching ..................................... 17
       2.4.3 Optimal Matching ............................ 18
   2.5 Matching Process Models ......................... 19
       2.5.1 Categorization of Attributes ................. 20
       2.5.2 Matching Approaches ........................ 21
       2.5.3 Change Detection via Change Operations .... 22
   2.6 Summary .......................................... 25

3 Requirements for Business Process Matching ....... 27
   3.1 Model Matching in Business-driven Development . . 27
       3.1.1 Version Control System (Use Case 1) ........ 27
       3.1.2 BPEL-2-BPMN Feedback (Use Case 2) .......... 28
       3.1.3 Reference Model Customization (Use Case 3) ... 29
   3.2 On Optimal Matching for Change Detection ....... 30
       3.2.1 On Defining Concrete Confidence Functions .... 30
       3.2.2 A Concrete Confidence Function for Change Detection ... 32
   3.3 Requirements for Model Matching ............... 32
       3.3.1 High-level Requirements ....................... 33
       3.3.2 Low-level Requirements ....................... 34
       3.3.3 Suitability of Matching Approaches .......... 40
3.4 Discussion of Related Work .................................. 40
  3.4.1 Generic approaches .................................... 41
  3.4.2 Schema Matching ...................................... 42
  3.4.3 Source Code Matching .................................. 43
  3.4.4 Matching of UML Diagrams ............................... 43
  3.4.5 Matching of Process Models ............................. 44
  3.4.6 Overviews on Model Matching and Model Comparison .... 45
  3.4.7 Results of Discussion .................................. 46
3.5 Summary ......................................................... 46

4 Matching Metrics and Algorithms ................................. 47
  4.1 Existing Model Matching Algorithms ......................... 47
    4.1.1 Chawathe-Algorithm .................................. 48
    4.1.2 Similarity flooding .................................. 50
  4.2 Matching Metrics ............................................ 54
    4.2.1 ID-based Similarity .................................. 54
    4.2.2 Name Similarity ..................................... 55
    4.2.3 Contained Correspondences Similarity .................. 56
    4.2.4 Depthsum of Nodes .................................... 58
    4.2.5 Entry- and Exit-Node Similarity ....................... 59
    4.2.6 Common Neighborhood Similarity ....................... 60
    4.2.7 Source- and Target-Node Similarity .................... 65
  4.3 Matching Algorithm ......................................... 66
    4.3.1 ID-based Matching .................................... 68
    4.3.2 Activity Matching .................................... 68
    4.3.3 Preliminary Fragment Matching ......................... 72
    4.3.4 Entry-/Exit-Gateway Matching of Well-structured Fragments .... 73
    4.3.5 Control Node Matching based on Neighborhood .......... 76
    4.3.6 Leaf-Node-based Fragment Matching .................... 78
    4.3.7 Loop Node Pair Matching ............................... 78
    4.3.8 Subprocess Node Pair Matching ........................ 79
    4.3.9 Edge Matching ........................................ 80
    4.3.10 On the Optimality of the Proposed Algorithm ........ 81
    4.3.11 Comparison to Presented Model Matching Algorithms .... 82
  4.4 Summary ......................................................... 82

5 Implementation and Integration ..................................... 83
  5.1 WebSphere Business Modeler ................................ 83
  5.2 Process Compare and Merge .................................. 83
  5.3 Requirements for the Process Matching Component .......... 84
  5.4 Component Design and Architecture .......................... 86
    5.4.1 Matching Framework .................................... 86
    5.4.2 Components .......................................... 87
    5.4.3 Implementation of the Matching Algorithm .............. 88
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDD</td>
<td>Business-driven Development</td>
</tr>
<tr>
<td>BPEL</td>
<td>Business Process Execution Language</td>
</tr>
<tr>
<td>BPM</td>
<td>Business Process Modeling</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Modeling Notation</td>
</tr>
<tr>
<td>CVS</td>
<td>Concurrent Versions System</td>
</tr>
<tr>
<td>EPC</td>
<td>Event-driven Process Chains</td>
</tr>
<tr>
<td>HR</td>
<td>High-level Requirement</td>
</tr>
<tr>
<td>IAA</td>
<td>IBM Insurance Application Architecture</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
</tr>
<tr>
<td>ID</td>
<td>Unique Identifier</td>
</tr>
<tr>
<td>IR</td>
<td>Implementation Requirement</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>LR</td>
<td>Low-level Requirement</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-driven Development</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>PMG</td>
<td>Process Merging Graph</td>
</tr>
<tr>
<td>PST</td>
<td>Process Structure Tree</td>
</tr>
<tr>
<td>RML</td>
<td>Reference Process Model Library</td>
</tr>
<tr>
<td>RPM</td>
<td>Reference Process Model</td>
</tr>
<tr>
<td>SESE fragment</td>
<td>Single-Entry-Single-Exit fragment</td>
</tr>
<tr>
<td>SOA</td>
<td>Service-Oriented Architecture</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>UML2-AD</td>
<td>Unified Modeling Language Version 2.0 - Activity Diagrams</td>
</tr>
<tr>
<td>WFG</td>
<td>Workflow Graph</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Motivation

To succeed in today’s markets, businesses constantly have to adapt to changes in their environment. In response to new market trends and increasing competition, they have to adjust their strategies and business processes on a regular basis. Most processes depend on IT systems which in practice are often inflexible and not agile in terms of adjustment to process changes. This inhibits and slows down organizational changes and adaptation. In order to keep their competitive advantage, companies are forced to enhance the capabilities and the flexibility of their IT. One possible way to achieve this is Business-driven Development (BDD).

BDD provides principles for the development of solutions that directly implement one or more business services [39]. As such, BDD is getting more and more popular, e.g. as a methodology to derive Service-Oriented Architectures (SOA) [7]. In BDD, process models are the central design artifact. They are refined continuously until IT implementations can be derived from them. With every change in a business process of the company, which reflects the changed demands of the business environment, the process models are adapted accordingly. These models are then used to directly derive and implement the necessary changes to the IT.

Although the ideas of BDD are known for several years already, the tool support for the daily work of business analysts and developers is still in an immature stage. This is one of the main reasons why the full potential of this new paradigm cannot be leveraged [26]. One of the problematic areas is the change detection and comparison of business process models. Here the analyst wants to have an overview of the differences between two versions of a process model. There are several use cases where two versions of a model have to be compared. For example, by employing a version control system the user would like to compare his local version of a process model to different historic versions in the repository. Another example is the merger of two companies, for which overlapping processes need to be compared for identifying common and differing parts, before the processes can be merged as well. As a third example the following use case may be helpful: during the refinement of process models, different representations are used on different conceptual levels (e.g. design and implementation level). Changes on
different levels make it necessary to compare and synchronize the models in order to keep them in a consistent state.

Differences between process models can be described via the set of change operations that transform the one model into the other. Here we can differentiate between low-level and high-level operations. While low-level operations describe atomic changes that affect single model elements, high-level changes comprise the atomic operations and describe the change of element groups or whole model regions, e.g., process fragments [58]. Figure 1.1 depicts a simple example in UML 2.0 Activity Diagram (UML2-AD) notation. It shows two process models in which the lower one was derived from the upper one by applying changes. Table 1.1 shows sets of operations that describe these changes. The first set of atomic changes is much bigger than the second set which describes high-level changes. Hence, high-level changes give the user a better overview, as the intended changes by the user usually comprise several atomic operations (e.g., insert an activity between two other activities and connect it to its pre- and successor).

![Figure 1.1: Two differing processes with matching activities in UML-AD notation.](image)

Change operations can directly be derived from a matching between model elements [29]: unmatched elements denote a “delete” or an “insert” operation; matched elements denote a “move” operation or no operation. Additionally, operations for fragments can be computed by the appropriate matching of fragments.

As the first step towards detecting changes a mapping between model elements and fragments has to be established, which is commonly known as matching. The vertical dashed lines in Figure 1.1 denote the matching of activities. The matching of other model elements is left out for illustrational reasons.

Creating a matching between model elements and fragments is not as straightforward as it might seem at first glance. In related work (e.g., [4, 50]) unique identifiers (IDs) are often used to establish a matching between similar model elements. The matching
Table 1.1: Differences in Figure 1.1 expressed with different types of operations.

<table>
<thead>
<tr>
<th>Low-Level Operations</th>
<th>Compound Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete Edge('Initial Node', 'Check Claim')</td>
<td>Move Activity ('Record Claim', 'Initial Node', 'Check Claim')</td>
</tr>
<tr>
<td>Delete Edge('Check Claim', 'Record Claim')</td>
<td></td>
</tr>
<tr>
<td>Delete Edge('Record Claim', 'Claim Accepted?')</td>
<td></td>
</tr>
<tr>
<td>Insert Edge('Initial Node', 'Record Claim')</td>
<td></td>
</tr>
<tr>
<td>Insert Edge('Record Claim', 'Check Claim')</td>
<td></td>
</tr>
<tr>
<td>Insert Edge('Check Claim', 'Claim Accepted?')</td>
<td></td>
</tr>
</tbody>
</table>

- Insert Fragment('Settle Claim', 'Merge')
- Insert Fork ('Fork', 'Settle Claim', 'Pay Out')
- Insert Join ('Join', 'Pay Out', 'Merge')
- Insert Activity ('Pay Out', 'Claim Accepted?', 'Join')
- Insert Activity ('Write Accept. Letter', 'Claim Accepted?', 'Join')
- Insert Edge ('Settle Claim', 'Fork')

In practical terms ID-based matching has two major drawbacks. It is based on the assumption that elements in the two models under consideration have the same ID if they match to each other. For this reason it is assumed that a matching algorithm can be based on matching elements with identical IDs. The first problem is that this is the case only in a limited subset of scenarios, e.g. when models were derived from each other. It cannot be applied when models do not share a common modeling history as it is the case in the earlier mentioned company merger scenario. The second problem is that even in those scenarios where IDs are provided it is difficult to maintain the ID mapping, for example when elements are deleted and recreated at the identical position. Usually, the new element is assigned a new ID and cannot not be matched to the element with the former ID. Figure 1.2 depicts the same example as in Figure 1.1, however now with an ID-based matching. Each ID is depicted next to its respective model element. As before, vertical dashed lines denote a matching for activities, here based on IDs. The activity 'Reject Claim' in the lower model was deleted and recreated later on. Thus, it was assigned a new ID and cannot be paired with the activity 'Reject Claim' in the upper model anymore (compare Figure 1.1). This leads to the detection of the redundant pair of a deletion and an insertion operation for this element.

**Task Definition**

The goal of this thesis is to develop a matching algorithm that on the one hand overcomes the explained problems by using different features, besides IDs, to match process model elements; on the other hand is capable of matching process fragments. The algorithm ought to be applicable in use cases where IDs can be exploited, but also when IDs cannot be used. As part of this thesis a matching component should be implemented which uses the developed algorithm. The implementation should be used to evaluate the matching algorithm for its usability in practice.
Chapter 1. Introduction

Figure 1.2: A matching of model elements based on unique identifiers.

Thesis Structure

This thesis is organized as follows: in Chapter 2 the basic concepts and fundamental terminology for this thesis are introduced. In Chapter 3 we present three use cases, derive the requirements for the matching algorithm, and discuss the related work. In Chapter 4 two algorithms from related work are introduced, different similarity metrics are developed and combined to a matching algorithm which is presented thereafter. In Chapter 5 the implementation of the matching algorithm is discussed, being followed by an evaluation of the approach in Chapter 6. The thesis closes with concluding remarks and a brief review of possible extensions in Chapter 7.
Chapter 2

Foundations

This chapter introduces the terminology and concepts that are crucial for the understanding of this thesis. First, Business-driven Development is presented. Second, business process modeling languages are discussed. Third, we introduce the process decomposition technique used to group related process elements. Fourth, the terminology around matching is explained. To show where and how process model matching is used, the computation of change operations is described at the end.

2.1 Business-driven Development

Business-driven Development is a methodology for developing IT implementations that directly satisfy business requirements [26]. BDD requires that “IT efforts are interlocked with business strategy and requirements through an execution framework that [...] can be executed repeatedly”[39]. Traditionally the IT landscape consisted of siloed IT projects, thus reusing business artifacts was difficult. By applying BDD the IT is modeled according to the processes and process steps that constitute the company. The IT components that fulfill the business requirements are designed as small composable services that implement one or more business processes. This approach eases the mapping between business requirements and their IT implementation. Compared to traditional IT implementations the IT services can be reused and invoked in different order more easily. This makes the IT support for business functions more responsive and more agile in terms of changes.

BDD is a holistic approach to capture business requirements, refine them to models and derive an appropriate IT solution. It can be used as an approach to derive a SOA [7], however, it also covers the continuous modification and adoption of services, both on the business and the IT level.

Since process models are adopted, refined, and changed continuously sufficient tool support is crucial. Especially the detection of changes between models has to be improved to fully leverage the potential of BDD. In Section 3.1 three BDD-related use cases are described for which comparing models is important.
Process models are the central artifact in BDD. In the following section we will introduce a business process modeling language that is used to describe and formalize process models.

## 2.2 Business Process Modeling Languages

Business processes can be modeled using different modeling languages. A modeling language provides the possibility to formulate the flow and order of activities that should be carried out. Figure 2.1 provides an example for a simple process model: the start event (black dot on the left) denotes the start of the execution flow and ‘Check Claim’ is the first activity that is executed, followed by ‘Record Claim’. When ‘Record Claim’ is finished ‘Settle Claim’ or ‘Reject Claim’ are executed depending on whether “Claim accepted?” evaluates to true or not. The process finishes when the taken control flow reaches its end.

![Figure 2.1: A simple business process for claim management in UML-AD notation.](image)

Modeling languages support further constructs to modify the control flow. Many modeling languages also support data flow, which is the formalization of the flow of information in the model. In this thesis data flow is not considered, instead the focus is on control flow only.

Nowadays, there are many business process modeling languages in use, for example Event-driven Process Chains (EPC) [23], UML2-AD [44], or the Business Process Modeling Notation (BPMN) [43]. We will introduce BPMN here, a standard by the Object Management Group (OMG) [1], which is of growing importance. BPMN is a widespread and very expressive modeling language, so that an overview of the main model elements is given in the following section.

### 2.2.1 BPMN

BPMN is a visual modeling language for business process models that is widespread and widely accepted in academia and industry [49]. The current version is 1.2, however, version 2.0 is going to be released soon, introducing a common exchange format and executable semantics.

BPMN is a rich modeling language and supports more constructs than presented here. Figure 2.2 presents the most important ones, which also constitute the set of
constructs actively supported by our matching algorithm. In the following each of these constructs is explained briefly.

![Image of BPMN node types](attachment:figure22.png)

Figure 2.2: The different node types of BPMN.

**Activity** An activity is a step in the business process that has to be accomplished. It can be an (atomic) task or a decomposable activity, called a subprocess or a loop. There exist other activity types that are not described here.

A task is an atomic step in the execution of a business process.

A subprocess can be understood as a small process in itself that is activated when the subprocess is triggered. There are two graphical representations. In the collapsed representation subprocesses hide their internal details; in their expanded representation they show them. Figure 2.3 illustrates this.

Loop activities are iterated if a loop condition is true. As for subprocesses their content is executed whenever they are active (iterated). Again, there is a collapsed and an expanded graphical representation.

**Gateway** Gateways direct the control flow in the model. They are used as points where the control flow splits or where the control flow merges. Different gateway types exist, but here the focus is on the exclusive gateway and the parallel gateway.

Each gateway type can be divided into two groups. For a splitting gateway the control flow is continued on one or more outgoing edges when it is activated. Which outgoing edges are activated depends on the type of the gateway and the defined conditions. A joining gateway activates its outgoing edge when one or more incoming edges are activated, based on the defined conditions and its type.

Exclusive gateways model an XOR-logic. For the case of splitting gateways, exactly one outgoing branch is taken based on the condition. Splitting exclusive gateways resemble the decision nodes of UML2-AD. In Figure 2.4, for example, the lower branch with the activity 'Reject Claim' is taken when the claim is not accepted. Alternatively, the upper branch can be taken. Joining exclusive gateways resemble the merge nodes of UML2-AD. For the case of joining gateways,
the exclusive gateway awaits one incoming branch to activate the outgoing one. In the Figure 2.4 the outgoing edge of the exclusive join is executed after 'Reject Claim' or the upper branch has been executed.

Parallel gateways, in contrast, are similar to fork and join nodes of UML2-AD and model an AND-logic. When splitting, all outgoing branches are activated. When joining, the parallel gateway awaits all incoming branches before it activates the outgoing one. In the Figure 2.4 the activities 'Pay Out' and 'Write Accept. Letter' are executed in parallel, when the upper branch is taken (and the claim is accepted). The parallel join does not execute its outgoing edge before both activities are finished.

**Event** Events mark the start (start event) or the end (end event) of a control flow in a business model.

**Edge** There are two types of edges: control flow and data flow edges. *Control flow edges* connect activities, events and gateways and denote the order of execution flow in the model. *Data flow edges* additionally define which data is sent around between the activities. Since data flow is not part of this thesis, the term 'edge' is used as an abbreviated word for 'control flow edge'.

![Figure 2.3: A collapsed and expanded subprocess in BPMN notation.](image)

Figure 2.4 takes the example from Figure 1.2 of the previous chapter, however, here in BPMN notation. The main difference is that the activity 'Check Claim' is now modeled as a subprocess instead of a simple task.

### 2.2.2 Workflow Graph

Most process modeling languages share a common set of modeling constructs. Even if the semantic meaning might not always be exactly the same, one can abstract from
the concrete languages and define a generalization of a concrete process model that is formalized in a specific language.

To define the generalization, the definition of a workflow graph is given (see Definition 1), which is an extension of existing definitions in literature (e.g. [57, 56]). We extended it to allow for multiple start and termination nodes and moreover to include representations of loops and subprocesses. Using the workflow graph we can define the matching algorithm on a more abstract level such that it can be applied to different process modeling languages. Additionally, the use of workflow graphs allows the application of the decomposition technique presented in Section 2.3.

**Definition 1 (Workflow Graph).** A workflow graph is a directed graph $WFG = (N, E)$ where the nodes are partitioned into the sets $N = \{S, T, A, G, L, P\}$ with $S$ being start nodes, $T$ being termination nodes, $A$ being activity nodes, $G$ being gateway nodes, $L$ being loop nodes, and $P$ being subprocess nodes.

For the gateway nodes we define a mapping $t_G$ with $t_G : G \rightarrow \{\text{decision, fork, merge, join}\}$. For the loop nodes and subprocess nodes we define a mapping $t_{L,P}$ with $t_{L,P} : L \cup P \rightarrow \{\text{opening, closing}\}$.

In this thesis we will, furthermore, use the term control nodes as a synonym for the set of start, termination and gateway nodes. Figure 2.5 shows the different node types.

In the following we define well-formed workflow graphs, which are workflow graphs with additional constraints (Definition 2).

**Definition 2 (Well-Formed Workflow Graph).** A well-formed workflow graph is a workflow graph $WFG = (\{S, T, A, G, L, P\}, E)$ that fulfills the following conditions:

1. $S \neq \emptyset$ and $T \neq \emptyset$ (at least one start and one termination node per process)
2. $\forall s \in S : |\{(s, x) : (s, x) \in E\}| = 1 \land |\{(x, s) : (x, s) \in E\}| \leq 1$ (connections of start node)
Chapter 2. Foundations

![Workflow Graph Diagram]

Figure 2.5: The different node types of a workflow graph.

3. \( \forall t \in T : |\{(t,x) : (t,x) \in E\}| \leq 1 \ \land \ |\{(x,t) : (x,t) \in E\}| = 1 \) (connections of termination node)

4. \( \forall a \in A : |\{(a,x) : (a,x) \in E\}| = 1 \ \land \ |\{(x,a) : (x,a) \in E\}| = 1 \) (connections of activity node)

5. \( \forall g \in G, t_G(g) \in \{\text{decision, fork}\} : |\{(g,x) : (g,x) \in E\}| \geq 2 \ \land \ |\{(g,t) : (g,t) \in E\}| = 1 \) (connections of decision and fork nodes)

6. \( \forall g \in G, t_G(g) \in \{\text{merge, join}\} : |\{(g,x) : (g,x) \in E\}| = 1 \ \land \ |\{(g,t) : (g,t) \in E\}| \geq 2 \) (connections of merge and join nodes)

7. \( \forall o \in L \cup P, t_{L,P}(o) \in \{\text{opening}\} : |\{(o,x) : (o,x) \in E\}| \geq 1 \ \land \ |\{(o,t) : (o,t) \in E\}| \geq 1 \) (connections of opening loop and subprocess nodes)

8. \( \forall c \in L \cup P, t_{L,P}(c) \in \{\text{closing}\} : |\{(c,x) : (c,x) \in E\}| \geq 1 \ \land \ |\{(c,t) : (c,t) \in E\}| \geq 1 \) (connections of closing loop and subprocess nodes)

9. Let \( M(x) \) be a bijective mapping function \( M : L \cup P \rightarrow L \cup P \) s.t.:

   (constraints for loops and subprocesses)
   
   - \( \forall x, y \in L \cup P : M(x) = M(y) \Rightarrow x = y \) (\( M(x) \) is injective)
   - \( \forall x \in L \cup P : \exists y \in L \cup P : M(x) = y \) (\( M(x) \) is surjective)
   - \( \forall x \in L \cup P : x \in P \Leftrightarrow M(x) \in P \) (both subprocess nodes)
   - \( \forall x \in L \cup P : x \in L \Leftrightarrow M(x) \in L \) (both loop nodes)
   - \( \forall x \in L \cup P : t_{L,P}(x) = \{\text{opening}\} \land t_{L,P}(x) \neq t_{L,P}(M(x)) \) (opening mapped to closed)

Then \( \exists M(x) : \forall o \in L \cup P : \exists S \subset N : S \neq \emptyset \)

\( \land \forall (s,t) \in E : s \in S \Leftrightarrow t \in S \cup \{M(o)\} \) (isolated subgraph)

\( \land \exists (o,g) \in E : g \cap S \neq \emptyset \) (incoming connection)
10. Let \( p = (u_0, v_0) \ldots (u_m, v_m) \) with \( (u_i, v_i) \in E, m \in \mathbb{N} \) and \( v_{i-1} = u_i \forall i \in 1, \ldots, m \) denote a path in the workflow graph. Then \( \forall x \in N \setminus (S \cup T) \exists s \in S, \exists t \in T : \exists p \) with \( u_0 = s, v_i = x, v_m = t, i \in \{1 \ldots m - 1\} \) (each node on path from a source to a target)

The first constraint (1) ensures that the workflow graph has at least one start and one termination node.

Furthermore, each start node has exactly one outgoing edge and at most one incoming edge (2), whereas each termination node has exactly one outgoing edge and at most one incoming edge (3). All activity nodes have at exactly one incoming and one outgoing edge (4). The decision and fork nodes have exactly one incoming edge and two or more outgoing edges (5), whereas merge and join nodes have two or more incoming edges and exactly one outgoing edge (6). The (7)-th constraint ensures that every opening loop or subprocess node has at least one outgoing edge and at least one incoming edge. Every closing loop or subprocess node also has at least one outgoing edge and at least one incoming edge (8).

The (9)-th constraint assures that pairs of loop and subprocess nodes are building encapsulating building blocks for other nodes. In Figure 2.6 these building blocks are depicted by boxes with dashed lines. The definition says that there exists a bijective mapping function that assigns an opening loop node to every closing loop node; and an opening subprocess node to every closing subprocess node. In this way opening and closing loop/subprocess nodes are paired with each other. In addition, there exists a non-empty subgraph \( S \) (in the figure denoted by the dashed rectangles) between both nodes of this pair \( P \), which fulfills the following contraints: Every node in \( S \) has outgoing edges only to other nodes in \( S \) and/or the closing node of \( P \) and it has incoming edges only from other nodes in \( S \) and/or the opening node of \( P \). There exists at least one edge from the opening node of \( P \) to a node in \( S \) and at least one edge from the closing node of \( P \) to a node in \( S \). We say that the elements in \( S \) are enclosed by the opening and closing nodes of \( P \). In this example the non-empty subgraph of the subprocess 'Check Claim' is denoted by all the nodes enclosed by the opening and closing subprocess nodes. This also includes the loop 'Collect addit. Information', which is enclosed by the opening and closing loop nodes. The last condition of the constraint 9 assures that all node pairs of opening and closing nodes are nested properly or completely independent. This condition avoids the situation shown in Figure 2.7 where the subprocess and loop are overlapping.

The last condition (10) ensures that the graph is connected and each node is on a path from a start to a termination node.

Only well-formed workflow graphs are considered in this thesis, thus, whenever the term 'workflow graph' is used, it refers to 'well-formed workflow graph'.
Relationship between BPMN and WFG

There exists a mapping between the presented model elements of (well-formed) BPMN models and WFGs. The mapping of other BPMN model elements is omitted, since they are out of scope of this work. Table 2.1 shows which elements correspond to each other. Mostly there exists a one-to-one mapping. The only elements that cannot be translated in a straightforward manner are loops and subprocesses. In BPMN these are compound nodes that directly contain other elements. In the workflow graph a node cannot contain other nodes. Instead, the contained nodes of a BPMN subprocess or loop are enclosed by (edges to the) opening and closing nodes of the respective type (loop or subprocess). Figure 2.8 shows the WFG abstraction of the BPMN example in the previous section (Figure 2.4). The subprocess element is flattened by enclosing its content with the two subprocess nodes. Furthermore, the start node of the subprocess in the WFG, which corresponds to the start event in the BPMN model, is connected to the opening subprocess node. Similarly, the termination node of the subprocess is connected to the closing subprocess node.

2.3 Process Decomposition

To determine and match groups of coherent model elements, these groups have to be identified and computed. One possibility is to decompose the process into its so-called Single-Entry-Single-Exit (SESE) fragments and to organize these fragments in a tree hierarchy. The details are given in the following.
Table 2.1: BPMN model elements and their workflow graph equivalent.

<table>
<thead>
<tr>
<th>BPMN</th>
<th>WFG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>Activity Node</td>
</tr>
<tr>
<td>Subprocess</td>
<td>Opening Subprocess, Closing Subprocess</td>
</tr>
<tr>
<td>Loop</td>
<td>Opening Loop Node, Closing Loop Node</td>
</tr>
<tr>
<td>Exclusive Split</td>
<td>Decision Node</td>
</tr>
<tr>
<td>Exclusive Join</td>
<td>Merge Node</td>
</tr>
<tr>
<td>Parallel Split</td>
<td>Fork Node</td>
</tr>
<tr>
<td>Parallel Join</td>
<td>Join Node</td>
</tr>
<tr>
<td>Start Event</td>
<td>Start Node</td>
</tr>
<tr>
<td>End Event</td>
<td>Termination Node</td>
</tr>
</tbody>
</table>

Figure 2.8: A workflow graph of a business process for claim management that contains a subprocess.
2.3.1 Single-Entry-Single-Exit Fragments

A SESE fragment (fragment for short) is a non-empty subgraph of the workflow graph that has a single edge that enters the subgraph from outside and a single edge that exits the subgraph [57]. These edges are called single-entry-edge and single-exit-edge, respectively. Figure 2.9 shows a workflow graph and its SESE fragments. The single-entry-edge of fragment $f_Y$, for example, is the edge between the activity 'Check Claim' and the decision node and the single-exit-edge is the edge between the merge and termination nodes. We say that the fragment $f_Y$ contains other elements. In this case it contains for example the activities 'Pay Out' and 'Reject Claim'.

The SESE decomposition technique is known from compiler theory and can be performed in linear time [22]. A more formal definition of a SESE fragment is given in Definition 3.

**Definition 3 (SESE Fragment [57]).** Let $G = (N, E)$ be a workflow graph. A SESE fragment $F = (N', E')$ is a nonempty subgraph of $G$, i.e., $N' \subseteq N$ and $E' = E \cap (N' \times N')$ such there exist edges $e, e' \in E$ with $E \in ((N \setminus N') \times N') = \{e\}$ and $E \in (N' \times (N \setminus N')) = \{e'\}$; $e$ and $e'$ are called the entry-edge and the exit-edge of $F$, respectively.

2.3.2 Process Structure Tree

Figure 2.10 presents an example for a process decomposition into its canonical [57] SESE fragments. A process structure tree (PST) $PST(WFG)$ is a unique decomposition of a workflow graph $WFG$ into its canonical SESE fragments which are either disjoint or fully nested in each other. Like its SESE fragments the PST can be computed in linear time. It is unique and modular, i.e. a local change of the workflow graph causes only a local change of the decomposition [22]. In the following only fragments are discussed that are part of a PST, which is a decomposition of canonical fragments. Thus, from now on the term 'fragment' is used as a abbreviation of the term 'canonical fragment'.
2.3.3 Fragment Properties

Entry-/Exit-Nodes

Every fragment has one unique incoming and one unique outgoing edge. The target of the incoming edge is the entry-node of the fragment. The source of the outgoing edge is the exit-node of the fragment (as defined in Definition 4). For example, the entry-node for the fragment \( f_Y \) in Figure 2.9 is the decision node and its exit-node is the merge node.

**Definition 4** (Entry-/Exit-Node of a Fragment). Let \( F \) be a fragment of a workflow graph. Let \( e = (s_e, t_e) \) be the entry-edge of the fragment. Then the entry-node of the fragment \( F \) is the node \( t_e \). Let \( ex = (s_{ex}, t_{ex}) \) be the exit-edge. Then similarly the exit-node of the fragment is the node \( s_{ex} \).

Well-Structured Fragments

In industrial process models gateways in one model can often be paired with each other such that they fully enclose model regions (evaluated in Section 6.1). In such cases the incoming edge of the decision/fork node is the only incoming edge of a whole region; the outgoing edge of the merge/join is the only outgoing edge of that region. As an example consider the decision/merge pair of fragment \( f_Y \) in Figure 2.9. The described gateway pairs form SESE regions and are the entry- and exit-nodes of the respective fragments in the PST. When these fragments do not contain other gateways as direct child nodes they fully encapsulate other elements in a nicely structured way and can be called well-structured. The fragments \( f_Y \) and \( f_X \) in Figure 2.9 are examples for well-structured fragments. A fragment that does not contain any gateways is well-structured as such.
Fragments like $f_Y$ in Figure 2.11 are not well-structured and thus called unstructured. Based on [57] we define well-structured fragments in Definition 5.

**Definition 5** (Well-Structured Fragment [57]). Let $F$ be a fragment of a workflow graph. Then $F$ is well-structured if it satisfies one of the following conditions:

- $F$ has no decisions, merges, forks, joins as direct children in the process structure tree.

- $F$ has exactly one decision and exactly one merge, but no forks and joins as direct children. The entry-edge of $F$ is the incoming edge of the decision, and the exit-edge of $F$ is the outgoing edge of the merge.

- $F$ has exactly one decision and exactly one merge, but no forks and joins as direct children. The entry-edge of $F$ is the incoming edge of the merge, and the exit-edge of $F$ is the outgoing edge of the decision.

- $F$ has exactly one fork, exactly one join, but no decisions and merges as direct children. The entry-edge is the incoming edge of the fork. The exit-edge is the outgoing edge of the join.

2.4 Matching Terminology

For analyzing the relationship between business process models it is important to know which elements in one business process correspond to which elements in the other. This is also known as a matching of elements [35, 48]. In this section we discuss the terminology around matching that is used in this thesis.

2.4.1 Correspondence

A pair consisting of a model element (or SESE fragment) from the one model and its counterpart in the other model is called a correspondence (compare [35, 47]).
Figure 2.12 presents two workflow graphs where corresponding model elements are denoted by dashed lines. For example, activities ‘Check Claim’ in the upper graph and ‘Check Claim’ in the lower graph correspond to each other. Correspondences for other elements than activities are denoted by light grey lines (e.g. between the upper and lower termination nodes). The pair of elements in a correspondence needs to share a set of properties which can be a subset of their syntactical information [33].

A correspondence is a mapping between two corresponding elements from both workflow graphs (or process structure trees) which are conceptually the same entities in the two compared systems, even after renaming or moving [59].

Figure 2.12: Two workflow graphs with correspondences between some of their elements.

Correspondences are formally defined in Definition 6. If two elements $me_1$ and $me_2$ are elements in the same correspondence, then they correspond to each other or in other words, that they are corresponding. If there exists a correspondence for an element $me_1$ such that the element corresponds to at least one other element, then $me_1$ is matched.

**Definition 6** (Correspondence of Model Elements or Model Fragments). A correspondence $C = (me_1, me_2)$ for two workflow graphs $WFG_1 = (N_1, E_1)$ and $WFG_2 = (N_2, E_2)$ or two process structure trees $PST(WFG_1)$ and $PST(WFG_2)$ with the set of fragments $F_1$ and $F_2$ is a 2-tuple for model elements or fragments with either $me_1 \in N_1, me_2 \in N_2$ or $me_1 \in E_1, me_2 \in E_2$ or $me_1 \in F_1, me_2 \in F_2$.

2.4.2 Matching

A matching is a mapping between elements of two workflow graphs (or fragments of two PSTs) that correspond to each other (compare [48]). It can be expressed by a set of correspondences as defined by Definition 7.
Definition 7 (Matching of Model Elements or Model Fragments). Let $WFG_1 = (N_1, E_1)$ and $WFG_2 = (N_2, E_2)$ be two workflow graphs and let $PST(WFG_1)$ and $PST(WFG_2)$ the process structure trees of $WFG_1$ and $WFG_2$ with the set of fragments $F_1$ and $F_2$. Let $(me_i, me'_i)$, $(me_j, me'_j)$ denote correspondences between two elements from $WFG_1$ and $WFG_2$ or two fragments from $F_1$ and $F_2$: $me_i \in N_1, me'_i \in N_2$ or $me_i \in E_1, me'_i \in E_2$ or $me_i \in F_1, me'_i \in F_2$. For a correspondence $(me_i, me'_i) \in M$ we say $me_i$ is mapped to $me'_i$.

Then a matching $M \subseteq (N_1 \times N_2) \cup (E_1 \times E_2) \cup (F_1 \times F_2)$ for $WFG_1$ and $WFG_2$ is a set of correspondences with each element in $WFG_1$ and $WFG_2$ and $PST(WFG_1)$ and $PST(WFG_2)$ being mapped at most once: $\forall (me_i, me'_i), (me_j, me'_j) \in M : me_i = me_j \Leftrightarrow me'_i = me'_j$. We define $\mathcal{M}$ as the set of all possible matchings $M$ for two workflow graphs.

A complete matching $M$ for two workflow graphs $WFG_1$ and $WFG_2$ is a matching that cannot be extended with further correspondences without violating the property that every element is matched at most once. If a matching is not a complete matching then it is a partial matching, especially when it only contains correspondences for certain element types.

When used as a verb, matching refers to the process of comparing model elements or model fragments and creating correspondences.

2.4.3 Optimal Matching

An optimal matching is a matching that cannot be improved by adding, deleting, or replacing the contained correspondences without violating the property that every element is matched at most once or without violating other requirements. There is no general definition for the optimal matching in related literature. For this reason we introduce our own definition of an optimal matching. We describe the optimal matching in two steps. First, we define a function that expresses the quality of a correspondence between two model elements which we call the confidence-function. Then we define the optimal matching using this function.

Confidence for Model Elements or Model Fragments

The degree of confidence for two model elements or model fragments $m_1$ and $m_2$ of two models $M_1$ and $M_2$ is the range of certainty with which they represent the same element. It is expressed in a quantitative way by a number $n \in [-1, 1]$. The lower the confidence value is, the more different the two elements are. Similarly, the higher the confidence value is, the more identical the two elements are. The formal definition is given in Definition 8. A concrete confidence function remains to be defined.

Definition 8 (Confidence of two Model Elements or Fragments). The confidence $\text{conf}(me_1, me_2)$ with $\text{conf} : (N \cup E \cup F) \times (N \cup E \cup F) \rightarrow [-1, 1]$ for elements or fragments $me_1 \in (N_1 \cup E_1 \cup F_1), me_2 \in (N_2 \cup E_2 \cup F_2)$ from two workflow graphs $WFG_1 = (N_1, E_1), WFG_2 = (N_2, E_2)$ or the set of fragments $F_1$ and $F_2$ from
Chapter 2. Foundations

\[ \text{PST}(WFG_1) \text{ and } \text{PST}(WFG_2) \text{ is a function that maps a 2-tuple of model elements or fragments to a value in } [-1,1]. \text{ The value represents the certainty with which } me_1 \text{ and } me_2 \text{ represent the same element.} \]

The concept of using a function to determine the quality of a result is well-known in related fields. In genetic algorithms [17] fitness functions are used to quantify the optimality of the proposed solution computed by an iteration of the algorithm. In speech recognition confidence scores are used to correct recognition errors. The recognized string is replaced with the second-best hypothesis when the confidence score for it is higher than for the recognized string based on the current context [53]. The confidence score is based for instance on the statistical likelihood of the occurrence of a word.

**Optimal Matching**

In Definition 9 we first define how two matchings can be compared with respect to a confidence function.

**Definition 9 (\(\leq_{\text{conf}}\), Comparison Operator for Matchings).** Let \(M_1\) and \(M_2\) be two matchings for a pair of workflow graphs \(WFG_1\) and \(WFG_2\) and their process structure trees PST(\(WFG_1\)) and PST(\(WFG_2\)). Let \((me_i, me_j)\) be a correspondence and \(conf\) be a confidence function. Then the comparison operator \(\leq_{\text{conf}}\) is defined as follows:

\[
M_1 \leq_{\text{conf}} M_2 \iff \sum_{(me_1, me_2) \in M_1} conf(me_1, me_2) \leq \sum_{(me'_1, me'_2) \in M_2} conf(me'_1, me'_2)
\]

We call \(\sum_{(me_i, me_j) \in M} conf(me_i, me_j)\) the overall confidence of the matching \(M\). Using the comparison operator \(\leq_{\text{conf}}\) for matchings we define an optimal matching with respect to a confidence function in Definition 10.

**Definition 10 (Optimal Matching).** Let \(\mathcal{M}\) be the set of all matchings for two workflow graphs \(WFG_1\) and \(WFG_2\) and their process structure trees PST(\(WFG_1\)) and PST(\(WFG_2\)). \(M \in \mathcal{M}\) is an optimal matching with respect to the confidence function \(conf\) if and only if: \(\forall M' \in \mathcal{M}: M' \leq_{\text{conf}} M\).

The definition of an optimal matching depends on a confidence function. In order to compare matchings for two workflow graphs, e.g. for an evaluation, a concrete confidence function \(conf\) has to be defined (see Section 3.2 and Definition 11).

### 2.5 Matching Process Models

In the previous section we introduced the more general definitions around the matching of process models and workflow graphs. This section focuses on the terminology more specific to matching approaches and algorithms and their evaluation.
2.5.1 Categorization of Attributes

In general, model elements and fragments have many attributes that can be utilized to establish a matching. To categorize the attributes Selonen et al. [51] differentiate between **basic attributes** and **non-basic attributes**. Whether only basic or additionally non-basic attributes are used has implications for the matching process:

**Basic Attributes** These are the properties of an element $me$ that can be directly evaluated by looking at the element $me$ itself. Examples for basic attributes are the type of an element, the number of its relations to other elements or some property owned by it, e.g. its name [59].

**Non-Basic Attributes** These are properties of an element that cannot be evaluated independently of other elements in the same model, e.g. concrete relationships.

For example, the edge similarity [33] for two nodes $n \in M, n' \in M'$ from two different models $M, M'$ denotes the edges that $n$ and $n'$ have in common (corresponding edges). Two edges correspond if they represent the same edge in both models: their sources $n$ and $n'$ and their targets $t, t'$ correspond to each other. To decide whether a node $n$ has a common edge with a node $n'$, the targets (or sources) of these edges have to be checked for a correspondence. Figure 2.13 illustrates this problem. Non-basic attributes are here for example the source nodes of the incoming edges to $n$ or $n'$. Thus, the edge similarity cannot be computed by checking $n$ and $n'$ only. Another example for a non-basic attribute is the position of an element in the process with respect to its predecessors or successors.

![Figure 2.13: Two activities $n$ and $n'$ and directly connected corresponding activities.](image)

The main difference between basic and non-basic attributes is that the matching of two elements based on basic attributes does not depend on the matching of other elements. Using non-basic attributes, in contrast, can cause dependencies to the matching of other elements. As a consequence, their use can introduce cyclic dependencies in the matching process.
2.5.2 Matching Approaches

There are different approaches to model matching in terms of what is taken into account when correspondences are established. We differentiate three different types [27]:

**static identity-based** The matching is based on the unique identifiers (ID) of model elements. These have to be equal in both models or a precomputed mapping for the identifiers has to exist. While the actual matching can be performed in a straightforward manner (see example in Section 1) this approach also has its limitations. First of all, it is often not the case that two matching model elements have an identical unique identifier (e.g. recall the company merger example in Section 1). Furthermore, even if they have such an ID then the user is usually not aware of it [15]. This can lead to confusion when for example model elements are moved and used in a completely different context or when elements are deleted and recreated and assigned a different ID. In Figure 2.14 activities A, C, and D can be matched using their IDs. The activity B has no counterpart in the other model, and cannot be matched as no other element has the same ID.

![Figure 2.14: Two workflow graphs matched using a static identity-based approach.](image)

**signature-based** Here the similarity between model elements is not determined by a static identifier, but by the combination of different basic attributes of the element. These are used to compute unique hash values, allowing to match the elements in the same way as with IDs. Because the hash values are determined by a combination of basic attributes of the elements, the use and semantic meaning of these is captured better than in the ID-based approach. However, for many modeling domains it is difficult to define a hash function that differentiates the model elements sufficiently enough, especially if they are characterized more by their use and relationships than by their basic attributes. In Figure 2.15 the same signature value is computed for the two activities A from the upper and lower models. The signature is computed using the name, transaction costs, and other attributes of the activities. Activities C and D are matched likewise, however, the activity B

![Figure 2.15: Same signature value computed for activities A.](image)
from the upper model has no counterpart in the lower model that has the same signature and cannot be matched.

Figure 2.15: Two workflow graphs matched using a signature-based approach.

**similarity-based** Similarity-based matching approaches use similarity functions to compute quantitative scores for element pairs from the two processes. These similarity metrics try to capture the degree of similarity of two elements. The scores are based on the comparison of basic and non-basic attributes of the two elements. Whether two elements are considered to be equal relies on a minimum bound, the **threshold value**. Different strategies can be applied to create a matching consisting of the best correspondences above the threshold in terms of the similarity score of their elements. Figure 2.16 shows an example. We assume that the threshold value is set to 0.6. Starting with the pair of highest similarity the activities A, C, and D are matched to their counterparts. Although the similarity of activity B to activity C is higher than 0.6 they cannot be matched because C has already been matched to activity C in the other model.

2.5.3 Change Detection via Change Operations

In this section we will introduce and describe how changes between two process models are detected and resolved as presented by Küster et al. in [29]. The change detection algorithm consists of two major steps: First, an ID-based matching is established. Second, change operations are computed based on that matching. In this thesis we want to replace the static identity-based matching step with a general matching that supports additional scenarios in which ID-based matching is not possible.
Motivation

Business-driven development requires the construction of process models at different abstraction levels and by different people, thus, there is the demand for consolidating different versions of process models. Küster et al. provide a first solution to this problem by detecting and resolving changes between the models using change operations. Their approach does not rely on a change log, but derives the change operations from a matching, which has to be computed beforehand.

Change Operations

Instead of using atomic operations for every model element, changes are grouped into compound operations that are more meaningful. We use Figure 2.17, which shows two differing workflow graphs, to explain this. One difference is the activity ’Pay Out’ that is present only in the lower graph. Instead of the atomic operations \texttt{insertActivity(’Pay Out’)}, \texttt{insertEdge(fork, ’Pay Out’)}, and \texttt{insertEdge(’Pay Out’, join)} the compound operation \texttt{insertActivity(’Pay Out’, fork, join)} is generated. The supported operation types presented in [29] are \texttt{insertActivity}, \texttt{deleteActivity} and \texttt{moveActivity}. Additional operations that were introduced later also cover control nodes, loops, subprocesses and single edges. Moreover, Küster et al. also compute operations for changes in the fragment structure of a PST (again \texttt{insert}, \texttt{delete} and \texttt{move operations}). These additional fragment operations comprise other compound operations such that operations that affect elements in a single fragment are grouped as such. For example, for the changes in Figure 2.17 the operations of inserting the elements \texttt{fork}, \texttt{join}, \texttt{’Pay Out’} and \texttt{’Write Accept. Letter’} are not shown independently, but grouped under the insert operation for the fragment $f_r$. Küster et al. compute the hierarchy of changes as illustrated by Figure 2.18.
Figure 2.17: Two workflow graphs with correspondences between their elements. The highlighted elements are affected by an operation.

Figure 2.18: A process structure tree with annotated operations (compare [29]).
Appropriate Matching

In the remainder of this chapter, we describe the matching properties that are necessary to derive the appropriate change operations to convert one model into the other. The detection of operations depends directly on the contained correspondences in the matching $M$. Thus, it is important to explain the necessary properties of the correspondences for computing the appropriate operations for the affected model element(s). For two workflow graphs $WFG_1$ and $WFG_2$ the following is required for the matching $M$:

- $(me_1, me_2) \in M$ for elements (element pairs) $me_1 \in WFG_1, me_2 \in WFG_2$ that were moved or that are unchanged.
- $(me_1, x) \notin M$ for elements $me_1 \in WFG_1$ that were deleted, $x \in WFG_2$.
- $(y, me_2) \notin M$ for elements $me_2 \in WFG_2$ that were inserted, $y \in WFG_1$.

The matching of the two workflow graphs should contain only correspondences for elements that were not changed or moved. All deleted and inserted elements should not be part of a correspondence of $M$. Consequently, for two identical workflow graphs correspondences have to be created for all model elements. Furthermore, the right elements have to be matched with each other; otherwise move operations will be computed which move the element to the position of the corresponding element.

2.6 Summary

In this chapter we presented how process models are defined and how they can be decomposed into PSTs. We introduced the terminology around matching and described how a matching relates to the detection of changes. In the following chapter we introduce different use cases where change detection plays an important role and define the requirements for matching workflow graphs.
Chapter 2. Foundations
Chapter 3

Requirements for Business Process Matching

In this chapter we first discuss different use cases from the area of Business-driven Development that have some importance to Business Process Matching. Thereafter, we define requirements for a matching algorithm that is suitable for these scenarios.

3.1 Model Matching in Business-driven Development

In the following, we present three different use cases for the matching of business process models that are relevant from the Business-driven Development perspective.

3.1.1 Version Control System (Use Case 1)

A version control system supports the user with the management of changes that were applied to a model over time. It helps him to keep track of model versions and differences between them. Such systems have a long history in the field of source code management [54, 18, 46] and were introduced recently to manage models (e.g. [41]). One possibility to present the differences is the use of change operations as suggested in [29].

In this use case subsequent model versions are compared with each other. For this reason the ID information of model elements can be used for the matching. Using IDs for matching and, consequently, change detection allows to match elements that a similarity-based approach has difficulties to match (e.g. renamed and moved activities). Nevertheless, static identity-based approaches also have their limitations. When elements are deleted and recreated - even if at the same position - they are usually assigned a new ID and cannot be matched anymore.

As all elements that are unchanged carry the same ID, static identity-based matching can be applied to them. However, as recreated elements do not carry the same ID, this approach has to be extended with signature- or similarity-based approaches. Figure 3.1 illustrates this. It shows two workflow graphs that were derived from each other. Many of the depicted model elements have a unique ID that can be used for matching (differing
IDs in the lower model are left out for illustrational reasons). However, the start node and the activity ‘D’ cannot be matched using static identity-based matching.

![Workflow graph](image)

Figure 3.1: A workflow graph from a repository and the same model with local changes. Most unique identifiers are preserved.

### 3.1.2 BPEL-2-BPMN Feedback (Use Case 2)

In Business-driven Development and Model-driven Development [42] in general, round-trip engineering is an important aspect. It denotes the ability to automatically maintain the consistency between multiple, changing software artifacts [52]. Changes in the one model can be propagated into the other, minimizing the effort to keep models on different abstraction levels or with different views consistent.

In the case of Business-driven Development, at one point in time BPMN models are transformed to BPEL [40] models that can be deployed on an execution runtime [39]. Usually these BPEL models are refined or changed such that it is desirable to feed the changes back to the BPMN level [26]. This can be done with a two-step approach. First, the BPEL model is exported back to a BPMN model. Afterwards, this exported model is compared to the original BPMN model using a change detection algorithm.

While it is possible to preserve a mapping between model element IDs for activities in both representations, it is more difficult to preserve IDs for other model elements. This is due to the fact that there is no direct one-to-one mapping between the other model elements in BPMN and BPEL. Consequently, after one round-trip only the IDs of activities are preserved and other IDs are missing. Figure 3.2 illustrates the scenario. The upper model represents the original BPMN model, in which all model elements carry IDs. The lower model is one that was exported to BPEL and fed back to BPMN. The fragment with the activities E and A was inserted. With respect to elements that are present in both models only activities carry IDs. Activity D was assigned an ID, because it was deleted and recreated (the differing ID is left out for illustrational reasons).

In this use case the available IDs for activities can be utilized to establish a matching. For elements without IDs the matching has to be signature- or similarity-based. In this, the BPEL-2-BPMN Feedback use case is very similar to the one presented in Section 3.1.1.
Figure 3.2: The workflow graphs of a BPMN model that was exported to BPEL, changed, and reimported back to BPMN. Unique identifiers are preserved only for activities.

3.1.3 Reference Model Customization (Use Case 3)

Reference models are generic models that represent recommended practices for a specific process of a certain domain [45]. They provide generic solutions that can be customized to the specific needs of a company. By the use of reference models the time and effort to design and deploy processes can be reduced [31]. Additionally, using reference models leads to better and optimized process designs as these models have been developed by experts over a longer period of time [30]. It is common practice to customize the relevant subset of the models that are provided by large standard application packages like ERP products, e.g. the SAP reference model [9] or IBM’s Insurance Application Architecture (IAA) [19].

Customizing reference process models usually involves adding, removing or modifying process model elements, such as activities, control nodes, and subprocesses.

Reference Model Customization is used by established, mature companies. These companies have their own processes already in place. They want to determine, how much their processes adhere to the reference models and adjust their processes such that these are executed in a more standard or recommended way [30].

A related use case is the merger of two companies, for which existing processes of the two former companies have to be merged and adapted to fulfill the needs of the new company.

In both scenarios a matching between the own process and the corresponding reference (or company) model would support the business analysts in their task of detecting common and differing parts of the models. The adherence to the other model is expressed by the ratio of matched and unmatched model elements (for the Reference Model Customization use case). Based on the matching, change operations can be computed which indicate the necessary adjustments. The business analyst can then manually select the desired changes from the set of all change operations that would transform one model completely into the other (in both use cases).

As the two models usually do not share any common history, there is no ID mapping that could be exploited. In contrast to the two previous use cases the matching has to
be solely signature- or similarity-based. Figure 3.3 exemplifies this. The lower workflow graph stands for a customized company process and the upper workflow graph represents a general reference model, where the activities $B'$ and $B''$ are generalized to the activity $B$. The two models do not share common IDs for their model elements.

![Figure 3.3: The workflow graphs of a reference process and a refined process that do not share common IDs.](image)

3.2 On Optimal Matching for Change Detection

In the previous section we introduced three use cases that require a matching of model elements is required. In this section it is discussed how an optimal matching is characterized and to which extend a confidence function for the presented use cases can be defined. Confidence functions express the quality of correspondences and are necessary to compare matchings with each other (see Section 2.4.3).

3.2.1 On Defining Concrete Confidence Functions

The criteria that are used to match model elements are “based on heuristics that are not easily captured in a precise mathematical way” [48]. For example, in related work a “good” matching is defined as being “correct” and “accurate” [14]. Rivera et al. [50] describe matching two models $M_1$ and $M_2$ as finding the different objects from both models that “represent the same element”. What is meant by “correct”, “accurate” and “represent the same element” is not formalized. The reason is that defining a concrete confidence function is difficult, especially for the general case.

A concrete confidence function depends on multiple dimensions. In the following we provide a list of conditions that influences the perception of an optimal matching and thus the confidence function. The list should give an impression of the various factors that influence the optimum.

**Use Case** How a matching should be established heavily depends on the concrete use case. We introduced the use case of Version Control (Use Case 1) and Reference
Model Customization (Use Case 3) in the previous Section 3.1. There we explained that it might make sense to match an activity to a subprocess in Use Case 3 if this indicates a refinement, but that only elements with the same type should be matched in Use Case 1 in order to compute appropriate change operations.

**User Preference** Different users expect different matchings (compare e.g. [38]), especially when there are different plausible possibilities to create correspondences.

**Model Size** The optimal matching is influenced by the model size and the number of model elements. Whether two model elements should be matched depends on the size of the models. For example, when a model contains only one decision node then the user is more likely to expect it to be matched to the only decision node in another model. In bigger models the user expects the model structure to be taken into account much more than in small models. In the latter, the single model elements are valued more than the overall structure. This is especially true for elements without an identity, e.g. control nodes.

**Model History** For the Use Cases 1 and 2, in which a second model is derived by changes from the first, the model history is significant. The user expects the computed change operations to conform to his actions upon the model. The computed operations depend directly on the established matching. Consider the following example: the original model $M$ contains two decision nodes $d_1$ and $d_2$. The derived model $M'$ contains two decision nodes $d'_1$ and $d'_2$, but at completely different positions in the model. An optimal matching would lead to the detection of move operations that correspond to the actions of the user. Out of the two possibilities to match the four decision nodes, the correspondences that conform to the user actions (the model history) have to be established in order to compute the correct change operations.

The described conditions show that it is difficult to achieve an optimal matching via a matching algorithm. For example, a matching algorithm cannot correspond to the preferences of all users at the same time or compute an optimal matching when user preferences are not known beforehand [38].

Another example is the model history. It can be impossible for a matching algorithm to compute the optimal matching with respect to the model history only by comparing two models.

Consider the example that was given for the condition “Model History”: assume that the decision nodes were deleted, recreated, and then moved. The user expects the detection of two move operations that conform to his actions. For the computation of these operations the correct decision nodes need to be matched to each other. The IDs of the decision nodes cannot be used for the matching, because they changed due to the recreation of the elements. Thus, the optimal matching cannot be computed by comparing the models: both decision nodes have different positions in both models and it cannot be decided which decision was moved whereto.
3.2.2 A Concrete Confidence Function for Change Detection

A confidence function that takes into account all the conditions that influence the optimal matching (see Section 3.2.1) and that is applicable in the general case is too complex and impractical. It might not even be possible to define such a function (e.g. such that it reflects the user preference of all different users). For this reason we define a concrete confidence function for evaluation purposes that is only applicable for a subset of possible matchings of two workflow graphs. For this subset, however, it is clear how the confidence function has to be defined. We first restrict the use case: the matching should be optimal in terms of its suitability to compute type-consistent change operations as required by Use Case 1 and opposed to Use Case 3.

Because of the condition “Model History”, we further restrict the confidence function to a subset of all possible models: we only consider cases where one model $M_2$ was derived from another model $M_1$ by a set $C$ of non-overlapping change actions. “Non-overlapping” means that operations do not cancel out (e.g. insert and delete of same element) or influence each other (e.g. insert a move of same element). For this case we can clearly state how the confidence function has to look like using the applied changes.

The confidence function $\text{conf}_{\text{diff}}$ for two workflow graphs $WFG_1 = (N_1, E_1)$ and $WFG_2 = (N_2, E_2)$ that were derived from each other is defined in Definition 11:

**Definition 11 (conf}_{\text{diff}}, Confidence Function for Change Detection.).** Let $\text{PST}(WFG_1)$ and $\text{PST}(WFG_2)$ be the process structure trees of the workflow graphs $WFG_1 = (N_1, E_1)$ and $WFG_2 = (N_2, E_2)$, where $WFG_2$ was derived from $WFG_1$ by a set $C$ of non-intersecting changes. Let $F_1$ and $F_2$ be the sets of fragments of $\text{PST}(WFG_1)$ and $\text{PST}(WFG_2)$. Then the confidence function $\text{conf}_{\text{diff}}$ for model elements $me_1 \in N_1 \cup E_1 \cup F_1$, $me_2 \in N_2 \cup E_2 \cup F_2$ is defined as follows:

\[
\text{conf}_{\text{diff}}(me_1, me_2) = \begin{cases} 
1 & \text{if } me_1 \text{ and } me_2 \text{ are counterparts and not affected} \\
& \text{by an operation in } C. \\
1 & \text{if } me_2 \text{ was moved and } me_1 \text{ is the counterpart.} \\
-1 & \text{if } me_1 \text{ was deleted.} \\
-1 & \text{if } me_2 \text{ was inserted.} \\
-1 & \text{else.}
\end{cases}
\]

The confidence function for change detection can be used to evaluate a matching, but it cannot be used to define a matching algorithm or a matching metric because it is defined only for a very limited scenario. Also the algorithm should not use explicit information about the model history which is assumed not to be available.

3.3 Requirements for Model Matching

In this section we want to elaborate on the requirements for a matching algorithm that is suitable for the different use cases of change detection presented in Section 3.1. The
requirements are used to choose appropriate matching criteria and to combine them to an algorithm for process model matching. Additional requirements that aim at the implementation of this algorithm are covered in Chapter 5. We take into account the work by Selonen et al. [51] and Foertsch et al. [15] who have developed general requirements for differencing tools and model comparison approaches. Their work, however, is fairly abstract and general as it discusses model comparison in general. Neither does it focus on business process models nor on the three presented use cases. Thus, it is merely a starting point for our work.

We group the requirements into two groups. First, we explain the general “high-level” requirements for the matching algorithm. Afterwards, the requirements for matching up single model elements (“low-level” requirements) are depicted.

### 3.3.1 High-level Requirements

This section lists the general high-level requirements of process model matching. These requirements concern general algorithm features and do not address the comparison of single model elements. Table 3.1 shows an overview of the requirements. The first column shows the short name of the requirement which we will use to refer to it in the future. The second column provides a brief description. In the following we will discuss each requirement individually:

**HR1** In Section 2.4.2 we defined the optimal matching by using an abstract compare operator for matchings. In Section 3.2.2 we defined a concrete confidence function that can be used with these abstract definitions to compare concrete matchings for models derived from each others via non-intersecting changes. The matching algorithm should compute an optimal matching for the cases in which that confidence function is applicable.

**HR2** In Section 2.5.2 we introduced the three approaches to model matching. The matching algorithm suitable for the different use cases has to be capable of combining these approaches. Elements that carry an ID have to be matched with a static identity-based approach. The remaining elements have to be matched with one of the other two approaches (signature- or similarity-based).

**HR3** The matching algorithm should be capable of matching SESE fragments such that this matching can be used to compute appropriate compound change operations (see also requirement HR 1).
### Chapter 3. Requirements for Business Process Matching

#### Table 3.2: Low-level requirements for matching workflow graphs.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Element Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR 1</td>
<td>General</td>
<td>Type-consistent matching</td>
</tr>
<tr>
<td>LR 2</td>
<td>General</td>
<td>Attribute-based matching: ID</td>
</tr>
<tr>
<td>LR 3</td>
<td>General</td>
<td>Activity-centric matching</td>
</tr>
<tr>
<td>LR 4</td>
<td>Activity</td>
<td>Attribute-based matching: Name</td>
</tr>
<tr>
<td>LR 5</td>
<td>Decision, Fork, Join, Merge</td>
<td>Containment-based matching</td>
</tr>
<tr>
<td>LR 6</td>
<td>Decision, Fork, Join, Merge</td>
<td>Neighborhood-based matching</td>
</tr>
<tr>
<td>LR 7</td>
<td>Start, End</td>
<td>Neighborhood-based matching</td>
</tr>
<tr>
<td>LR 8</td>
<td>Loop, Subprocess</td>
<td>Containment-based matching</td>
</tr>
<tr>
<td>LR 9</td>
<td>Fragment</td>
<td>Containment-based matching</td>
</tr>
<tr>
<td>LR 10</td>
<td>Edge</td>
<td>Neighborhood-based matching</td>
</tr>
<tr>
<td>LR 11</td>
<td>General</td>
<td>Relaxed matching conditions</td>
</tr>
</tbody>
</table>

The matching algorithm should have a reasonable computational complexity/runtime such that it can be used in a tool implementation. The overall complexity should be polynomial in the number of workflow graph nodes and edges. The matching should run in a few seconds on an average PC for average-sized (see Section 6.1) process models.

#### 3.3.2 Low-level Requirements

In the beginning of this chapter we described the use cases that we want to support. We then defined the high level requirements for the matching of process models in general. This section lists the requirements for comparing model elements in general, but also the requirements for the matching of each specific element type. Table 3.2 provides an overview of the requirements that we will now discuss one by one. The first column shows the short name of the requirement which we will use to refer to it in the future. The second column shows the affected element types and the third column provides a brief description.

Many of the low-level requirements indicate the necessary matching strategy. A matching strategy denotes which set of features of the two compared workflow graphs is used to create correspondences for other elements. For example, LR 6 requires a neighborhood-based matching strategy. A neighborhood-based matching strategy matches elements based on the correspondences of their directly and indirectly connected neighboring nodes in the workflow graph.

In the following we discuss each requirement:

**LR1** The matching should be type-consistent such that only elements of the same type are matched.

For the Reference-Model-Customization scenario (Use Case 3) there are specific cases where it would be meaningful to match elements of differing types. This
Chapter 3. Requirements for Business Process Matching

would correspond to a refinement of model elements. For example, if a subprocess structure in the concrete model is replaced by an activity in the reference model, this could hint on the generalization of the subprocess into an atomic process step. However, the same difference could as well only be an arbitrary change such that the subprocess structure is replaced by an activity with a completely different meaning. Hence, matching elements with differing type is not desirable in general.

With respect to the first and second use cases matching elements with different types is less important or even undesirable, for example because of the different semantical meaning of the elements. With regard to Requirement HR 1 and the example above we could not detect the change and compute an appropriate operation since a “generalization” operation is not supported (see Section 2.5.3). As a (partial) matching that was established using IDs is in its nature type-consistent, the (remaining) matching based on other approaches should be consistent with this.

Assessing the arguments, there are more reasons for matching elements with the same type only. However, with the appropriate generalization/refinement operations available and with the focus on Use Case 3 it would make sense to match elements of differing types in specific cases.

 LR2 For the elements in Use Case 3 a matching based on IDs would not be successful. For the other use cases an ID-based matching can be established partially. The two process models should always be matched using a static identity-based approach first. Afterwards, (for the elements that could not be matched using IDs) other approaches should be applied. In Figure 3.1, for example, all nodes except for the start node and the activity $D$ can be matched to their counterpart using IDs. The two remaining nodes have to be matched with other approaches (the elements in the grey box remain unmatched because they do not have a counterpart in the first model).

 LR3 Process models are characterized mainly by the process steps that constitute them. Thus, activities should be the foundation of the signature- or similarity-based part of the matching process (see also Requirement LR 2). As a consequence, the matching of elements should be based on corresponding activities, but also on elements with matching IDs, as the correspondences indicated by them are presumed.

 LR4 Activities should be matched based on their semantics. Their position and their connections to other elements are less important than their actual meaning. For this reason a matching strategy based on their attributes should be used. The semantics of an activity is mainly defined by its naming, not its connections. For example, an activity at the same position with a totally different name does not represent the same semantic idea and should not be matched. In contrast, an activity with similar naming at a different position in the process model should be matched. This is illustrated by Figure 3.4. The activities of the two depicted
workflow graphs are matched based on their names and not, e.g., based on their position in the process.

Figure 3.4: Two workflow graphs with corresponding activities.

**LR5** Most of the fragments in process models are well-structured (see Section 6.1). Thus, decision&merge and fork&join pairs fully enclose parallel or alternative sequences of nodes. As a consequence the user comprehends them as a structure that encloses other elements. In these cases fork, decision, join, and merge nodes should be matched based on the nodes they enclose. That means that even if the order of elements that are encapsulated by them changes, the enclosing gateway nodes should still be matched. Thus, a containment-based matching strategy should be used to match these gateways. Figure 3.5 shows an example, where decision&merge pairs enclose activities. The gray shaded boxes denote the pairs that correspond to each other due to similar enclosed nodes.

**LR6** Fork, decision, join, and merge nodes should be matched based on their neighboring nodes because that is what defines them foremost even if they have other attributes like branching conditions (neighborhood-based matching strategy). A control node at the same absolute position in a model should not be matched if most of its neighboring nodes have changed (case 1). However, a control node at a different absolute position should be matched if it has the same or a similar relative position to corresponding neighboring nodes (case 2). In the change detection scenarios for example this would correspond to the deletion or replacement (case 1) or the shift to another position (case 2) of a whole model area. Figure 3.6 illustrates this.
Figure 3.5: Two workflow graphs with corresponding decision&merge pairs that enclose corresponding activities.

Figure 3.6: Two workflow graphs with corresponding decision nodes that are surrounded by corresponding activities.
Similar to requirement LR 6 start and termination nodes should be matched based on their type (LR 1) and their predecessors or successors, respectively. Using a neighborhood-based matching strategy is the only possibility to detect corresponding elements as they have no other properties (e.g., names).

In the workflow graph loops and subprocesses are represented by pairs of opening and closing nodes that enclose other elements (see Section 2.2.2). These structures should be matched based on the nodes they enclose because this is what defines them the most (containment-based matching strategy). A loop structure with the same enclosed content at a different absolute position should be matched (case 1), while a loop structure at the same absolute position with totally different enclosed nodes does not represent the same loop (case 2) and should not be matched. For change detection this would correspond to the shift to another position (case 1) or the deletion and replacement of the whole loop or subprocess structure (case 2). Figure 3.7 and Figure 3.8 illustrate the containment-based matching. In both cases the loop and subprocess nodes that correspond to each other have different predecessors and successors, but enclose the similar elements.
Similar to loop and subprocess structures, fragments should be matched based on a containment-based matching strategy as fragments are a mechanism to group (contained) model elements. Thus, it does not make sense to group fragments that do not contain the same nodes and the reasoning for loops and subprocesses also applies to fragments. Figure 3.9 highlights the corresponding fragments of the two depicted workflow graphs by thicker lines. They were matched up due to similar contained elements.

Edges should be matched using a neighborhood-based matching strategy, based on source and target nodes. Since only untyped control flow edges are considered in this thesis, edges can be discriminated and identified only based on their source and their target. Thus, edges $e_1 = (s_1, t_1)$, $e_2 = (s_2, t_2)$ should only be matched if their sources $s_1, s_2$ and their targets $t_1, t_2$ correspond to each other.

The matching of model elements should not be too strict: if meaningful amounts of nodes enclosed by gateway pairs (LR 5) and loop or subprocess structures (LR 8) correspond to each other the enclosing nodes should be matched. Similarly, fragments should be matched if most of the nodes they contain correspond to each other (LR 9). Activities (LR 4) should be matched even if the names are not equal, but similar enough. Control nodes (LR 6, LR 7) should be matched if their neighborhood is similar, but not equal. For example, even if no directly connected node is the same but the bigger neighborhood shares a lot of common nodes, control nodes should be matched (compare shaded boxes in Figure 3.6).

For change detection (HR 1) the matching of elements which do not share the exact same set of features is important in order to detect operations that express changes upon elements, e.g. move operations. Thus, a matching with relaxed matching criteria is necessary to compute good matchings in terms of optimality. It is also much more challenging and interesting from a scientific point of view.
3.3.3 Suitability of Matching Approaches

In this chapter we discussed the three use cases and deduced the requirements for process model matching. They can be summed up as follows: first, the matching algorithm has to be capable of matching model elements and model fragments. Second, it has to be capable of matching model elements based on element attributes like IDs or names. Third, the algorithm also has to be capable of matching model elements based on their neighboring nodes in the workflow graph. Fourth, it has to be capable of matching model elements and fragments based on contained nodes. In the following we want to discuss shortly how suitable the different matching approaches are that were presented in Section 2.5.2.

The first presented approach, the static identity-based approach, is clearly not suitable for the third use case where no static unique identifiers for corresponding elements are present. However, such an approach should be used for elements that are assigned an ID in the other two use cases. An ID-based matching approach is explicitly demanded by Requirement LR 2. Still, for the matching of fragments and corresponding elements without IDs another solution has to be found.

The signature-based approach is also not suitable for process model matching in the presented use cases. According to the requirements the matching has to be founded, for example, on neighboring nodes (e.g. LR 6 or LR 10). Especially because of Requirement LR 11 it cannot be assumed that the neighborhood of two nodes \( n_1, n_2 \) that correspond to each other is completely identical. As a consequence, it will be difficult to compute a unique signature for these nodes \( n_1, n_2 \) in order to match them with this approach. This example shows that the signature-based approach is too inflexible to use it for change detection.

Similarity-based matching seems to be much more suitable than signature-based matching to compute a matching for elements that were not matched based on IDs. First, it is easier to fulfill Requirement LR 11 that requires matching elements that are not completely the same with respect to some measure. This is opposed to the idea of signature-based matching where elements are matched based on identical signatures. Second, the similarity-based approach can incorporate static identity-based matching by defining a similarity metric that is based on ID equivalence. Hence, similarity-based matching can be combined easily with static identity-based matching. Comparing the three matching approaches similarity-based matching is the only approach that can incorporate all the strategies that are needed with regard to the requirements.

3.4 Discussion of Related Work

In this section we discuss the related work of process model comparison. There are many areas where model comparison plays a role and where different approaches were developed that influenced process model matching. Table 3.3 gives an overview of the approaches that we discuss in the following. It also shows the matching approaches and capabilities needed to fulfill the requirements (bottom row). A plus sign denotes the capability or support for the aspect, while a minus sign expresses that it is missing. We
first present generic approaches for model matching that are not specific to a certain type of model. Afterwards, we describe related work about schema matching, source code matching, and UML model comparison. Thereafter, we present process model comparison approaches.

### 3.4.1 Generic approaches

Chawathe et al. [8] propose a similarity-based algorithm for the change detection between two trees. In an initial matching step tree elements are matched, afterwards the change operations are computed. The matching algorithm of the first step consists of two phases. First, an abstract compare function is used to decide whether leaf nodes correspond to each other. Second, an inner node matching based on this leaf node matching is computed. By defining a concrete compare function that does so, the matching of leaf nodes can be based on their attributes. The matching of leaf nodes could also be based on their neighborhood in the graph, however, the algorithm originally works on trees instead of general graphs and implementing such a compare function requires more user effort. The inner node matching of the trees is based on the subtree leaf node correspondences. In this, the inner node matching suits the requirement LR 9 for the matching of fragments very well. Because the algorithm is simple, it can be easily extended to match some leaf node types based on inner nodes.

In [4] Alanen and Porres discuss the difference and union of models in the context of version control systems. For the difference detection they use a completely static identity-based matching algorithm. Hence, matching elements based on their attributes

---

**Table 3.3: Comparison of related work with respect to the requirements.**

<table>
<thead>
<tr>
<th>Authors</th>
<th>Matching Approach</th>
<th>Matching Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>static iden.</td>
<td>signature</td>
</tr>
<tr>
<td>Generic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chawathe et al. [8]</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Alanen &amp; Porres [4]</td>
<td>–</td>
<td>+</td>
</tr>
<tr>
<td>Lin et al. [33]</td>
<td>–</td>
<td>+</td>
</tr>
<tr>
<td>Rivera &amp; Vallecillo [50]</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>DB Schema</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Melnik et al. [38]</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Source Code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fluri et al. [14]</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>UML</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Melnik et al. [38]</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Xing et al. [59]</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Treude et al. [55]</td>
<td>–</td>
<td>+</td>
</tr>
<tr>
<td>Kolovos et al. [28]</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Process Models (struct.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Madhusudan et al. [37]</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Küster et al. [29]</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>Process Models (behav.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lu et al. [33]</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Dijkman et al. [10]</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Requirements</td>
<td>(+)</td>
<td>–</td>
</tr>
</tbody>
</table>

...
(other than IDs), their neighborhood or their containment relationship is not supported at all.

Lin et al. [33] present a signature-based algorithm to compare models with different meta models. The meta models are represented as hierarchical graphs of nested sub-models. The algorithm matches level-wise starting with the outmost root model and continues top down with the contained sub-models. Two elements of a sub-model are matched if the computed signature for them is identical. If there is more than one candidate in the other model, a candidate is picked by using a further metric. This metric is the ratio of identical signatures of directly connected nodes of the two elements. The algorithm directly supports the matching of elements based on their attributes, but as signatures have to be identical (not similar), the algorithm is stricter than similarity-based approaches. The neighborhood of two nodes is considered only in some specific cases, but even then only with distance one to the compared elements. Matching based on containment is not supported as the algorithm works top down (and not bottom up). As it matches level-wise, it does not match sub-models across levels. However, this would be required to match workflow graph fragments that were moved across levels.

In [50] Rivera and Vallecillo discuss the calculation of differences between models conforming to arbitrary meta models. The matching is based on IDs (static identity) and similarity values for basic attributes. They give formulae to calculate the similarity between boolean, string, and numerical attributes as well as references (outgoing edges to other nodes). To compare two referenced elements their similarity is taken into account. However, references from these two elements to other elements are left out in the similarity computation to avoid cyclic dependencies. After pairwise comparison of the elements the matching is created. While the matching based on attribute similarity is supported directly, only elements with distance one are considered with respect to the neighborhood. The matching based on containment is not supported.

3.4.2 Schema Matching

Melnik et al. describe a similarity-based algorithm [38] for graph matching that is based on an iterative fixpoint computation. They apply this algorithm to schema matching by converting schemas into a so called propagation graph. The basic matching heuristic is that two elements are more similar to each other if elements connected to them are similar. They apply an initial element matching based on the similarity of element names and run the fixpoint calculation afterwards. After a filtering step they establish the matching based on the computed similarity values for element pairs. This algorithm is capable of matching elements based on their attributes. It shows particular strength in matching elements based on their neighboring elements as required by Requirement LR 6, because the similarity of elements is propagated beyond directly connected elements by the iterative computation. In contrast, containment-based matching is not supported at all.
3.4.3 Source Code Matching

In [14] Fluri et al. present an algorithm to detect structural changes in source code. It uses the abstract syntax tree of the code and is an extension of the algorithm presented by Chawathe et al. [8]. They propose different changes to the original algorithm and present different string similarity metrics for the matching of leaf nodes and then evaluate different combinations of proposed changes to the original algorithm. The solution is tailored to deal with problems that arise in this particular domain. As the algorithm is mainly an adaption of the original algorithm by Chawathe et al., it has the same capabilities and deficiencies with respect to our requirements.

3.4.4 Matching of UML Diagrams

Kelter et al. give an algorithm to compare UML diagrams in [24]. They define an own meta model that is simpler than UML, but more generic and they are able to compare instances of this. They argue that a mapping for every diagram type of UML to this meta model can be defined and apply their approach to class diagrams. The algorithm is divided into bottom up and top down phases with respect to the different levels of the model (e.g. package, class and operation). First, correspondences are computed bottom up from the classifier to the package level based on the similarity between elements. A correspondence is only created if there is a unique similar element. Then every time a new correspondence is computed, it is propagated top down and can lead to changed similarity values and hence, changed correspondences for the contained elements on lower levels. To speed up the computation a hashing algorithm is used to identify matching elements more easily. The authors state that the algorithm is suitable for documents with many composite relations, but not that much for graph-like documents, in which relations between elements (on the same level) play an important role, as for example in state charts. The similarity between elements is determined by their attributes, correspondences of their parent elements, or elements they contain. Matching elements based on their neighborhood at the same level is not supported.

Xing et al. present a similarity-based algorithm to detect changes in subsequent versions of object-oriented systems [59]. It detects the changes between two class models that were reverse engineered from two corresponding Java source code versions. They use an own meta model that is defined according to the semantics of UML. The algorithm works top down from packages over classes and interfaces down to methods and fields. On each level it matches corresponding elements (e.g. packages or classes). The similarity of two elements is based on the similarity of their names and their direct relations to other elements (e.g. when a method reads a field). Moved elements are detected based on their name similarity and type. The algorithm directly supports the matching based on attributes of elements, although it uses their names only. The neighborhood of elements is taken into account, but only the directly connected elements with distance one. Because the algorithm works top down (e.g. from classes to fields), it does not match entities based on the contained elements. Instead elements are matched based on their parent entity.
Chapter 3. Requirements for Business Process Matching

In [55] Treude et al. present a signature- and similarity-based algorithm to compute the differences between models. The matching is based on several phases. The first phase hashes model elements to speed up their comparison. Two elements from the two different models are immediately matched when they have the same hash value. In the second phase the remaining elements are indexed and one or several search trees are created. A search tree efficiently finds for each element the most similar elements in the other model. In the third phase exact similarity values and matchings are computed. Treude et al. apply their approach to match UML class diagrams. Similarities are computed in a bottom-up/top-down order. Elements that are considered to be similar and which have no other similar element are matched immediately. Each created correspondence causes the algorithm to switch over to a top-down phase that propagates the new correspondence downwards to the children. If a correspondence of a child element is changed, the change is propagated back bottom up to the parent element. The computation of the similarity of elements has to handle cycles caused by cross-references between elements. Such cycles are dealt with by iterating over a cycle as long as new matches can be found. Summing up, the algorithm creates a matching based on the similarity of element attributes, their containment relationship, and references to other elements (neighborhood). For the neighborhood, however, only directly connected elements are considered.

In [28] Kolovos et al. derive requirements for model comparison and present a concrete rule-based comparison approach. They explain the syntax of their rules and exemplify it by presenting a rule set to match a UML model against a relational database model. If two elements are not known to be corresponding, the system finds and executes the defined matching rules for those element types. If the evaluation of such a rule depends on whether other elements are matched and if that is still unknown, the system matches these elements first by executing applicable rules. Elements are corresponding if an applicable matching rule evaluates to true. The rule-based approach does conform to none of the three presented matching approaches as it is not based on identical static identifiers, signatures, or similarity values. The rules that are presented match elements based on their type, their name, and the name of elements referenced by them. The user can define rules that compare elements based on other attributes as well. As rules to match elements based on their neighboring correspondences or their containment are not described, the approach is not capable of using these matching strategies at this point. However, the user could define such rules on his own. For this reason we do not rate the matching capabilities of this approach.

3.4.5 Matching of Process Models

Structural Matching

Madhusudan et al. [37] propose a framework for workflow modeling and design by adapting workflow cases from a repository of process models. For the retrieval of suitable workflows from the repository they propose a similarity flooding algorithm based on the al-
algorithm of Melnik et al. [38]. Because the algorithm is an adaption, it shows the same strengths and weaknesses with respect to our requirements as the original algorithm.

Küster et al. [29] present a change detection algorithm for process models. The change detection is based on a matching that is established using a static identity-based approach. First, model nodes are matched based on their IDs. Then, edges are matched based on IDs and their source and target nodes. Küster et al. use a PST decomposition (see Section 2.3.2) to identify fragments and a set of rules to propagate the ID-based element matching to the process fragments. The matching of elements is based only on the ID attributes of model elements, while the neighborhood of elements is not considered at all. Process fragments are identified and matched but not in the way required by Requirement LR 9.

**Behavioral Matching**

Behavioral matching of process models matches process models with regards to common execution behavior and not based on similar elements in terms of their attributes, their neighborhood, and their containment. For this reason they do not fulfill our requirements to structural model matching. We list them to provide the reader a more complete picture of related work in the field of process model comparison.

Lu et al. [34] describe a technique for the discovery of preferred process variants from a repository based on the similarity between a query process and the process variants in the repository. Their approach provides a quantitative measure for the similarity of whole processes. To define the overall similarity of process model variants their structure and their execution behavior is taken into account. They do the first by checking the overall structure of the processes for identical parts (in contrast to checking for similar elements and relations). They do the latter by reducing two models to their common activities using a graph reduction technique and comparing the execution paths.

In [10] Dijkman et al. present a technique to diagnose differences between business process models in EPC notation [23] with finite state space. The differences are typed according to a typology developed in previous work. Differences between models are detected in terms of complete trace equivalence and reduction rules are applied to reduce the state space. The initial mapping of activities is done by hand, while it is argued that there could be computational support.

**3.4.6 Overviews on Model Matching and Model Comparison**

In this section we list work that provides an overview on model matching and model comparison in related work.

**Generic**

Kolovos et al. [27] provide an overview of fundamental steps of the model differencing process and compare existing approaches with a focus on their trade-offs. They classify the existing work based on the division of the differencing problem into three phases and four strategies.
Chapter 3. Requirements for Business Process Matching

Bendix et al. [5] give an overview on the work of differencing and merge support for model based development by briefly presenting the approaches of different researchers. They point out that it still lacks sufficient tool support.

**UML-specific**

In [15] Försch and Westfechtel discuss the state of the art in fine-grained version control for software diagrams with an emphasis on UML diagrams. They propose requirements to assess the tools and propose steps for the design of such a tool. Then they provide a classification of existing tools.

In [51] Selonen et al. compare the characteristics of currently available UML-based model comparison approaches. After discussing how these approaches are compared they argue that the approaches are too subjective and call for more systematic evaluation.

### 3.4.7 Results of Discussion

In this section we discussed the related work and the capabilities of the presented approaches. Table 3.3 shows that no presented approach provides all the capabilities required for the matching algorithm (compare with the last line of the table). While a static identity-based matching can be incorporated by a similarity-based approach (see Section 3.3.3) the algorithm has to support matching elements based on their similarity.

It has to be capable of matching elements based on their attributes and their neighborhood. Here the matching based on directly connected elements is not sufficient. The matching of hierarchical process fragments has to be based on contained element correspondences.

Nevertheless, there are algorithms that show strengths in some of the required areas. The algorithm by Chawathe et al. and its adaption by Fluri et al. have a strong focus on matching inner tree elements based on contained correspondences as required e.g. for matching process fragments. The similarity flooding algorithm by Melnik et al. and the related algorithm by Madhusudan et al. are tailored towards matching elements based on their neighborhood beyond directly connected elements. Both algorithms on their own would not fulfill the requirements, but they shall be examined in greater detail to find out if we can reuse their algorithmic ideas.

### 3.5 Summary

In this chapter we described three use cases that require model matching and defined a confidence function for a subset of model pairs for the Version Control use case. We discussed the requirements for the matching algorithm and presented the approaches from related work. The discussion showed that there is no known approach that fulfills all requirements. In the following chapter two algorithms with particular strengths from related work are described in more detail. Thereafter, we reuse algorithmic ideas from these approaches and define metrics for measuring the similarity between elements. We then combine them to a matching algorithm.
Chapter 4

Matching Metrics and Algorithms

In this chapter we present two algorithms that showed particular strengths in the discussion of related work in the proceeding chapter. We reuse their algorithmic ideas and define other metrics for measuring the similarity between elements. In the last part of this chapter we describe how we combine these metrics to our matching algorithm.

4.1 Existing Model Matching Algorithms

In Section 3.3.3 we discussed the advantages of similarity-based matching that incorporates an ID-based approach as a part of it. In Section 3.4 we examined the capabilities of algorithms in related work. Now we focus on two of these algorithms which showed strengths in the three different areas important for the matching of process models (see Table 3.3).

The first one proposed by Chawathe et al. [8] is a rather generic algorithm to match trees and can be adapted to match workflow graphs in their PST representation. It heavily influenced the matching algorithm developed in this thesis that we will present in Section 4.3, especially the sub-algorithm to match fragments.

The other algorithm is the similarity flooding by Melnik et al. [38] that was adapted for matching process models by Madhusudan et al. [37]. It is a fairly different approach than the algorithm by Chawathe et al and works on directed graphs. It bases the similarity value for two nodes on the similarity value of their neighboring nodes. This approach provided ideas for the sub-algorithm that we use to match control nodes. The algorithm is based on an iterative fixpoint calculation which solves the cyclic dependencies in the computation of the similarity values.

In this section these two algorithms are presented in further detail and their advantages and drawbacks are discussed. The concrete commonalities and differences between our work and these algorithms will be discussed in Section 4.3.11.
Chapter 4. Matching Metrics and Algorithms

4.1.1 Chawathe-Algorithm

The algorithm proposed by Chawathe et al. [8] is designed for the change detection in hierarchically structured information and assumes a tree structure. We will first present the algorithm and then discuss its advantages and drawbacks.

Matching Algorithm

The algorithm matches the nodes of two trees in two steps: first the leaf nodes and afterwards the inner nodes are matched. Each node \( n \) in a tree has a label \( l(n) \) and a value \( v(n) \).

The matching of the leaf nodes is based on a function \( \text{compare} \) that returns a distance value between 0 and 2 for a pair \( n, n' \) of compared elements. The distance value can be understood as the opposite of the confidence, which was discussed in Section 2.4.3. A concrete compare-function is not part of the algorithm, only its in- and output is defined.

Two leaf nodes can be matched if they are not matched to other elements already and if the equality function \( \text{equal} \) for leaf nodes returns true for the two nodes. It is defined in Definition 12.

**Definition 12** (Equality of Leaf Nodes [8]). Let \( T_1 = (N_1, E_1) \) and \( T_2 = (N_2, E_2) \) be two trees with \( n \in N_1 \) and \( n' \in N_2 \) and \( n, n' \) being leaf nodes. Let \( 0 \leq \text{compare}(v(n), v(n')) \leq 2 \) be a compare-function for leaf nodes. Furthermore, let \( 0 \leq f \leq 1 \) be a threshold value. Then the equality of leaf nodes \( \text{equal}(n,n') \) is defined as follows\(^1\):

\[
\text{equal}(n,n') = \begin{cases} 
  \text{true} & \text{if } l(n) = l(n') \text{ and } \text{compare}(v(n), v(n')) \geq f \\
  \text{false} & \text{otherwise.}
\end{cases}
\]

The matching for the inner nodes is based on an equality function \( \text{equal} \) for inner nodes. The function returns true if the two compared nodes share enough correspondences among their leaf nodes. Two inner nodes are matched if they are not already matched to some other node and if the equality function returns true. It is defined in Definition 13.

**Definition 13** (Equality of Inner Nodes [8]). Let \( T_1 = (N_1, E_1) \) and \( T_2 = (N_2, E_2) \) be two trees with \( n, m \in N_1 \) and \( n', m' \in N_2 \) and \( n, n' \) being inner nodes and \( m, m' \) being leaf nodes. Let \( M \subseteq N_1 \times N_2 \) be a matching of leaf nodes. Furthermore, let \( |n| \) denote the number of leaf nodes in the subtree of \( n \). Let \( \frac{1}{2} < t \leq 1 \) be a threshold value.

We first define two auxiliary functions:

\[
\text{contains}(n,m) = \begin{cases} 
  \text{true} & \text{if } n \text{ contains } m \text{ in its subtree} \\
  \text{false} & \text{otherwise.}
\end{cases}
\]

\[
\text{common}(n,n') = \{(m,m') \in M | \text{contains}(n,m) \text{ and contains}(n',m')\}
\]

\(^{1}\)the original compare function evaluates to true when \( \text{compare}(v(n), v(n')) \leq f \), so it returns a distance and not a similarity. We change this detail to make the compare function more intuitive and easier to understand for the user.
The equality of inner nodes $equal(n, n')$ is defined as follows:

$$equal(n, n') = \begin{cases} 
true & \text{if } l(n) = l(n') \text{ and } \frac{|\text{common}(n, n')|}{\max(|n|, |n'|)} > t \\
false & \text{otherwise.}
\end{cases}$$

The matching algorithm that uses the described equal-functions works in two steps. First, it iterates over all leaf node combinations, so it processes pairs of leaf nodes where one element is from one tree and the other is from the other tree. If both leaf nodes of a pair are unmatched, the equal-function for leaf nodes is evaluated. Otherwise the next pair is processed. As soon as the $equal$-function returns true for two nodes, a correspondence for these nodes is added to the Matching $M$. For example, the function returns false for the elements $E$ and $Q$ in Figure 4.1 (compare Table 4.1). Thus, no correspondence is created for them. In contrast, the function returns true for the elements $E$ and $H$ and a correspondence for these nodes is created.

![Figure 4.1: Two trees with corresponding leaf and inner nodes.](image)

Table 4.1: Similarity values for leaf nodes of the trees in Figure 4.1. The bold values denote the created leaf node correspondences.

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>0.3</td>
<td>0.1</td>
<td>0.3</td>
<td>0.2</td>
</tr>
<tr>
<td>N</td>
<td><strong>0.9</strong></td>
<td>0.4</td>
<td>0.1</td>
<td>0.3</td>
</tr>
<tr>
<td>O</td>
<td>0.8</td>
<td>0.4</td>
<td>0.2</td>
<td><strong>0.8</strong></td>
</tr>
<tr>
<td>P</td>
<td>0.6</td>
<td>0.3</td>
<td>0.7</td>
<td>0.9</td>
</tr>
<tr>
<td>Q</td>
<td>0.1</td>
<td>0.2</td>
<td>0.6</td>
<td>0.3</td>
</tr>
</tbody>
</table>

The order in which the nodes are iterated influences the leaf node matching, because the next possible (opposed to best possible) correspondence is created based on the currently evaluated leaf pair. For example, $P$ and $H$ have a higher similarity value, but
are not matched to each other, because $H$ and $O$ are traversed and matched to each other before.

After the correspondences for the leaf nodes are computed, the inner node correspondences are calculated using the formula depicted in Definition 13. Similar to the matching of leaf nodes the algorithm iterates over all pairs of inner nodes. If both inner nodes of the pair are unmatched, the equal-function for inner nodes is evaluated. Otherwise the next element pair is processed. As soon as the equal-function for a pair returns true, a correspondence with these nodes is added to the Matching $M$. In the figure the nodes $B$ and $J$ are matched to each other, because they have the same label and their subtree contains two out of three leaf nodes correspondences: $2/3$ is above the threshold. $A$ and $I$ are matched, because they share all correspondences ($4/4$) and $L$ and $D$ are not matched, because they do not share enough correspondences ($0/1$). Again, the order of iteration influences the matching.

Advantages and Drawbacks

The algorithm itself is fairly intuitive as it is based directly on the comparison (for leaf matching) or the correspondences (for inner node matching) of leaf nodes. It is flexible as different compare()-functions can be used. By defining the appropriate compare()-function, domain knowledge can be leveraged to match the trees. The matching can be further adjusted by using the threshold parameters. The algorithm has the runtime $O(N_1 \times N_2 \times t_c)$, where $t_c$ is the runtime of the compare-function that is used. As the overall runtime also depends on this compare-function, which has to be defined by the user, the runtime can be very high when complex metrics are used.

One general drawback of the algorithm is that always the next possible correspondence is established opposed to the one with the highest similarity value. This accelerates the algorithm, but can lead to a sub-optimal matching as the first possible correspondence does not have to be the most suitable one. As soon as an element is matched it will be ignored for the matching of other elements. Another drawback is that no concrete algorithm for the matching of leaf nodes is given. Before this approach can be used appropriate metrics and subalgorithms for the compare-function have to be developed. While attribute-based and neighborhood-based metrics could be implemented, containment-based matching of leaf nodes is not possible as the leaf nodes are processed before inner node correspondences are established. Hence, the algorithm cannot be used for matching in its current form.

4.1.2 Similarity flooding

The similarity flooding algorithm is a matching algorithm for directed labeled graphs. The algorithm was originally used for schema matching [38], but Madhusdusan et al. adopted it for the matching of process models [37].

The algorithm uses an iterative fix-point computation to find corresponding elements. The basic idea is that two elements are similar if their adjacent elements are similar to each other. The algorithm first computes initial similarity values for all node pairs,
then it propagates these similarities along the edges to the adjacent node pairs until an equilibrium of similarity values is reached.

Matching Algorithm

We explain the approach presented by Madhusudan et al. but simplify steps to increase the readability. Their algorithm is in principle the same as the original by Melnik et al., but different parameters were adapted for the comparison of process models.

The algorithm consists of four steps. First, initial similarity values are computed. Second, an appropriate data structure called propagation graph is constructed from the two process models. Third, the actual fixpoint calculation is executed. The last step is to establish the correspondences for the element pairs with the highest similarity values.

Propagation Graph

There is one node pair \([n_i, m_i]\) with \(n_i \in N_1, m_i \in N_2\) in the propagation graph for every element of the cross-product \(N_1 \times N_2\) of the nodes of the two workflow graphs \(WFG_1 = (N_1, E_1), WFG_2 = (N_2, E_2)\). Figure 4.2b shows the propagation graph for the workflow graphs in Figure 4.2a.

Figure 4.2: Example for the similarity flooding algorithm: (a) two workflow graph snippets, (b) the according propagation graph and (c) a visualization of the propagation of similarity values upstream and downstream.

Every edge in the propagation graph has corresponding edges in the original process models. This means that for the two workflow graphs \(WFG_1 = (N_1, E_1), WFG_2 = (N_2, E_2)\) there is an edge \(([n_i, m_i], [n_j, m_j])\) in the propagation graph if and only if \((n_i, n_j) \in E_1\) and \((m_i, m_j) \in E_2\). The figure shows that there is an edge from \(a_1\) to \(b_1\) and \(a_2\) to \(b_2\) in the original workflow graphs. Consequently, there is an edge from the node pair \([a_1, a_2]\) to the node pair \([b_1, b_2]\).
After the graph structure is created, two weight values are assigned to each edge. One weight $w_f(i)$ is used to weigh the similarity that is propagated from the source to the target of the edge $i$. The other weight $w_b(i)$ is used to weigh the similarity that is propagated backwards from the target to the source of the edge $i$. The idea is that the similarity of a node should be spread out equally among all of its outgoing edges (using $w_f(i)$). Moreover, its similarity should be spread out equally among all of its incoming edges (using $w_b(i)$). For this reason, the weights are computed as follows: For each map pair $i$ in the propagation graph with $O_{total}$ outgoing edges the weight $w_f(i)$ of its edges is $w_f(i) = 1.0/O_{total}$. For each map pair $j$ in the propagation graph with $I_{total}$ incoming edges the weight $w_b(j)$ of its edges is $w_b(j) = 1.0/I_{total}$. Figures 4.2b and 4.2c illustrate this. The node pair $[a_1, a_2]$ has four outgoing edges in the propagation graph, hence its weight $w_f$ is 0.25 as depicted in Figure 4.2c. The weight of a map pair is propagated along all its outgoing edges using $w_f(j)$. Furthermore, it is propagated along all its incoming edges using $w_b(j)$.

**Fixpoint calculation** During the fixpoint calculation similarity values for the set of all node pairs are computed iteratively. Let $\sigma^{i+1}([x, y]) \leq 0$ be the similarity function of a node pair that represents two nodes $x \in N_1, y \in N_2$ for two workflow graphs $WFG_1 = (N_1, E_1), WFG_2 = (N_2, E_2)$. The algorithm computes $\sigma$-values iteratively. Let $\sigma^i_{all}$ denote the similarity values for $WFG_1$ and $WFG_2$ after the $i$-th iteration. $\sigma^0_{all}$ denotes the initial similarity values computed in step 1. Then the basic update formula for a node pair is [38]:

$$
\sigma^{i+1}([x, y]) = \sigma^i([x, y]) + \sum_{(n,x) \in E_1, (m,y) \in E_2} \sigma^i([n, m]) \cdot w_{f,[n,m]}
+ \sum_{(x,k) \in E_1, (y,l) \in E_2} \sigma^i([k, l]) \cdot w_{b,[k,l]}
$$

Thus, the new similarity value for a node pair $[x, y]$ is the result of the previous iteration summed up with the weighted similarities of adjacent node pairs. The similarity value of each node pair $[n, m]$ that has an outgoing edge to $[x, y]$ is multiplied with its forward weight $w_{f,[n,m]}$. The similarity of each node pair $[k, l]$ that has an incoming edge from $[x, y]$ is multiplied with the backward weight $w_{b,[k,l]}$. Figure 4.2c exemplifies this. The node pair $[a_1, a_2]$ propagates its weighted similarity value of the previous iteration among its outgoing edge using $w_f$. This value is added to the similarity value of $[c_1, b_2]$ for the current iteration. The node pair $[c_1, b_2]$ propagates its weighted similarity value of the previous iteration among its incoming edge (depicted by the dotted backedge) using $w_b$. This value is added to the similarity value of $[a_1, a_2]$ for the current iteration.

After the similarity values for all node pairs have been computed for the current iteration, they are normalized via division by the highest similarity value of that iteration. It follows directly that after the normalization phase the highest similarity value is 1.

The iteration terminates when the difference between $\sigma^i_{all}$ and $\sigma^{i+1}_{all}$ is smaller than some convergence value or if the upper bound on the number of iterations is reached. For
Chapter 4. Matching Metrics and Algorithms

example, the iteration could terminate after 500 iterations or as soon as the difference between the similarity values of two iterations $|\sigma^i([x,y]) - \sigma^{i+1}([x,y])|$ is smaller than 0.05 for each node pair.

**Correspondences Creation**  With the end of the fixpoint calculation the final similarity values for all combination of nodes are known. For the pairs with the highest similarity values correspondences and a matching can now be established. Here domain specific filter criteria can be applied to establish a correct matching. The criteria that are used by Madhusudan et al. are not stated in their publication. One example for a filter criterion could be that correspondences can be created only between nodes with the same type.

**Initial Similarity**  Different functions are used to compute initial similarity values. For activities the string similarity of their names is used. It returns a value between 0 and 1 that reflects how similar the names are (see Section 4.2.2). For gateways pairs of the same type the value 1 is assigned. Gateway pairs with differing types are assigned the value 0. Loops, subprocesses, start, and termination nodes are not mentioned in the publication by Madhusudan et al.

**Advantages and Drawbacks**

Madhusudan et al. see the advantage of the similarity flooding algorithm in its ability to cope with node labeling mismatches between both process models and missing or additional elements in one of the two models [37]. We believe that its iterative fixpoint computation is a huge advantage. It enables the algorithm to overcome the cyclic dependency in the similarity metric used (the similarity value for a node pair depends on the similarity of its adjacent node pairs). If the approach were extended to consider other (possibly non-basic) attributes for computing similarity values, cyclic dependencies there could also be mastered.

Nevertheless, there are different drawbacks concerning this approach. One is that due to the way it works the algorithm takes the topology of the process model stronger into account than the initial similarity values. Its resistance to “labeling mismatches” leads to the problem that activities are potentially matched even though their names are completely different (compare LR 3). It also bears the problem that node pairs with many edges tend to receive a high similarity value. However, the number of edges of a node pair depends purely on the topology of the two processes and not, e.g., only on the direct edges of the involved nodes. The algorithm does not support containment-based metrics, because the hierarchical decomposition of the graph is not taken into account at all. As another big disadvantage it is difficult to adapt the algorithm with respect to specific requirements, because it is based on the fixpoint computation for the whole model.

The algorithm might perform well when searching for the most similar model in a repository (as done by Madhusudan et al.), but it does not when used to match models
to detect, e.g., moved elements. In addition, start, end, subprocess, and loop nodes are not supported yet.

For this reason the similarity flooding is not practical for us since it would fulfill the requirements presented in Section 3.3 only indirectly and coincidentally. Adjusting the approach such that the requirements are fulfilled is difficult and error-prone as the fixpoint calculation can be influenced only indirectly by changing parameters for the single iteration.

### 4.2 Matching Metrics

In Section 2.5.2 different approaches to model matching were presented. Based on the results of the discussion in Section 3.3.3 we will use a similarity-based matching approach. For similarity-based matching we have to define the functions that compute similarity values for model elements pairs in a requirements-conforming way.

As we use a similarity-based approach, we have to define requirements-conforming compare functions to compute the similarity values for model element pairs. The element pairs with the highest similarity values will then constitute the final matching. The compare functions have to be based on model attributes (metrics) that conform to the requirements.

In the last section we discussed the advantages and drawbacks of the two presented similarity-based algorithms from related work. We explained that we cannot directly apply them for change detection.

Although we cannot reuse the whole algorithms, the functions to compute similarity values for model element pairs can be reused and extended. We call the measurement functions for the similarity values matching metrics or similarity metrics. In this section we describe and define the matching metrics of the compare-functions we use in our similarity-based algorithm. They are partly based on the algorithms presented in the previous section. The commonalities and differences between these algorithms and our approach will be discussed in Section 4.3.11.

In the following first the metrics based on basic attributes will be described: ID-based similarity and name similarity. Afterwards, metrics based on non-basic attributes are presented. The metrics are either derived from existing model matching approaches (e.g. name similarity) or developed from scratch (e.g. common neighborhood similarity).

#### 4.2.1 ID-based Similarity

Requirements HR 2 and LR 2 state that model elements should be matched by their IDs if possible. This static identity-based approach has to be combined with a similarity-based algorithm for all other elements. For the fulfillment of these requirements we could design a hybrid matching algorithm which would consist of two separate algorithms: we could first run a static identity-based matching algorithm and a similarity-based one afterwards. We use a different approach instead: similarity-based matching can incorporate static identity-based matching by defining a comparison-function that is based on the IDs of model elements.
Using such a metric simplifies the overall algorithm since it remains completely similarity-based. This approach is also more flexible because it gives us more freedom in the design of the algorithm. In Definition 14 we define a similarity function that assigns the value 1 to two elements with the same ID and 0 otherwise.

**Definition 14** (ID-based Similarity). Let $me_1 \in N_1 \cup E_1$ and $me_2 \in N_2 \cup E_2$ be two model elements from two workflow graphs $WFG_1 = (N_1, E_1)$ and $WFG_2 = (N_2, E_2)$. Let $i$ be the unique identifier of $me_1$ and $j$ be the unique identifier of $me_2$. The ID-based similarity of $me_1$ and $me_2$, denoted $idSim(me_1, me_2)$ is:

$$idSim(me_1, me_2) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

### 4.2.2 Name Similarity

Activities should be matched based on their names (see requirement LR 4). The naive approach would be to use them as unique identifiers and check for their equality. A more advanced approach is not to require that the name of two elements is identical but merely similar. This way the name attribute can be used as a metric that fulfills requirement LR 11.

There are many different ways to compute such a similarity. We will describe the (normalized) Levenshtein-distance [32] and $n$-grams [3] here. The Levenshtein-distance is used, e.g., in [50] and denotes the minimal number of operations necessary to transform one string into another. Three operations are allowed: the insertion, the deletion and the substitution of a character. The number of operations can be computed efficiently using dynamic programming and reflects the similarity of both strings. For example, for the two strings ‘Check Claim’ and ‘Check Claims’ one operation is required to transform the first string into the second (insert ‘s’ at the end). For the strings ‘Check Claim’ and ‘Claim Checking’ ten operations are required, although the difference in string size is only three. A different approach to compute the similarity between names is the use of $n$-grams, for which the name strings are divided into character groups of size 2, 3 or in general $n$. For example, the set of 3-grams for the string ‘Check’ is {‘Che’, ‘hec’, ‘eck’}. The $n$-grams similarity of two strings is computed using the set size $c$ of the sets containing common $n$-grams and the size $a$ of the set containing all $n$-grams of both words. The similarity is denoted by the fraction of the two values: $c/a$. For example, the set with (distinct) 3-grams for the two strings ‘Check Claim’ and ‘Claim Checking’ has fifteen elements. Six of these elements are 3-grams of both strings. Consequently, their 3-grams similarity is $6/15 = 0.4$. Compared to the Levenshtein-distance the $n$-grams technique is more robust against spelling mistakes and the reordering of characters in names, however, its computation is more complex.

These similarity values work on the syntactic level as they only compare character groups. To capture the semantic level, word lists with synonyms or ontologies, e.g. WordNet [2], can be used. Here two strings are equal if they are a synonym of each other. Capturing the semantic level is especially interesting for the third use case, which
Chapter 4. Matching Metrics and Algorithms

was presented in Section 3.1.3. It is likely that the activity names in a reference process or the process of another company are not the same as in the own process. Instead, the use of synonyms is probable.

The use of wordlists or ontologies, however, is less important for the first two use cases where models with common modeling history or subsequent model versions are compared. Here, usually only few names are changed and it is less likely that different names for the same elements are chosen.

For our purpose we measure the similarity between activity names using a syntactic string similarity measure, because that is sufficient for two out of the three use cases. Furthermore, checking for semantically equal name strings is not in the focus of this thesis. However, it is a well-established field of research with many publications, e.g. [6, 36]. Extending the name similarity metric to also consider word lists and ontologies is always possible and remains for future work.

We will use the Levenshtein-distance as our name similarity measure due to its sufficiency for our application and low computational complexity. We use a modification of the original measure called normalized Levenshtein-distance [11], which returns a value between 0 and 1. A value of 0 means that the compared names are completely different. A value of 1 means that they are equal. The exact definition is given in Definition 15.

**Definition 15** (String Edit Distance [32], Name Similarity [11]). Let \( n_1 \) and \( n_2 \) be two nodes with \( s \) being the name string of \( n_1 \) and \( t \) the name string of \( n_2 \). Let \( |x| \) represent the number of characters in a string \( x \). The string edit distance of \( s \) and \( t \), denoted \( ed(s, t) \), is the minimal number of atomic string operations necessary to transform \( s \) into \( t \) or vice versa. The atomic string operations are: Insertion of a character, deletion of a character or substitution of a character. The name similarity of \( n_1 \) and \( n_2 \), denoted \( nameSim(n_1, n_2) \), is:

\[
nameSim(n_1, n_2) = 1.0 - \frac{ed(s, t)}{\max(|s|, |t|)}
\]

4.2.3 Contained Correspondences Similarity

There are two kinds of nodes that encapsulate other elements. First, opening and closing loop or subprocess nodes of a workflow graph form pairs that *enclose* other model elements. Second, PST fragments directly *contain* other model elements. A similarity value for these nodes can be computed based on the matching of elements that they encapsulate as it is required by requirements LR 9 and LR 8.

Every pair \( n, n' \) of opening and closing loop or subprocess nodes of a workflow graph is directly contained by a parent fragment \( f^2 \). Thus, the opening node \( n \) of the pair is the entry-node of \( f \) and the closing node \( n' \) is the exit-node of \( f \). The upper workflow graphs of Figures 3.7 and 3.8 provide examples. There the loop and subprocess nodes

\(^2\)this is not true for the original decomposition if the opening and closing nodes enclose only a sequence of elements. We use a slightly adapted decomposition that always creates the encapsulating fragment \( f \).
after the two activities ‘A’ are opening nodes of the shaded fragment. The loop and subprocess nodes before the two activities ‘D’ are closing subprocess nodes of the shaded fragment.

After having established a fragment matching we can match loop and subprocess nodes of two workflow graphs by using their parent fragment correspondence. We simply have to create correspondences for the entry- and exit-nodes of these fragments. As the matching of loop and subprocess nodes can be derived from the matching of their parent fragment, we concentrate on fragment matching in the following.

The idea is that two fragments $f_1$ and $f_2$ that each contain a set of elements $C_1$ or $C_2$, are more similar if many elements in $C_1$ and $C_2$ have a correspondence to each other. The metric is presented in Definition 16. The function $\text{matched}(n)$ denotes the set of subtree leaf node correspondences in general and the function $\text{common}(n_1, n_2)$ denotes the set of subtree leaf node correspondences of nodes that are matched to the opposite subtree. Thus, $\text{common}(n_1, n_2) \subseteq \text{matched}(n_1)$ and $\text{common}(n_1, n_2) \subseteq \text{matched}(n_2)$.

**Definition 16** (Contained Correspondences Similarity). Let $\text{PST}(WFG_1)$ and $\text{PST}(WFG_2)$ be the process structure trees of the workflow graphs $WFG_1 = (M_1, E_1)$ and $WFG_2 = (M_2, E_2)$ and let $n_1 \in M_1$ and $n_2 \in M_2$ be two nodes. Let $N_1 \subseteq M_1$ and $N_2 \subseteq M_2$ be the sets of leaf nodes of the two trees and furthermore, let $M \subseteq N_1 \times N_2$ be a matching of leaf nodes. Let $\text{leafs}(n)$ denote the set of leaf nodes in the subtree of $n$.

Then the contained correspondences similarity of the nodes $n_1$ and $n_2$, denoted $\text{containedSim}(n_1, n_2)$, is defined as follows:

$$\text{matched}(n) = \{ (\nu_1, \nu_2) | (\nu_1, \nu_2) \in M, \nu_1 \in \text{leafs}(n) \lor \nu_2 \in \text{leafs}(n) \}$$

$$\text{common}(n_1, n_2) = \{ (\nu_1, \nu_2) | (\nu_1, \nu_2) \in M, \nu_1 \in \text{leafs}(n_1) \land \nu_2 \in \text{leafs}(n_2) \}$$

$$\text{containedSim}(n_1, n_2) = \frac{|\text{common}(n_1, n_2)|}{\max\{|\text{matched}(n_1)|, |\text{matched}(n_2)|\}}$$

Figure 4.3a presents an example. It illustrates the correspondences between elements contained in the depicted fragments. The fragments $f_Z$ and $f_X$ share three correspondences (out of four) and the $f_Y$ and $f_W$ share two (out of three). Figure 4.3b shows the resulting similarity values for the corresponding fragments in the PSTs of the workflow graphs of Figure 4.3a.

The contained correspondences similarity concentrates on subtree leaves of the fragments $f, f'$ that are already matched. So it computes the fraction of leaf node correspondences that $f$ and $f'$ have in common and leaf node correspondences of $f$ and $f'$ in general. The idea behind this is that we ignore leaf nodes that were deleted or inserted. Considering only matched elements allows us to use this metric in the use cases, where parts of the leaf node matchings were established via IDs but others were not, as in Use Cases 1 and 2.

This metric was inspired by the algorithm of Chawathe et al., which was presented in Section 4.1.1. The commonalities and differences to our metric and algorithm are discussed in Section 4.3.11.
4.2.4 Depthsum of Nodes

The depth of a node in the PST is an attribute that can be used as an additive measure when several correspondences have the same similarity value. It helps to identify the best correspondences among these to create the matching. The depth in the PST is taken into account during a partial fragment matching (see Section 4.3.3), which is a preliminary matching based on incomplete information. The matching metric that we use is the depthsum of two nodes as formalized in Definition 17.

**Definition 17 (Depthsum of Nodes).** Let \( \text{PST}(WFG_1) \) and \( \text{PST}(WFG_2) \) be the process structure trees of the workflow graphs \( WFG_1 = (N_1, E_1) \) and \( WFG_2 = (N_2, E_2) \) and let \( n_1 \in N_1 \) and \( n_2 \in N_2 \) be two nodes. Let \( \text{depth}(n) \) denote the depth of a node \( n \) in a process structure tree \( \text{PST} \) which is the length of the shortest path from the root of \( \text{PST} \) to \( n \). Then the \( \text{depthSum}(n_1, n_2) \) of nodes \( n_1 \) and \( n_2 \) is defined as follows:

\[
\text{depthSum}(n_1, n_2) = \text{depth}(n_1) + \text{depth}(n_2)
\]

Figure 4.4 illustrates the case where this metric is necessary. On the left, fragments of two workflow graphs are shown. As depicted only one correspondence for the two activities is created at this point. The matching of the gateways is not known yet. The right shows the fragments from the workflow graphs on the left in their PSTs and with their similarity values. The fragments can be matched to each other in two ways: \((f_Z, f_X)\) and \((f_Y, f_W)\) or \((f_Z, f_W)\) and \((f_Y, f_X)\). Both are valid with respect to the similarity values, which are equal because the content of the outer fragments \( f_Z \) and \( f_X \) is not matched yet, but only one matching is correct. Using the depth in the PSTs as an additional metric allows us to establish an order for the fragment pairs and thereby create the correct correspondences for the fragments. Cases that require this metric are more likely to occur in the third use case than in the other two, where additional ID information can be leveraged and more elements are matched early in the matching process. Thus, this matching metric is required with respect to a correct fragment matching (LR 9) and correct matching in general (HR 1).

Figure 4.3: Two workflow graph snippets with two fragments each and contained correspondences and their fragment similarity values: (a) the fragments and their contained correspondences, (b) the fragment correspondences computed based on their similarity values.
4.2.5 Entry- and Exit-Node Similarity

The entry- and exit-node similarity is required by the gateway matching presented in Section 4.3.4 to fulfill requirement LR 5. It states that in process models pairs of gateways often fully enclose parallel or alternative sequences of nodes. These pairs should be matched to corresponding gateway pairs in the other model-based on the enclosed elements. If the two pairs of the two workflow graphs enclose enough corresponding elements a correspondence should be created for the two opening (decision or fork) and and the two closing (merge or join) nodes of the gateway pairs.

The entry- and exit-node similarity is also used for the matching of loop and subprocess node pairs of workflow graphs (see Sections 4.2.3, 4.3.7 and 4.3.8).

The metric is based on the entry- and exit-node property of fragments (see Section 2.3.3). The entry-node similarity for two fragments \( f_1 \) and \( f_2 \) states whether their entry-nodes have the same type. The exit-node similarity for two fragments \( f_1 \) and \( f_2 \) states whether their exit-nodes have the same type. Combined with additional conditions, this metric is sufficient to match the appropriate entry- or exit-nodes. The metric is formalized in Definition 18.

**Definition 18** (Entry-Node Similarity, Exit-Node Similarity). Let \( PST(WFG_1) \) and \( PST(WFG_2) \) be the process structure trees of the workflow graphs \( WFG_1 = (N_1, E_1) \) and \( WFG_2 = (N_2, E_2) \). Let \( F_1 \) and \( F_2 \) be the set of fragments of \( PST(WFG_1) \) and \( PST(WFG_2) \). Let \( f_1 \in F_1 \) and \( f_2 \in F \) be fragments. Let \( t_1 \) denote the type of the entry-node of \( f_1 \) and \( t_2 \) the type of the entry-node of \( f_2 \). If the entry-node \( n_1 \) of \( f_1 \) is a gateway, then \( t_1 \) is the mapped type \( t_G(n_1) \). If the entry-node \( n_2 \) of \( f_2 \) is a gateway, then \( t_2 \) is the mapped type \( t_G(n_2) \). The entry-node similarity of two fragments, denoted \( entrySim(f_1, f_2) \), is defined as follows:

\[
entrySim(f_1, f_2) = \begin{cases} 
1 & \text{if } t_1 = t_2 \\
0 & \text{otherwise} 
\end{cases}
\]
Let \( tx_1 \) be the type of the exit-node of \( f_1 \) and let \( tx_2 \) be the type of the exit-node of \( f_2 \). If the entry-node \( n_1 \) of \( f_1 \) is a gateway, then \( t_1 \) is the mapped type \( t_G(n_1) \). If the entry-node \( n_2 \) of \( f_2 \) is a gateway, then \( t_2 \) is the mapped type \( t_G(n_2) \). The exit-node similarity \( \text{exitSim}(f_1, f_2) \) is then defined similar to the entry-node similarity as:

\[
\text{exitSim}(f_1, f_2) = \begin{cases} 
1 & \text{if } tx_1 = tx_2 \\
0 & \text{otherwise.}
\end{cases}
\]

### 4.2.6 Common Neighborhood Similarity

The common neighborhood similarity is a matching metric that is based on the neighboring nodes of two nodes as required by requirements LR 6 and LR 7. The metric takes into account neighboring nodes of the elements that are compared, even if they have different positions and distances in both workflow graphs. Thereby it also fulfills requirement LR 11, which states that the matching criteria should not be too strict.

Consider the example given in Figure 4.5a. It shows snippets of two workflow graphs. The two gateways in the middle are somewhat similar, because they are surrounded by corresponding activities. Still, the concrete relationships differ as the nodes have different distances (e.g. activities \( E \) and \( E' \)) and/or directions (e.g. activities \( C \) and \( C' \)) in both workflow graphs. By direction we mean whether a node is one of the predecessors (upstream) or one of the successors (downstream) of another node. To match the two depicted gateways the used similarity measure must be capable of taking into account changing distances and directions of the neighboring corresponding nodes.

This is what the common neighborhood similarity does. For the two nodes \( n_1 \in M_1 \) and \( n_2 \in M_1 \) under comparison from the two workflow graphs \( WFG = (M_1, E_1) \) and

![Figure 4.5](image-url)
The metric takes into account the sets of matched nodes $X_1 \subseteq M_1$ and $X_2 \subseteq M_2$ up to distance $N$ from $n_1$ and $n_2$ respectively. In Figure 4.5a the set of nodes for distance $N = 2$ for the upper workflow graph is $X_1 = \{A, B, D, E\}$. The metric measures the ratio of nodes in $X_1$ and $X_2$ that have a correspondence to each other compared to the overall size of $X_1$ and $X_2$. The correspondences are measured among two different dimensions: quality and distance. Three different quality types are distinguished. We use Figure 4.5a to describe the three different types. The activities $A$ and $A'$ have the same distance and direction with respect to the two control nodes (type 1). Activities $B$ and $B'$ have the same direction but a slightly different distance relative to the control nodes (type 2). Same is true for the activities $D$ and $D'$ as well as $E$ and $E'$. Activities $C$ and $C'$ have opposite directions and different distances (type 3) to the compared gateway nodes.

Correspondences with higher quality contribute stronger to the similarity value: correspondences with quality type 1 are valued more than correspondences with type 2. Type 2 correspondences are valued more than the ones of type 3. Type 1 correspondences are a special case for type 2 correspondences, which in turn are special cases for type 3 correspondences. The correspondence types subsume each other: let $C$ be the set of all correspondences in the neighborhood of two nodes under comparison, let $corr_{type1}$ be the set of type 1 correspondences, let $corr_{type2}$ be the set of type 2 correspondences, and let $corr_{type3}$ be the set of type 3 correspondences, then $corr_{type1} \subseteq corr_{type2} \subseteq corr_{type3} = C$. Every time type 2 correspondences are counted, type 1 correspondences are subsumed. This is one way to give correspondences of higher quality additional weight. Its distance also influences how much a correspondence contributes to the similarity value. The higher its distance is, the less a correspondence is taken into account.

The diagram in Figure 4.5b exemplifies the influence of distance and correspondence type. It shows the distances of activities of the upper workflow graph with respect to the decision node $d1$ ordered by their correspondence type. As illustrated, node correspondences add less to the similarity if they are more distanced from the compared nodes. They are also weighted less with decreasing quality. For the example, correspondence $(A, A')$ is valued more than correspondence $(C, C')$.

The idea of basing the similarity of model elements on correspondences in their neighborhood is related to the idea behind the similarity flooding algorithm that was presented in Section 4.1.2. The commonalities and differences between our algorithm and similarity flooding are discussed in Section 4.3.11.

Before the definition of the common neighborhood similarity is presented, additional auxiliary definitions will be given. The common neighborhood similarity is based on matched elements within a certain distance from the compared nodes. In the following, we define distance measures that will be used in the definitions that follow thereafter.

**Definition 19** (Distance of Nodes, In-Distance, Out-Distance, Undirected Distance). Let $n_1, n_2 \in M$ be two nodes of the workflow graph $WFG = (M, E)$. Let $I$ be the set of all paths from $n_1$ to $n_2$ in the opposite direction of the control flow, so the path along the predecessors of $n_1$. Let $O$ be the set of all paths from $n_1$ to $n_2$ in the direction of
the control flow, so the paths along the successors of \( n_1 \). Let \(|p|\) denote the length of the path \( p \).

Then the in-distance of node \( n_1 \) to its predecessor \( n_2 \), denoted by \( D_i(n_1, n_2) \) is defined as the shortest path from \( n_1 \) to \( n_2 \). Similarly, the out-distance of node \( n_1 \) to the successor \( n_2 \) is defined as \( D_o(n_1, n_2) \). We additionally define the undirected distance \( D_u(n_1, n_2) \):

\[
D_i(n_1, n_2) = \begin{cases} 
\infty & \text{if } I = \emptyset \\
|p| \text{ with } p \in I, \forall p' \in I : |p| \leq |p'| & \text{else.}
\end{cases}
\]

\[
D_o(n_1, n_2) = \begin{cases} 
\infty & \text{if } O = \emptyset \\
|p| \text{ with } p \in O, \forall p' \in O : |p| \leq |p'| & \text{else.}
\end{cases}
\]

\[
D_u(n_1, n_2) = \min\{D_i(n_1, n_2), D_o(n_1, n_2)\}
\]

For the nodes in the upper workflow graph of the figure we have the following distances: \( D_i(d1, C) = 3 \), \( D_o(d1, B) = 1 \) and \( D_i(d1, B) = \infty \).

As stated above we distinguish three different correspondence types for neighboring nodes of the two nodes under consideration. In the following we present and define three different auxiliary matching metrics which will be aggregated into the single metric of common neighborhood similarity later on:

1. directed identical distance similarity (\( \alpha - \text{Similarity} \))

2. directed distance similarity, (\( \beta - \text{Similarity} \))

3. undirected distance similarity, (\( \gamma - \text{Similarity} \))

The \( \alpha - \text{Similarity} \) captures correspondences of type 1, the \( \beta - \text{Similarity} \) captures correspondences of type 2 and the \( \gamma - \text{Similarity} \) computes a similarity value for correspondences of type 3.

All three metrics have a similar structure. For each distance \( d \) up to the maximal considered distance \( N \) from the compared node pair they compute a fraction. One component of the fraction is the number of nodes \( \text{nodes}_{\text{corr},d} \) with distance \( d \) that have correspondences of the respective type (depending on the matching metric). The second component is the number of all nodes \( \text{nodes}_{\text{all},d} \) with distance \( d \) from the compared node pair that have (any) correspondence. This fraction \( \frac{\text{nodes}_{\text{corr},d}}{\text{nodes}_{\text{all},d}} \) is computed for every distance \( d \) up to the maximal distance \( N \). The sum of these fractions is then aggregated into a single value, where fractions with smaller distance are valued more. This is achieved using weight factors \( w_d \) that decrease with growing distance \( d \): 

\[
\frac{\sum_{d=1}^{N} \text{nodes}_{\text{corr},d} \cdot w_d}{\sum_{d=1}^{N} w_d} \text{ with } w_d > w_{d+1}.
\]
Directed Identical Distance Similarity

The directed identical distance similarity is defined in Definition 20. It describes the ratio of nodes \( x \in M_1, y \in M_2 \) in the neighborhood of the two compared nodes \( n_1 \in M_1, n_2 \in M_2 \) that are matched to each other and that have the same in- or out-distance \( d \leq N \) to \( n_1 \) and \( n_2 \) respectively (type 1 correspondences). We define the similarity \( \alpha_{in} - Sim_{d}(n_1, n_2) \) for the distance \( d \) to the nodes under consideration. We then aggregate the \( \alpha_{in} \)-Similarity values for all distances (up to \( N \)) into a single value. We use \( \text{weight}_{\alpha} = w^d \) with \( w < 1 \) as the factor that reduces the weight of more distanced matching ratios such that close nodes have a higher impact on the similarity value. If the more distanced ratios should be taken into account stronger, the weight could be increased or, e.g., a polynomial factor could be used: \( \text{weight}_{\alpha} = w \cdot d \) with \( w < 1 \).

**Definition 20** (Directed Identical Distance Similarity, \( \alpha \)-Similarity). Let \( n_1 \in M_1 \) and \( n_2 \in M_2 \) be nodes from two workflow graphs \( WFG_1 = (M_1, E_1) \) and \( WFG_2 = (M_2, E_2) \). Let \( M \subseteq M_1 \times M_2 \) be the Matching of nodes and let \( \max(|S|, |S'|) \) denote the size of the bigger set: \( \max(|S|, |S'|) \). Let \( \text{weight}_{\alpha} \) be a weight factor and let \( N \in \mathbb{N} \) denote the maximal considered distance from \( n_1, n_2 \).

Then the directed identical distance similarity \( \alpha - Sim(n_1, n_2, N) \) is defined as follows:

\[
\alpha_{in} - Sim_{d}(n_1, n_2) = \frac{|\{(x, y)\} |(x, y) \in M, D_1(n_1, x) = D_1(n_2, y) = d|}{\max\{\{|(i, k)\} |(i, k) \in M, D_1(n_1, i) = d\}, \{|(l, j)\} |(l, j) \in M, D_1(n_2, j) = d\}}
\]

\[
\alpha_{out} - Sim_{d}(n_1, n_2) = \frac{|\{(x, y)\} |(x, y) \in M, D_1(n_1, x) = D_1(n_2, y) = d|}{\max\{\{|(i, k)\} |(i, k) \in M, D_1(n_1, i) = d\}, \{|(l, j)\} |(l, j) \in M, D_1(n_2, j) = d\}}
\]

\[
\alpha - Sim(n_1, n_2, N) = \frac{\sum_{d=1}^{N} \alpha - Sim(n_1, n_2) \cdot \text{weight}_{\alpha}}{\sum_{d=1}^{N} \text{weight}_{\alpha}}
\]

For the nodes in Figure 4.5 we compute for example the following values:

\( \alpha_{out} - Sim_1(d_1, d_2) = \frac{|\{(A, A')\} |}{\max\{\{|(E)|, \{|(A', E')|\}\}} = \frac{1}{2} \)

\( \alpha_{in} - Sim_1(d_1, d_2) = \frac{|\emptyset|}{\max\{\{|(E)|, \{|(A', E')|\}\}} = 0 \)

\( \alpha_{out} - Sim_2(d_1, d_2) = \frac{|\{(A, A')\} |}{\max\{\{|(E)|, \{|(A', E')|\}\}} = \frac{1}{3} \)
Directed Distance Similarity

The second similarity metric is the similarity with respect to corresponding nodes in the neighborhood of the two compared nodes that have a different in- our out-distance in both workflow graphs (type 2 correspondences). The correspondence only adds to the similarity value if both nodes are within some well-defined interval from the compared pair. Again, we first define the similarity for distance $d$ only and then derive a formula for the general directed distance similarity for nodes within a maximal distance $N$. As for the identical distance similarity, we use an exponential weight factor that reduces the weight of more distanced correspondences: $\text{weight}_\beta = w^d$ with $w < 1$. The directed distance similarity is formally described in Definition 21.

**Definition 21** (Directed Distance Similarity, $\beta$ – Similarity). Let $n_1 \in M_1$ and $n_2 \in M_2$ be nodes from two workflow graphs $WFG_1 = (M_1, E_1)$ and $WFG_2 = (M_2, E_2)$ . Let $M \subseteq M_1 \times M_2$ be the Matching of nodes and let $\max(S, S')$ denote the size of the bigger set: $\max(|S|, |S'|)$. Let $\text{weight}_\beta$ be a weight factor and let $N \in \mathbb{N}$ denote the maximal considered distance from $n_1, n_2$.

Then the directed similarity distance $\beta – \text{Sim}(n_1, n_2)$ is defined as follows:

$$
\beta_{\text{in}} – \text{Sim}_d(n_1, n_2) = \frac{|\{(x, y) | (x, y) \in M, D_i(n_1, x) \leq d \land D_i(n_2, y) \leq d\}|}{\max(|\{(i, k) | (i, k) \in M, D_i(n_1, i) \leq d\}, \{(l, j) | (l, j) \in M, D_i(n_2, j) \leq d\})}
$$

$$
\beta_{\text{out}} – \text{Sim}_d(n_1, n_2) = \frac{|\{(x, y) | (x, y) \in M, D_o(n_1, x) \leq d \land D_o(n_2, y) \leq d\}|}{\max(|\{(i, k) | (i, k) \in M, D_o(n_1, i) \leq d\}, \{(l, j) | (l, j) \in M, D_o(n_2, j) \leq d\})}
$$

$$
\beta – \text{Sim}_d(n_1, n_2) = \frac{\beta_{\text{in}} – \text{Sim}_d(n_1, n_2) + \beta_{\text{out}} – \text{Sim}_d(n_1, n_2)}{2}
$$

$\beta – \text{Sim}_d(n_1, n_2, N) = \sum_{d=1}^{N} \frac{\beta – \text{Sim}_d(n_1, n_2) \ast \text{weight}_\beta}{\sum_{d=1}^{N} \text{weight}_\beta}$

For the nodes in Figure 4.5 we compute for example the following values:

- $\beta_{\text{in}} – \text{Sim}_1(d_1, d_2) = \frac{|E|}{\max(|E|, |E'|)} = 0$
- $\beta_{\text{in}} – \text{Sim}_2(d_1, d_2) = \frac{|\{(E, E'), (D, D')\}|}{\max(|D, E|, |D'|, |E'|)} = 1$

Undirected Distance Similarity

The third similarity metrics resembles the second one. The only difference is that the direction of the distance for two nodes in a correspondence is ignored (type 3 correspondences). As for the two other similarity metrics, we first define a metric for the distance $d$ only and then derive a general formula for the general undirected distance similarity for nodes within a maximal distance $N$. 

Chapter 4. Matching Metrics and Algorithms

64
Again, we use an exponential weight factor that reduces the weight of more
distanced matching ratios: \( weight_\gamma = w^d \). The formal definition of the undirected distance
similarity is given in Definition 22.

**Definition 22** (Undirected Distance Similarity, \( \gamma - \) Similarity). Let \( n_1 \in M_1 \) and
\( n_2 \in M_2 \) be nodes from two workflow graphs \( WFG_1 = (M_1, E_1) \) and \( WFG_2 = (M_2, E_2) \).
Let \( M \subseteq M_1 \times M_2 \) be the Matching of nodes and let \( \max(S, S') \) denote the size of the
bigger set: \( \max\{|S|, |S'|\} \). Let \( weight_\gamma \) be a weight factor and let \( N \in \mathbb{N} \) denote the
maximal considered distance from \( n_1, n_2 \).

Then the undirected distance similarity \( \gamma - Sim(n_1, n_2, N) \) is defined as follows:

\[
\gamma - Sim(n_1, n_2, N) = \frac{|\{(x, y) | (x, y) \in M, D_u(n_1, x) \leq d \land D_u(n_2, y) \leq d\}|}{\max\{|\{(i, k) | (i, k) \in M, D_u(n_1, i) \leq d\}, |\{(l, j) | (l, j) \in M, D_u(n_2, j) \leq d\}|\}}
\]

For the nodes in Figure 4.5 we compute for example the following values:

- \( \gamma - Sim_1(d_1, d_2) = \frac{|\{(A, A') | (A, A') \in M, D_u(A, A') \leq d\}|}{\max\{|\{(A, B, E) | (A, B, E) \in M, D_u(A, B, E) \leq d\}|\}} = \frac{1}{3} \)
- \( \gamma - Sim_2(d_1, d_2) = \frac{|\{(A, A'), (B, B') | (A, A'), (B, B') \in M, D_u(A, A'), (B, B') \leq d\}|}{\max\{|\{(A, B, C, D) | (A, B, C, D) \in M, D_u(A, B, C, D) \leq d\}|\}} = 1 \)

**Common Neighborhood Similarity**

The common neighborhood similarity aggregates the three presented similarity values
into a single measure. Its definition is presented in Definition 23:

**Definition 23** (Common Neighborhood Similarity). Let \( n_1 \in M_1 \) and \( n_2 \in M_2 \) be nodes
from two workflow graphs \( WFG_1 = (M_1, E_1) \) and \( WFG_2 = (M_2, E_2) \). Let \( w_\alpha, w_\beta \) and
\( w_\gamma \) be weight factors and let \( N \in \mathbb{N} \) denote the maximal considered distance from \( n_1, n_2 \).

Then the common neighborhood similarity \( n - hoodSim(n_1, n_2, N) \) is defined as follows:

\[
n - hoodSim(n_1, n_2, N) = \frac{w_\alpha \cdot \alpha - Sim(n_1, n_2, N) + w_\beta \cdot \beta - Sim(n_1, n_2, N) + w_\gamma \cdot \gamma - Sim(n_1, n_2, N)}{w_\alpha + w_\beta + w_\gamma}
\]

### 4.2.7 Source- and Target-Node Similarity

In accordance to requirement LR 10 the similarity between edges is measured by looking
at their source and target nodes. We define the edge similarity as described by Definition 24: two edges have a similarity of 1 if their source and their target nodes correspond
to each other. Otherwise the similarity is 0.
Definition 24 (Source-Node and Target-Node Similarity). Let \( n_1, m_1 \in N_1 \) and \( n_2, m_2 \in N_2 \) be nodes from two workflow graphs \( WFG_1 = (N_1, E_1) \) and \( WFG_2 = (N_2, E_2) \). Let \( M \subseteq N_1 \times N_2 \) be a Matching of nodes and let \( e_1 = (n_1, m_1) \in E_1 \) and \( e_2 = (n_2, m_2) \) be two edges.

Then the source-node and target-node similarity of two edges, denoted \( \text{sourceTargetSim}(e_1, e_2) \), is defined as follows:

\[
\text{sourceTargetSim}(e_1, e_2) = \begin{cases} 
1 & \text{if } e_1 = (n_1, m_1) \land e_2 = (n_2, m_2) \land (n_1, n_2), (m_1, m_2) \in M \\
0 & \text{else.}
\end{cases}
\]

4.3 Matching Algorithm

Section 4.2 described the different matching metrics that fulfill the requirements presented in Section 3.3 and that shall be used by the matching algorithm. These metrics are not independent of each other and their computation bears the problem of a cyclic dependency. This cycle exists, because in order to compute a gateway-node matching using the entry- and exit-node similarity (see Section 4.2.5), a fragment matching has to be established beforehand. That in turn is based on the contained correspondences and, hence, also the contained gateways.

For this reason, the matching of workflow graphs and their fragments using the depicted metrics cannot be implemented in a straightforward manner. In the following we present a similarity-based matching algorithm for process models that is designed to overcome this problem. It is a stepwise approach that consecutively executes sub-algorithms which work on the partial matching that was computed by their predecessors. Algorithm 1 gives an overview of the sub-algorithms and the order in which they are invoked. It also shows for each algorithm in which section the algorithm is discussed. The main solution to overcome the cyclic dependency is the preliminary computation of the fragment matching (line 9). The fragment matching is then recomputed at the end (line 15). Table 4.2 shows which strategies the consecutively executed algorithms use to match the different element types.

<table>
<thead>
<tr>
<th>Element Type</th>
<th>Attribute</th>
<th>Neighborhood</th>
<th>Containment</th>
</tr>
</thead>
<tbody>
<tr>
<td>All elements (ID-based)</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Activities</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>prelim. Fragments</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Gateways (Frag.-based)</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Control Nodes (Neighb.-based)</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Fragments</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Loops</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Subprocesses</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Edges</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Matching order of element types and applied matching strategies.
**Algorithm 1** Overall Algorithm

- $M$ is the matching
- $N$ denotes a distance $N \geq 1$, $N \in \mathbb{N}$

5: $M \leftarrow \emptyset$

- see Section 4.3.1

5: $M \leftarrow \text{matchElementsBasedOnIDs}()$

5: $M \leftarrow \text{matchActivitiesBasedOnNameSimilarity}(M, \text{threshold}_{\text{activity}})$

- see Section 4.3.2

5: $M \leftarrow \text{matchFragmentsBasedOnContainedLeafs}(M, \text{threshold}_{\text{frg1}})$

- see Section 4.3.3

10: $M \leftarrow \text{matchEntryExitGatewayNodesOfWellStrucFragments}(M)$

- see Section 4.3.4

10: $M \leftarrow \text{matchControlNodesBasedOnCommonNeighbh}(M, N, \text{threshold}_{\text{nhop}})$

- see Section 4.3.5

15: $M \leftarrow \text{matchFragmentsBasedOnContainedLeafs}(M, \text{threshold}_{\text{frg2}})$

- see Section 4.3.6

15: $M \leftarrow \text{matchLoopsBasedOnContainedLeafs}(M)$

- see Section 4.3.7

15: $M \leftarrow \text{matchSubprocessesBasedOnContainedLeafs}(M)$

- see Section 4.3.8

20: $M \leftarrow \text{matchEdgesBasedOnSourceAndTarget}(M)$

- see Section 4.3.9

return $M$
In the following we will introduce each sub-algorithm. Figure 4.6 shows two slightly different workflow graphs for claim handling. As depicted, some elements are assigned matching IDs, others are not. We will use these workflow graphs as a running example to exemplify the matching process.

### 4.3.1 ID-based Matching

The first sub-algorithm that is executed creates correspondences based on IDs. It is assumed that only corresponding elements have the same ID and that the requirements are not violated by this, especially Requirement LR 1. Accordingly, elements with matching IDs are assumed to have the same type. Algorithm 2 shows a straightforward implementation of the ID-based matching.

**Algorithm 2 ID-based Matching**

- $ME_1, ME_2$ are the sets of model elements of the two workflow graphs under consideration.
- $me_1, me_2$ are model elements.
- $idSim()$ is defined in Definition 14.

5: $M \leftarrow \emptyset$

for all $me_1 \in ME_1$ do
  for all $me_2 \in ME_2$ do
    if $idSim(me_1, me_2) = 1$ then
      $M \leftarrow M \cup \{(me_1, me_2)\}$
  end if
end for
end for

return $M$

In Figure 4.6 the activities ’Inform Supervisor’, ’Issue Payment’, ’Reemburse Client’, and ’Write Accept. Letter’ from the upper and lower workflow graphs are assigned the same ID. Using this ID they can be matched with the ID-based matching algorithm. The same is true for the two termination nodes with the ID 23.

The runtime of this basic implementation is quadratic in the number of model elements: $O(|ME_1| \times |ME_2| \times t_{idSim})$. The ID-similarity for two elements $t_{idSim}$ can be determined in constant time. Thus, Performance-Requirement HR 4 is fulfilled.

### 4.3.2 Activity Matching

The activity matching is based on the matching metric presented in Section 4.2.2, which is based on the normalized Levenstein distance.

As shown in Algorithm 3, the similarity values for all combinations of unmatched activities between both processes are computed (line 9). Then a greedy algorithm is...
Figure 4.6: Two differing workflow graphs for the handling of claims.
applied to iteratively add those activity pairs to the matching whose activities were not matched before (as of line 12). The greedy algorithm starts top-down with the pair of the highest similarity and only considers pairs above a threshold value $\text{threshold}_{\text{activity}}$ that is an input parameter. Line 18 states that all pairs that contain one of the two activities of a correspondence are removed from the similarities set, as any activity can be matched at most once.

### Algorithm 3 Activity Matching

**Require:** $PM$ is a mapping between the elements of the processes as computed by Algorithm 2.

**Require:** $\text{threshold}_{\text{activity}}$ is the minimum acceptable degree of similarity.

$A_1, A_2$ are the sets of activities of the two processes under consideration

$a_1, a_2$ are activities

$nameSim()$ is defined in Definition 15

5: 
$M \leftarrow PM$

for all $a_1 \in A_1, (a_1, x) \notin M$ do

for all $a_2 \in A_2, (y, a_2) \notin M$ do

similarities $\leftarrow \{(a_1, a_2, nameSim(a_1, a_2))|a_1 \in A_1, a_2 \in A_2\}$

end for

end for

while similarities $\neq \emptyset$ do

select a $(a_1, a_2, s)$ from similarities such that

there does not exist an $(a_1', a_2', s')$ in similarities for which $s' > s$

15: 
similarities $\leftarrow$ similarities $\setminus \{(a_1', a_2', s')|(a_1', a_2', s') \in$ similarities, $a_1' \neq a_1, a_2' \neq a_2\}$

if $s > \text{threshold}_{\text{activities}}$ then

$M \leftarrow M \cup \{(a_1, a_2)\}$

similarities $\leftarrow \{(a_1', a_2', s')|(a_1', a_2', s') \in$ similarities, $a_1' \neq a_1, a_2' \neq a_2\}$

end if

20: end while

Table 4.3 shows the similarity values for the activities in Figure 4.6 that were not matched in the previous step. We assume $\text{threshold}_{\text{activity}}$ to be below 0.91. The algorithm starts with the pair of highest similarity when creating correspondences. Table 4.4 shows the order of activity correspondences that are created by this sub-algorithm. The pair ‘Fill in Payment Form’ & ‘Fill in Payment Form A’ has a similarity value of 0.91 and is for this reason added after the other two pairs, which have the higher similarity value 1.

The name similarity can be computed in $O(|A_1| \times |A_2| \times t_{nameSim})$ with $|A_1|$ and $|A_2|$ being the number of activities in both graphs. The computation of the activity name similarity for two tasks $t_{nameSim}$ is bounded by the square of the length of longest name in both models. Assuming a maximum length of the name strings of model elements it
### Table 4.3: Name-based similarity values of activities from Figure 4.6 that cannot be matched using IDs.

<table>
<thead>
<tr>
<th>Activity Name from 1st Model</th>
<th>Activity from 2nd Model</th>
<th>Similarity Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Compensate Third Party'</td>
<td>'Compensate Third Party'</td>
<td>1</td>
</tr>
<tr>
<td>'Compensate Third Party'</td>
<td>'Fill in Payment Form A'</td>
<td>$\frac{1}{22} = 0.05$</td>
</tr>
<tr>
<td>'Compensate Third Party'</td>
<td>'Archive Issue'</td>
<td>$\frac{3}{22} = 0.13$</td>
</tr>
<tr>
<td>'Compensate Third Party'</td>
<td>'Sent Transfer Confirmation'</td>
<td>$\frac{4}{26} = 0.02$</td>
</tr>
<tr>
<td>'Compensate Third Party'</td>
<td>'Call Client'</td>
<td>$\frac{5}{22} = 0.23$</td>
</tr>
<tr>
<td>'Fill in Payment Form'</td>
<td>'Compensate Third Party'</td>
<td>$\frac{2}{22} = 0.09$</td>
</tr>
<tr>
<td>'Fill in Payment Form'</td>
<td>'Fill in Payment Form A'</td>
<td>$\frac{20}{22} = 0.91$</td>
</tr>
<tr>
<td>'Fill in Payment Form'</td>
<td>'Archive Issue'</td>
<td>$\frac{4}{22} = 0.18$</td>
</tr>
<tr>
<td>'Fill in Payment Form'</td>
<td>'Sent Transfer Confirmation'</td>
<td>$\frac{4}{26} = 0.15$</td>
</tr>
<tr>
<td>'Fill in Payment Form'</td>
<td>'Call Client'</td>
<td>$\frac{6}{20} = 0.3$</td>
</tr>
<tr>
<td>'Sent Transfer Confirmation'</td>
<td>'Compensate Third Party'</td>
<td>$\frac{4}{26} = 0.15$</td>
</tr>
<tr>
<td>'Sent Transfer Confirmation'</td>
<td>'Fill in Payment Form A'</td>
<td>$\frac{4}{26} = 0.15$</td>
</tr>
<tr>
<td>'Sent Transfer Confirmation'</td>
<td>'Archive Issue'</td>
<td>$\frac{3}{26} = 0.12$</td>
</tr>
<tr>
<td>'Sent Transfer Confirmation'</td>
<td>'Sent Transfer Confirmation'</td>
<td>1</td>
</tr>
<tr>
<td>'Sent Transfer Confirmation'</td>
<td>'Call Client'</td>
<td>$\frac{5}{26} = 0.19$</td>
</tr>
</tbody>
</table>

Table 4.4: Order of created correspondences by Algorithm 3 based on the similarity values shown in Table 4.3.
can be treated as constant. Consequently, the overall runtime complexity is quadratic in the number of activities and justifiable with respect to requirement HR 4.

### 4.3.3 Preliminary Fragment Matching

The algorithm for the fragment matching is based on the metric described in Section 4.2.3. We use it to compute the content-based similarity of fragments.

As we are going to need the fragment matching for the matching of gateways of well-formed fragments (see Section 4.3.4), we have to compute the fragment matching in two steps. Here we compute the (mainly) activity-based fragment matching that matches (at least) all fragments that contain activities. By executing this algorithm again later in the matching process, we will compute the leaf-based fragment matching and additionally match fragments that do not contain any activities but control nodes. When an ID-based matching is possible (Use Cases 1 and 2) the situation is slightly different. In those cases we will already be able to match most of the fragments in this step, as there will already exist correspondences for other element types. We will describe the most general case, which is the case where only activities are already matched.

Algorithm 4 describes the algorithm for preliminary fragment matching. It is reused for the complete leaf-node-based fragment matching, which is described in Section 4.3.6, where additional matches for other node types will be taken into account. Except for slight changes, this algorithm is also used in the matching process of loops (see Section 4.3.7) and subprocesses (see Section 4.3.8).

First, the algorithm computes the similarity of all fragment pairs using the metric of contained correspondences described in Section 4.2.3 (line 7). Then a greedy algorithm is applied to iteratively add those fragment correspondences to the matching whose fragments have not been matched yet (as of line 8). It starts with the pair with the highest similarity and only considers pairs above the threshold.

Figure 4.7 shows the result for the application of this sub-algorithm. We assume that the leaf node correspondences were created by the preceding sub-aglorithms and that the threshold $\text{threshold}_{frg}$ is above 0.5. For example, the fragments $f_Z$ and $f_O$ are matched to each other based on the five common correspondences of the contained activities. In contrast, the fragments $f_R$ and $f_G$ or $f_R$ and $f_D$ are not matched to each other, because they only have half of the contained correspondences in common.

If there are several entries with equal similarity values then the element pair with the highest three-depth in terms of the depthsum metric (see Section 4.2.4) is added first. Figure 4.7 shows an example for the case, where this rule is necessary. All possible correspondences for $f_W, f_T$ and $f_L, f_K$ have the same similarity value (because all matched nodes that are contained are identical). Adding first the deepest fragment pair in the PST avoids creating the correspondences $(f_W, f_K)$ and $(f_T, f_L)$. The depthsum metric is necessary when nested fragments differ only by contained nodes that are unmatched. It avoids an arbitrary matching of fragments.

The statement in line 17 removes all pairs that contain one of the two fragments of a correspondence, as any fragment can be matched at most once.
Algorithm 4 Fragment Matching

Require: \( LM \) is a mapping between the leaf nodes of the processes, as computed by Algorithm 3.

Require: \( \text{threshold}_{frg} \) is the minimum acceptable degree of similarity.

\( F_1 \) and \( F_2 \) are the sets of fragments of the PSTs.  
\( \text{containedSim}() \) is defined in Definition 16  
\( \text{depthsum}() \) is defined in Definition 17

5:  
\[ M \leftarrow LM \]
\[ \text{similarities} \leftarrow \{(f_1, f_2, \text{containedSim}(f_1, f_2)) | f_1 \in F_1, f_2 \in F_2\} \]

while \( \text{similarities} \neq \emptyset \) do

\[ \text{select an } (f_1', f_2', s') \text{ from } \text{similarities} \text{ such that} \]

\[ s' > s, \text{ or} \]
\[ s' = s \text{ and } \text{depthsum}(f_1', f_2') > \text{depthsum}(f_1, f_2) \]

\[ \text{similarities} \leftarrow \text{similarities} \setminus (f_1, f_2, s) \]

if \( s > \text{threshold}_{frg} \) then

\[ M \leftarrow M \setminus \{(f_i, f_j) | (f_i, f_j) \in M, f_i = f_1 \lor f_j = f_2\} \]
\[ M \leftarrow M \cup \{(f_1', f_2')\} \]
\[ \text{similarities} \leftarrow \{(f_1', f_2', s') | (f_1', f_2', s') \in \text{similarities}, f_1' \neq f_1, f_2' \neq f_2\} \]

end if

end while

The runtime of this algorithm is \( O(\text{bfstime}_1 + \text{bfstime}_2 + |F_1| \times |F_2| \times t_{\text{containedSim}}) \). \( \text{bfstime}_i \) denotes the computation steps to determine the depth of the nodes of the respective workflow graph. It can be computed using breadth-first search, so \( O(\text{bfstime}_1 + \text{bfstime}_2) = O(2 \cdot |N| \times |E|) \) with \( |N| \) and \( |E| \) being the bigger number of nodes/edges of the respective workflow graphs. \( t_{\text{containedSim}} \) denotes the time to compute the contained correspondences similarity for two elements. The complexity of \( t_{\text{containedSim}} \) depends heavily on the implementation and data structures used. It can be reduced significantly by exploiting the tree structure and using appropriate data structures (e.g. hashmaps) and preprocessing steps. As a result of the aforementioned techniques, the overall runtime can be reduced to fulfill HR 4.

4.3.4 Entry-/Exit-Gateway Matching of Well-structured Fragments

Corresponding fragments share most of their activities due to the content-based strategy used for fragment matching. We use the following heuristic: for two well-structured corresponding fragments we can match their entry-nodes if these are gateways of the same type. Similarly, we match gateways of same type that are exit-nodes of these fragments. The intention behind this is that the gateways encapsulate many corresponding activities in a similar way. Hence, they should be matched as required by LR 5.
While this technique can be used for structured fragments, it does not work if at least one of the fragments is unstructured. Figure 4.8 shows an example where the use of this heuristic would match the wrong gateways. The figure shows two workflow graphs with the same activities but different structure. By Requirement LR 6 the merge nodes after the activity ‘Inform Supervisor’ should be matched to each other, because control nodes should be matched based on their neighboring nodes. Comparing the predecessors and successors of these merge nodes the requirement is fulfilled. However, using the parent fragment to match the control nodes we would match the merge node in the upper model to the last merge node of fragment \( f_U \) in the lower model (its exit-node).

This is due to the fact that unstructured fragments contain a fragment structure that is not as well-organized as in structured fragments. Hence, Requirement LR 5 is not applicable here.

Algorithm 5 shows how gateways of well-structured fragments are matched. For every correspondence in the matching it is checked whether its elements are well-structured fragments with unmatched entry- or exit-nodes (line 10). When this is the case, it is determined whether the entry-/exit-nodes of both fragments have the same type using the metric described in Section 4.2.5. Furthermore, it is checked if the entry-/exit-nodes are gateways. If they are, the correspondence of the nodes is added to the matching.

As an example, consider the fragment matching presented in Figure 4.7. Using the described sub-algorithm the decision and merge nodes of the corresponding fragments \( f_W \) and \( f_L \) are matched. \( f_Z \) and \( f_O \) are unstructured fragments and the fragments \( f_R \),
Algorithm 5 Entry-/Exit Gateway Node Matching of Well-structured Fragments

**Require:** $FM$ is the preliminary fragment matching, as computed by Algorithm 4.

wellstructured($f$) returns true if the fragment $f$ is well-structured as defined in Section 2.3.3.

isGateway($n$) returns true if the parameter $n$ is a gateway.

entryNode($f$) returns the entry-node of the fragment $f$.

exitNode($f$) returns the exit-node of the fragment $f$.

entrySim($n_1, n_2$) and exitSim($n_1, n_2$) are defined in Definition 18.

1. $M \leftarrow FM$
2. for all $(f_1, f_2) \in FM$ do
3.     if wellstructured($f_1$) and wellstructured($f_2$) then
4.         entry$_1 \leftarrow$ entryNode($f_1$)
5.         entry$_2 \leftarrow$ entryNode($f_2$)
6.         if entrySim($f_1, f_2$) = 1 and isGateway(entry$_1$) then
7.             if (entry$_1, x$) \notin M, (y, entry$_2$) \notin M then
8.                 $M \leftarrow M \cup \{(entry$_1$, entry$_2$)\}$
9.             end if
10.         end if
11.     end if
12.     exit$_1 \leftarrow$ exitNode($f_1$)
13.     exit$_2 \leftarrow$ exitNode($f_2$)
14.     if exitSim($f_1, f_2$) = 1 and isGateway(exit$_1$) then
15.         $M \leftarrow M \cup \{(exit$_1$, exit$_2$)\}$
16.     end if
17. end for
Chapter 4. Matching Metrics and Algorithms

Figure 4.8: A workflow graphs with two structured fragments $f_Y, f_X$ and a workflow graph with an unstructured fragment $f_U$.

$f_G$ and $f_D$ do not have a corresponding fragment. Hence, the entry and exit-nodes of these fragments are not considered by the sub-algorithm.

The entry-node and exit-node similarity can be computed in $O(|F_1| \times |F_2| \times t_{\text{compare}})$ with $t_{\text{compare}}$ denoting the check whether the entry- or exit-nodes can be matched. $t_{\text{compare}}$ can be computed in constant time. Thus, Performance-Requirement HR 4 is fulfilled.

4.3.5 Control Node Matching based on Neighborhood

Since it is not possible to match all gateways based on the heuristic described in Section 4.3.4, an additional strategy is used to match the remaining gateways and the other control node types based on the requirements LR 6 and LR 7. This strategy is based on the common neighborhood metric presented in Section 4.2.6.

Algorithm 6 shows the sub-algorithm. In general it uses the same basic algorithm that is used for the computation of the activity matching (Algorithm 3). The similarity value is computed for all control nodes of the same type that were neither matched by the heuristic in the previous step (see Section 4.3.4) nor by the ID-based matching (see Section 4.3.1). Then a greedy algorithm is used to add control node correspondences that have a similarity above the threshold to the matching (as of line 12). The greedy algorithm always picks the pair with the highest similarity. Then all pairs that contain
one of the two control nodes of the new correspondence are removed as any control node can be matched at most once (line 19).

**Algorithm 6** Common Neighborhood Control Node Matching

**Require:** $TM$ is a mapping between nodes of the processes as computed by previous Algorithms 2–5.

**Require:** $\text{threshold}_{\text{nhop}}$ is the minimum acceptable degree of similarity.

$M_1, M_2$ are the nodes of the two processes under consideration $c_{t,1}, c_{t,2}$ are control nodes of type $t$

neighborhoodSim() is defined in Definition 23.

5: $M \leftarrow TM$

for all $c_{t,1} \in M_1, (c_{t,1}, x) \notin M$ do

for all $c_{t,2} \in M_2, (y, c_{t,2}) \notin M$ do

$\text{similarities} \leftarrow \{ (c_{t,1}, c_{t,2}, \text{neighborhoodSim}(c_{t,1}, c_{t,2})) \}$

end for

end for

while $\text{similarities} \neq \emptyset$ do

select an $(c_{t,1}, c_{t,2}, s)$ from $\text{similarities}$ such that there does not exist an $(c'_{t,1}, c'_{t,2}, s')$

in $\text{similarities}$ for which $s' > s$

$\text{similarities} \leftarrow \text{similarities} \setminus (c_{t,1}, c_{t,2}, s)$

if $s > \text{threshold}_{\text{nhop}}$ then

$M \leftarrow M \cup \{ (c_{t,1}, c_{t,2}) \}$

$\text{similarities} \leftarrow \{ (c'_{t,1}, c'_{t,2}, s') | (c'_{t,1}, c'_{t,2}, s') \in \text{similarities}, c'_{t,1} \neq c_{t,1}, c'_{t,2} \neq c_{t,2} \}$

end if

end while

Figure 4.9 shows the matching for the running example after executing this subalgorithm. As depicted, the corresponding control nodes are matched up based on the correspondences in the neighborhood. For the first part of the workflow graphs the created correspondences are obvious as the closer neighborhood of corresponding control nodes is identical. For the termination nodes that are the very last elements of the workflow graphs, the closer neighborhood is not completely identical. If we chose inappropriate parameters, e.g. a threshold value for the threshold $\text{threshold}_{\text{nhop}}$ which is too high, these termination nodes would not be matched up.

The runtime is $O(|N_1| \times |N_2| \times t_{\text{nhop}})$ with $|N_1|$ and $|N_2|$ being the number of compared nodes and $t_{\text{nhop}}$ the time to compute their similarity $\text{neighborhoodSim}(n_1, n_2, N)$. The time to compute $\text{neighborhoodSim}(n_1, n_2, N)$ is mainly dominated by the amount $S$ of nodes in the $N$-distance neighborhood of each compared pair $n_1$ and $n_2$. The degree of nodes in a workflow graph is normally low and so is the number of nodes directly
connected to them. As a consequence, $S$ is usually comparably small and the overall runtime is sufficiently low with respect to requirement HR 4.

### 4.3.6 Leaf-Node-based Fragment Matching

After the control nodes are matched, the fragment matching Algorithm 4 is rerun. This time it also matches the fragments that do not contain any activities or IDs but control nodes. Since the algorithm adds information on matched control nodes, the fragment matching might change compared to the last time the algorithm was run.

For example, in Figure 4.9 for the previously unmatched fragments $f_R$ and $f_G$ a correspondence will be created, because the fork and join nodes are now matched.

### 4.3.7 Loop Node Pair Matching

Loop node pairs (the opening and closing nodes of a loop in workflow graph representation) can be matched using the existing fragment matching. Loop node pairs should be matched similar to fragments based on the leaf node correspondences they encapsulate. Because the two nodes of a loop node pair are always the entry- and exit-node of a fragment, they can be matched using their parent fragment correspondence. Like in the algorithm for gateways of well-structured fragments, only the entry- and exit-nodes of corresponding fragments have to be compared.
Chapter 4. Matching Metrics and Algorithms

Algorithm 7 shows the matching of loop nodes. For every fragment correspondence in the matching it is checked whether the entry- and exit-nodes are loop nodes. If this is the case, the corresponding nodes are matched.

**Algorithm 7** Loop Matching

**Require:** $FM$ is the final fragment matching, as computed by Algorithm 4 (described in Section 4.3.6).

1. $isLoop(n)$ returns true if the parameter $n$ is a loop node.
2. $entryNode(f)$ returns the entry-node of the fragment $f$.
3. $exitNode(f)$ returns the exit-node of the fragment $f$.

5: $entrySim(n_1, n_2)$ and $exitSim(n_1, n_2)$ are defined in Definition 18.

$$M \leftarrow FM$$

for all $(f_1, f_2) \in FM$ do

10: $entry_1 \leftarrow entryNode(f_1)$

11: $entry_2 \leftarrow entryNode(f_2)$

12: $exit_1 \leftarrow exitNode(f_1)$

13: $exit_2 \leftarrow exitNode(f_2)$

if $entrySim(f_1, f_2) = 1$ and $isLoop(entry_1)$ then

14: if $exitSim(f_1, f_2) = 1$ and $isLoop(exit_1)$ then

15: if $(entry_1, x) \notin M, (y, entry_2) \notin M, (exit_1, w) \notin M, (z, exit_2) \notin M$ then

16: $M \leftarrow M \cup \{(entry_1, entry_2)\}$

17: $M \leftarrow M \cup \{(exit_1, exit_2)\}$

18: end if

19: end if

20: end if

end for

The runtime of this sub-algorithm is similar to the one presented in Section 4.3.4, hence, the Requirement HR 4 is fulfilled.

### 4.3.8 Subprocess Node Pair Matching

For subprocess node pairs the same reasoning as for loop nodes can be applied (see Section 4.3.6). Thus, they are matched in a similar way. The only difference to Algorithm 7 is that in lines 13 and 14 it needs to be checked whether the nodes are subprocess nodes instead of checking for loop nodes.

For the workflow graphs in Figure 4.9 for example, the subprocess starting and ending nodes would be matched by this sub-algorithm based on the correspondence between their parent fragments $f_S$ and $f_H$. 
4.3.9 Edge Matching

The edges are matched based on the metric of corresponding source and target (see Section 4.2.7). Consequently, the edge matching algorithm is invoked as the last algorithm such that all possible correspondences between leaf nodes are already computed. Algorithm 8 shows how the edge matching is created. It pursues the same algorithmic idea as the one for activities.

First, the similarity values for all edge pairs are computed (line 7). Then a greedy algorithm is applied to iteratively add those edge correspondences to the matching, whose edges have not been matched yet (as of line 8). It starts top-down with the pair of the highest similarity and only considers pairs above the threshold. In line 14 all pairs that contain one of the two edges of a correspondence are removed, as any edge can be matched at most once.

Edges are matched whenever their source and target correspond to each other. In Figure 4.9, for example, the edges between the initial start and decision nodes would be matched. Also the edges with the targets 'Fill in Payment Form' and 'Fill in Payment Form A' are matched. In contrast, the edges with the final termination node as their target are not matched because their source nodes do not correspond to each other.

Algorithm 8 Edge Matching

Require: $TM$ is a mapping between nodes of the processes, as computed by previous node matching Algorithms 3–6.

$E_1, E_2$ are the sets of edges of the two processes under consideration

e_1, e_2$ are edges

$sourceTargetSim()$ is defined in Definition 24

5: $M ← ∅$

$similarities ← \{(e_1, e_2, sourceTargetSim(e_1, e_2)) | e_1 ∈ E_1, e_2 ∈ E_2\}$

while $similarities ≠ ∅$ do

10: select an $(e_1, e_2, s)$ from $similarities$ such that there does not exist an $(e'_1, e'_2, s')$ in $similarities$ for which $s' > s$

$similarities ← similarities \ \{(e_1, e_2, s)\}$

if $s >$ threshold then

15: $M ← M \cup \{(e_1, e_2)\}$

$similarities ← \{(e'_1, e'_2, s')(e'_1, e'_2, s') ∈ similarities, e'_1 ≠ e_1, e'_2 ≠ e_2\}$

end if

end while

The runtime is $O(|E_1| × |E_2| × t_{sourceTarget})$ with $|E_1|$ and $|E_2|$ being the number of compared edges and $t_{sourceTarget}$ the time to compute their similarity $sourceTargetSim(e_1, e_2)$. $sourceTargetSim(e_1, e_2)$ can be computed in constant time, so the overall runtime is quadratic in the number of edges and sufficiently low with respect to requirement HR 4.
4.3.10 On the Optimality of the Proposed Algorithm

In this section it is discussed whether the proposed algorithm computes optimal results. Our algorithm consists of several sub-algorithms that are executed consecutively. These sub-algorithms have very similar structures and usually find a solution by using a greedy strategy. Applying a greedy strategy has a shorter runtime compared to more sophisticated strategies but might also lead to sub-optimal results. Consider the example in Figure 4.10. We assume that the similarity values reflect the confidence for the respective element correspondences. Because the greedy strategy adds the next best solution it would first add the correspondence with the similarity value 0.9 to the matching. This correspondence would prevent the addition of the two correspondences with the similarity 0.7 each, because activities $A$ and $D$ would be matched already. Even if the threshold value allowed to create the correspondence with the similarity value 0.1, this would still be less optimal than adding correspondences $(A, C)$ and $(B, D)$ to the matching, as adding the two latter correspondences would lead to a more optimal result. This counter example shows that a greedy strategy might not lead to optimal matchings.

![Figure 4.10: Two workflow graph snippets with similarity values and correspondences established by a greedy strategy.](image)

Another source for sub-optimal results is that most of the sub-algorithms work on the correspondences created by their predecessors. If these correspondences are sub-optimal then the results of the depending sub-algorithm will not be optimal either. Consider leaf node correspondences with low confidence values that lead to a matching of fragments with low confidence values.

The problems can be avoided by using more complex algorithms, for example, instead of the greedy algorithm. Nonetheless, finding the optimal solution for a matching between two graphs is an NP-hard problem [25] and has an exponential runtime. Because of runtime Requirement HR 4 we cannot use such an algorithm. Like in related work (compare [15]) our algorithm is based on IDs and heuristics. We show in the evaluation that our approach, even if not optimal, creates sufficient results when applied in practice.
4.3.11 Comparison to Presented Model Matching Algorithms

In this chapter we first presented two algorithms from related work and then explained the matching algorithm we use to match the workflow graph elements and fragments. This section describes the differences and commonalities of our algorithm and the two algorithms presented in Section 4.1.

Chawathe-Algorithm

The sub-algorithm that matches workflow graph fragments is based on the Chawathe-algorithm presented in Section 4.1.1. Instead of the original compare-function for inner nodes a slightly different metric is used (see Definition 16): for the computation of the similarity it is divided only by the number of leaf nodes that have a correspondence. The original function divides by the number of all leaf nodes. Thus, unmatched leaf nodes are ignored for the similarity metric used by us. An algorithm that uses the original metric runs into problems when used on a partial leaf node matching, because although not all possible correspondences are created, the number of all leaf nodes is used to compute the ratio for the similarity of fragments. Thus, the ratio will be lower, and less fragments will be matched compared to the case where only matched leaf nodes are considered.

One other major difference is that the Chawathe-algorithm adds the next possible fragment pair as a correspondence to the matching. The next possible fragment pair is the first pair for which both fragments are unmatched and for which their similarity is above the threshold. In our algorithm, the similarity values for all pairs are computed first, and for correspondence creation the best ones are chosen using a greedy algorithm. This improves the results with respect to requirement HR 1. Still the runtime complexity stays low enough not to compromise requirement HR 4.

Similarity Flooding

The idea of matching model elements based on correspondences in the neighborhood is related to similarity flooding that was presented in Section 4.1.2. Our metric however is focused more on the direct neighborhood of nodes, while with the similarity flooding algorithm similarities are propagated throughout the whole model. In addition, the metric used by us is computed differently and allows it to influence the matching more directly compared to the fixpoint computation approach of similarity flooding.

4.4 Summary

In this chapter we presented two model matching algorithms and discussed their strengths and weaknesses. Afterwards, we defined different metrics to determine the similarity between model elements and model fragments and explained how they can be combined to a similarity-based matching algorithm. In the following chapter an implementation of this matching algorithm and its integration into an existing component for comparing and merging process models is described.
Chapter 5

Implementation and Integration

As a proof of concept we implemented our process model matching algorithm as an extension to an existing component [29, 16] that provides a compare and merge mechanism for the WebSphere Business Modeler [20].

In the following we first present the existing functionality and environment of the compare and merge mechanism. Then we describe requirements for the implementation of our matching algorithm. Thereafter, we present the design and architecture of our component and explain how it was integrated into the existing compare and merge component.

5.1 WebSphere Business Modeler

The WebSphere Business Modeler [20] is part of IBM’s WebSphere software platform for SOA environments, which provides comprehensive support for the development, optimization, and execution of business processes. WebSphere Business Modeler allows to create, simulate and deploy business process models for the use in an execution runtime. It is based on the Eclipse platform [13] and can be extended easily with new functionality.

5.2 Process Compare and Merge

The existing component for the compare and merge mechanism for the WebSphere Business Modeler allows the user to examine the changes between different versions of a model. As known from version control systems for source code (e.g. [18, 46]) it also allows to resolve differences between the local version of a model and the version checked-in into the repository. A possible front-end is depicted by Figure 5.1. The user can select the changes of the checked-in version that are to be carried over to his local copy. The local model is then automatically transformed by the underlying change operations of the selected changes.

As described in Section 2.5.3 the compare and merge mechanism does not rely on a change log, but derives the change operations from a matching which has to be com-
computed beforehand. The matching is based on a static identity-based approach (see Section 2.5.2). This prevents the use of this approach to compare arbitrary process models and incorporates the drawbacks described in Section 3.1.

5.3 Requirements for the Process Matching Component

In this section we explain those matching-related requirements that influence the design of the implemented component. Other requirements that play a role for good software quality [21], but do not directly relate to the matching (e.g. maintainability), are not described here.

On the one hand, the requirements are based on external factors for software quality [21] and, on the other hand, on the work by Selonen et al. [51] and Foertsch et al. [15]. They have examined existing differencing tools and model comparison approaches and derived general requirements that such systems should fulfill. We chose the requirements that are important for model matching and adapted them to reflect our supported use cases. Table 5.1 provides an overview of these requirements. In the following, every time we refer to the user, we are not talking about the end user, but the component or programmer that makes use of our matching component.
Table 5.1: Requirements for the matching component.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR 1</td>
<td>Functionality [21]/Capability</td>
</tr>
<tr>
<td>IR 2</td>
<td>Efficiency [21]</td>
</tr>
<tr>
<td>IR 3</td>
<td>Customizability</td>
</tr>
<tr>
<td>IR 4</td>
<td>Extensibility</td>
</tr>
<tr>
<td>IR 5</td>
<td>Composability</td>
</tr>
<tr>
<td>IR 6</td>
<td>Non-Intrusiveness</td>
</tr>
<tr>
<td>IR 7</td>
<td>Tool-Independence</td>
</tr>
</tbody>
</table>

**IR1** The matching component should correctly implement the matching algorithm that was described in Section 4.3 and support the different use cases presented in Section 3.1.

**IR2** Creating a matching with the matching component should have a reasonable runtime such that the component is usable in practice and can be integrated into existing tools (like the compare and merge component). The matching should run in a few seconds on an average PC for average-sized process models. The runtime requirement was already addressed by HR 4. We restate it here to make sure that it is still fulfilled with the additional computational effort that might arise from the way the matching algorithm (as described in Section 4.3) is implemented. With additional effort we mean computation that is not necessary due to the algorithm itself.

**IR3** The matching component should allow the user to change the threshold values and weights used during the matching. It should also allow for changing the invocation order of the sub-algorithms, as this might be required when the matching algorithm is extended (see also Requirement IR 4).

**IR4** The matching component should be easily extensible for the use with other process modeling languages or for the support of a larger subset of model elements. The component should also allow to easily adapt single sub-algorithms, e.g., to use a different sub-algorithm to match activities based on a more sophisticated matching metric (see discussion in Section 4.2.2).

**IR5** The matching component should be usable as part of a larger application especially the existing compare and merge component.

**IR6** The matching component should not change the original process models until requested by the user. With respect to IR 7 this requirement also minimizes interdependencies with other components that would be difficult to resolve.

**IR7** The matching component should be independent such that it does not rely directly on a certain tool or workflow language. This allows for an easy integration of the component into other systems and applications.
5.4 Component Design and Architecture

In this section we present the design of the matching component that was derived from the requirements stated in the previous section. We will describe how the component is used and present its architecture. Afterwards, we explain how it fulfills the requirements.

5.4.1 Matching Framework

We developed an Eclipse plug-in that is a general similarity-based matching framework for graphs in their PST representation. It is model-independent and does not rely directly on specific matching algorithms. In the following, we will give a brief overview on how the different components of this matching framework interact. In Section 5.4.3 we will describe how we used this matching framework to implement the matching algorithm described in this thesis.

The process to match two models using the framework consists of six steps. These are described in the following:

1. First, the two PSTs of the models are imported into a TreePair, which is their representation in the matching framework. This is done by a class that implements the ITreeImporter interface. Using a generic representation for the models allows all other classes of the framework to be independent of the modeling language and PST implementation used.

2. After the import of the PSTs, the model elements can be matched in the matching environment. The second step is to configure the concrete matching strategy. This is done by the use of the appropriate MicroMatcher classes. Every MicroMatcher extends an abstract class of the framework and represents an atomic matching strategy, e.g. for a single element type. More specific, every MicroMatcher can increase or decrease the similarity score for a pair of model elements of the two models. During this step the MicroMatchers that are used are registered with the TreeMatcher class, which is the main component.

3. During the third step an existing (partial) matching can be imported into the framework. This eases the integration with other tools, because existing matching components do not have to be redeveloped as MicroMatchers. As with the model import, the import of the matching is executed by a class that implements an specific interface, in this case the IMatchingsImporter. The existing correspondences are converted into similarity scores for the affected elements. As MicroMatchers can increase and decrease similarity scores, the imported correspondences (similarity scores) are not necessarily part of the final matching result. They are merely interpreted as starting points for the remaining matching process. The class that maintains the similarity values and offers methods to change them is called SimilaritiesKeeper.

4. In this step the MicroMatchers that were registered with the main component are executed. During their execution they loop over elements pairs and increase
5. After the calculation of the similarity values via the MicroMatchers the final set of correspondences (the matching) is computed by the SimilaritiesKeeper. The matching is derived using a greedy algorithm (current implementation).

6. As a last step the matching is exported back into the representation of the original models. Again, a class that implements an interface is used, in this case the IMatchingsExporter interface.

5.4.2 Components

In the last section we described how a matching is computed using the matching framework. Here we present the main components and their responsibilities. Figure 5.2 gives an overview of the components and their most important interfaces.

**TreeMatcher** The TreeMatcher component is the main component of the matching framework and the interface between the user (or using component) and the framework. It provides methods to register MicroMatcher and to invoke the similarity computation. It also provides methods to generate a matching and to return it in a language-specific format.

**SimilaritiesKeeper** The SimilaritiesKeeper maintains the similarity values of model elements pairs and provides an interface to change the values to the MicroMatchers. In addition it offers the TreeMatcher a method to create a matching based on the similarity values.
MicroMatcher A MicroMatcher computes similarity values for element pairs and implements a partial matching strategy, e.g. one of the sub-algorithms presented in this thesis. These values are added to the overall similarity of the element pairs using an interface of the SimilaritiesKeeper.

TreePair The TreePair is the internal data model of the two process model PSTs. Its elements can be dynamically enriched with properties that are exploited by the MicroMatchers during similarity calculation.

TreeImporter The TreeImporter converts the two models under comparison into a TreePair. It enriches the internal representation of the tree elements with properties that are exploited during the similarity computation.

MatchingsExporter The MatchingsExporter converts the matching into a representation that is supported by the component that uses the framework, e.g. the compare and merge component.

MatchingsImporter The MatchingsImporter converts an existing matching computed by another tool or component into initial similarity values of the SimilaritiesKeeper such that they can be considered for matching.

5.4.3 Implementation of the Matching Algorithm

In this section we describe how we implemented our matching algorithm that was developed in this thesis using the described framework. We created a class called MatchingsProvider that acts as the common interface for the matching component. The MatchingsProvider offers a method that first initializes the framework with the passed process models, then computes the matching, and finally returns it. The import and export capabilities are modeling language specific (see e.g. Section 5.5.2) and passed to the constructor of the MatchingsProvider as objects that implement the according interface.

Our matching algorithm presented in Section 4.3 mainly consists of different sub-algorithms that are executed in a specific order. We implemented the algorithm by first developing a MicroMatcher for each sub-algorithm. Then we set up their execution in the matching method of the MatchingsProvider. We did this by registering these MicroMatchers with the TreeMatcher in the specific order we need them to be executed.

5.4.4 Advantages of the Matching Framework

In Section 5.3 we described the requirements for the matching component. In the following, we will explain how our matching framework fulfills these requirements and what its strengths are.

To IR 1 The matching framework allows it to implement the matching algorithm presented in this thesis. Each sub-algorithms can be implemented as a MicroMatcher. The MatchingsKeeper establish the matching using a greedy algorithm that takes
into account the depthsum of nodes (see Section 4.2.4) – using the algorithm described in this thesis. The different scenarios are supported by the matching framework, because the import and export functionalities make the framework compatible with various modeling languages. An existing partial matchings can also be imported.

To IR 2 Additional computational overhead to the matching algorithm itself is introduced by the import of the models, the import of an existing matching, and the export of the final matching. The complexity of the import and export of the matching depends partly on the external representation that is used and for which we cannot make any assumptions here. In general it should be linear in the size of the models, assuming it is a list of correspondences. The import of the trees is also linear in the size of the models. Like the in- and export of matchings its complexity depends on the representation of the models. If all element properties that are necessary for the matching can be derived directly, we have a linear runtime. If additional computation is necessary to compute them, then the runtime could be worse.

Our matching algorithm is mainly based on basic type information of model elements. In the general case it should be possible to derive this information in constant time. Whether a fragment is well-structured can be determined in linear time. The computation of a BFS-traversal can also be computed in linear time (see Section 4.3.6) and so this requirement is fulfilled.

To IR 3 The matching framework is customizable, because the kind and number of MicroMatchers, their execution order, and the weights of the similarity values that they compute are parameterized and can be easily adapted to different use cases. Additionally, it is possible to define parameterized MicroMatchers and to change the matching threshold used for matching (see Section 2.5.2) at any time.

To IR 4 The framework can support new modeling languages and model element types by adjusting the import and export classes. To compute similarity values for new element types it might also be necessary to develop new MicroMatchers. The MicroMatchers work only on the TreePair and have no direct dependency to the original models.

Changes in sub-algorithms of the matching algorithm can be implemented by using derived or substituted MicroMatchers. Replacing or extending the similarity computation is simple, because the MicroMatchers are only loosely coupled with the TreeMatcher.

To IR 5 Due to its flexible in- and export the framework can be easily used by other components and larger applications. The tree matcher main component provides the central interface to other systems.

To IR 6 The original models are not affected by the matching process, because the MicroMatcher work only on the imported representation as a TreePair. In addi-
The matching is exported into a new representation of own choice, independent of the original models.

To IR 7 The matching framework is independent of concrete process modeling languages. The only language-dependent parts are the import and export classes. The MicroMatchers might be adapted to a concrete language, but as they work only on the TreePair, they do not have direct dependencies to this language. The invocation of MicroMatchers and their order can be easily changed. A new MicroMatcher can be specified easily, since only the method \textit{match()} of an abstract MicroMatcher base class has to be implemented. This method is called by the TreeMatcher and should compute the similarity value for the respective element pairs.

5.5 Integration with Process Compare and Merge

The process compare and merge component is on a static identity-based matching. This approach suffers from the problems described in Section 2.5.2. To overcome these problems the ID-based matching shall be extended with the matching created by our algorithm (where the first sub-algorithm, which matches elements based on their IDs, does not have to be invoked). In the following, we first describe the architecture of the existing compare and merge component. Afterwards, we explain how we integrated our matching component.

5.5.1 Compare and Merge Architecture

The existing compare and merge component consists of several sub-components which will be briefly presented in the following.

\textbf{PSTCreator} This component converts the process models into their PST decomposition (see Section 2.3.2).

\textbf{CorrespondencesCalculator} This component computes the static identity-based matching of the two PSTs by creating correspondences for model elements with the same ID.

\textbf{Process Merging Graph} The \textit{Process Merging Graph} (PMG) is the main component. It invokes the \textit{PSTCreator} to convert the process models into their PST decomposition and runs the \textit{CorrespondencesCalculator} to compute the matching. Afterwards, it calls the \textit{DifferencesCalculator} to compute the change operations.

\textbf{DifferencesCalculator} This component computes the change operations based on a matching of two PSTs (see Section 2.5.3).

\textbf{Difference View} The \textit{Difference View} invokes the \textit{Process Merging Graph} and shows the differences between two process models in form of change operations. The user can apply the changes by clicking on the appropriate entry.
The interaction between the different components takes place as follows: when the user wants to compare two process models, a Difference View is instantiated and invokes the Process Merging Graph with the two models. The PMG uses the PSTCreator to decompose the models into their PSTs and runs the CorrespondencesCalculator to compute the matching between them. Then the DifferencesCalculator is used to compute the change operations. These are then presented to the user by the Difference View.

5.5.2 Integration of Matching Component

For the integration of our matching component we had to extend the existing compare and merge component. First, we extended the CorrespondencesCalculator to call our matching component after the initial matching. Second, we had to extend the PMG with import and export functions for the matching. We developed a new lightweight data structure called BasePG to represent the matching. For our matching component we had to develop classes that implement the interfaces IMatchingsImporter and IMatchingsExporter of the matching framework and an appropriate importer class for the PSTs that are used.

We changed the existing compare and merge component as follows: the Difference View still instantiates the Process Merging Graph, which runs the CorrespondencesCalculator with the two PSTs. The CorrespondencesCalculator now invokes our matching component. It uses the export function of the PMG to pass the preliminary matching based on IDs. Our matching component imports the PSTs and the matching and computes the final matching that is exported to a BasePG. The CorrespondencesCalculator uses the import function of the PMG to read in the exported BasePG with the final matching. The next steps are unchanged: The DifferencesCalculator computes the operations based on the (updated) matching and these are presented by the Difference View.

5.6 Summary

In this chapter we described the process matching component that implements the developed matching algorithm. We first introduced the IBM WebSphere Business Modeler and an existing component for comparing and merging process models. We then described the requirements for our component, its architecture, and its integration into the existing compare and merge component. In the following chapter we summarize our work and give an outlook on future work.
Chapter 5. Implementation and Integration
Chapter 6

Evaluation

In this chapter we evaluate different aspects around process model matching with our algorithm. Our evaluation can be divided into two parts. First, we prove assumptions about properties of real world processes that were used in this thesis. We evaluate the amount of well-formed fragments of a process and the size of process models in terms of nodes and edges as well as the degree of nodes in an average model.

The second part of the evaluation deals with the matching created by our algorithm. We first describe how model matching algorithms are evaluated in related literature. Afterwards, we describe how we evaluate the correctness of our matching algorithm.

6.1 Process Model Properties

In this section we present the evaluation of different properties of real-life process models. We assumed certain characteristics when we designed the matching algorithm. Here we want to prove that our assumptions were right.

For the evaluation we investigated the processes of reference process model libraries (RML) from different industries. We translated the models into their workflow graph representation and collected different statistics. Table 6.1 provides an overview. In the following we want to briefly describe how the figures support our assumptions.

Size of Process Models In Requirement HR 4 we require the algorithm to run in seconds on an average PC for models of average size. The evaluation showed that the average size of models in terms of nodes and edges is about the same for all three reference libraries: the average model has 40 nodes and not more than 60 edges. The size is in the same order of magnitude that we assumed.

Well-Structured Fragments In Section 4.3.4 we described a sub-algorithm for the matching of gateways, which are the entry- or exit-node of well-structured fragments. We assumed that an average process model contains many well-structured fragments and that it is valuable to define a control node matching specific to gateways that are the entry- and exit-nodes of well-formed fragments. The figures
Chapter 6. Evaluation

<table>
<thead>
<tr>
<th>Library</th>
<th>RML1</th>
<th>RML2</th>
<th>RML3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes</td>
<td>284</td>
<td>421</td>
<td>318</td>
</tr>
<tr>
<td>Nodes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>avg. nodes/process</td>
<td>40.9</td>
<td>42.6</td>
<td>41</td>
</tr>
<tr>
<td>avg. activities/process</td>
<td>7.9</td>
<td>9.1</td>
<td>13.4</td>
</tr>
<tr>
<td>avg. gateways/process</td>
<td>12.6</td>
<td>11.8</td>
<td>3</td>
</tr>
<tr>
<td>Fragments in General</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>avg. fragments/process</td>
<td>51.6</td>
<td>47.8</td>
<td>14.2</td>
</tr>
<tr>
<td>Well-structured Fragments</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>avg. fragments/process</td>
<td>51.4</td>
<td>47.6</td>
<td>14.2</td>
</tr>
<tr>
<td>Edges</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>avg. edges/process</td>
<td>61.2</td>
<td>61.1</td>
<td>45.69</td>
</tr>
<tr>
<td>Degree of Nodes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>avg. edges/node</td>
<td>2</td>
<td>1.8</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Table 6.1: Characteristics of industrial process libraries.

show that nearly all fragments in process libraries are well-structured. In all three libraries there is only a very small fraction of unstructured fragments.

**Degree of Nodes** In Section 4.3.5 we estimated the runtime of the algorithm for the control node matching based on neighboring nodes. Our estimation was based on the assumption that the degree of nodes in process models is usually rather low. Our observations support this assumption: the average degree of nodes is between 1.1 and 2 for all three libraries.

### 6.2 Quality of Matching Algorithm

In this section we first briefly describe how matching algorithms are evaluated in related literature. Afterwards, we present our evaluation approach and discuss the results.

#### 6.2.1 Evaluation in Related Work

In related literature there are mainly two approaches to measure the quality of model change detection algorithms. One common approach is to use the number of detected operations (e.g. [8]): Let $S$ be the change log which is the set of operations necessary to transform one model into the other. Then two results $S$ and $S'$ are defined to be equally good when $|S| = |S'|$. Otherwise, the smaller of both sets is the better one. The optimal result can be defined as the set $S$ with $|S| \leq |S_i| : \forall S_i$, regarding the comparison of the same two models and operation types. This approach has the advantage that it captures the quality of change detection mathematically and by this in an objective way. This is the prerequisite for an unbiased and automatic evaluation. The drawback is that this approach only works for change detection and only in scenarios where the basic
assumption holds that fewer operations are equivalent to a qualitative better result. Furthermore, one has to know all possible change logs for two models in order to find the best one.

Another approach to the evaluation of change detection is to assess the results of the change detection manually (e.g. [59, 24]). In contrast to the other approach, here the quality of the change detection is measured in a subjective way [15]. Although this is less desirable in general, it allows to “go beyond mechanical comparison of diagrams/models and address the needs of actual software developers” [51]. For example, Melnik et al. [38] base the evaluation of their schema matching on the expectations of several persons. These were required to create manual matchings, which then were compared to the results of the matching algorithm. The algorithm was then evaluated with respect to the matching preferences of each user.

## 6.2.2 Evaluation of the Matching Algorithm

In the previous section we described the two approaches to evaluate model matchings used in related literature. The first approach assesses the matching based on the number of computed change operations. As described in Section 2.5.3 the change operations can be directly derived from a matching. Unfortunately, we cannot use this approach because the underlying assumption is not true in our case: fewer operations are not necessarily equivalent to a better result. Consider the case where instead of a delete and an insert operation a move operation is detected for a decision node. In this case the decision node of the one model has a correspondence to a decision node in the other model. However, this additional correspondence might violate Requirements LR 5 and LR 6, for example if the activities in the neighborhood are completely different. The matching is not correct although fewer operations are computed. Hence, the size of the change log cannot be used as a measure for the matching.

The second approach is completely subjective, since the results are evaluated manually. We want to avoid a subjective evaluation and choose a third evaluation approach based on our definition of an optimal matching.

In Section 2.4.3 we gave a definition of the optimal matching of two workflow graphs with respect to a confidence function. As it is impossible to describe this confidence function for the general case of model matching, we developed a confidence function for a specific scenario in Section 3.2.2. The presented confidence function allows it to asses a matching in the case of two models that are derived from each other and where the changes are known.

We first evaluate our matching algorithm for the special case of self-matching, which is the comparison of two identical copies of a process model. The second evaluation is based on the case of models that were derived from each other by non-overlapping changes. In both cases we can use the concrete confidence function that we defined earlier to asses the matching of model pairs. Thus, we can evaluate our matching in an objective way. As an additional test we compared the matchings by our algorithm to the matchings by an ID-based approach and collected first results. In the following we will explain the details and the results of our evaluation.
For the evaluation we use the implementation of the matching algorithm that was described in Chapter 5. The algorithm introduces different threshold values and weights. The values that we use were determined by empirical tests using reasonable starting points (e.g. \( \text{threshold}_{frg} = 0.5 \), see Section 4.3.6). Thus, there might be potential for improvement, however, how to systematically find the optimal threshold settings remains for future work.

Self-Matching

In our self-matching evaluation scenario we use our implementation to match a workflow graph of a process model to an identical copy of itself. We ignore the IDs contained in the model to mimic the third use case (see Section 3.1.3), which is the most demanding one in terms of matching. We do this by simply skipping the run of the sub-algorithm (MicroMatcher, see Section 5.4.3) that matches elements based on their IDs (see Section 4.3.1). This way the matching treats the models as if they would not contain IDs.

We computed the matching for the pair of identical workflow graphs for every process model in the presented libraries and checked the matching quality. Matching a workflow graph to an identical copy is a special case of matching two graphs that are derived from each other. It is simply the case where the second derived graph does not contain any changes.

To find out if a matching is correct, we do the following: consistent with the definition of the confidence function in Definition 11 we have to check whether the two elements of a correspondence are counterparts of each other. If so, the quality of the matching improves (+1), otherwise it decreases (-1). Hence, we can measure whether the matching is optimal by simply counting the wrong correspondences. We can easily determine whether the counterparts in two models are matched by using the model element IDs. For nodes the elements in a correspondence should have the same ID. For fragments, the IDs of the elements contained by the one fragment should be identical to the IDs of the elements contained by the other fragment.

We performed two self-matching tests for all models in our three reference model libraries. First, we evaluated the preliminary fragment matching of our matching algorithm and afterwards the complete matching.

In our matching algorithm, we compute a partial fragment matching that is based only on activity correspondences (in the case where no elements are matched based on IDs). We use it to match gateways based on their parent fragments (see Section 4.3.4). In the evaluation we investigate whether using only activities for the preliminary fragment matching causes wrong fragment correspondences. Table 6.2 shows the results of the evaluation. It shows that the matching of activities based on the name similarity was correct. It also shows that activity-containing fragments (other fragments are still unmatched) were matched correctly. Thus, the preliminary fragment matching was correct for all models of the checked libraries.

Besides the preliminary fragment matching we evaluated the complete matching by our whole algorithm (excluding the ID-based sub-algorithm). The results are shown in
Chapter 6. Evaluation

<table>
<thead>
<tr>
<th></th>
<th>RML1</th>
<th>RML2</th>
<th>RML3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cases of Activity Ambiguity</td>
<td>0</td>
<td>0</td>
<td>238</td>
</tr>
<tr>
<td>Preliminary Fragment Matching</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Corr. Activity Corresp.</td>
<td>2234</td>
<td>3850</td>
<td>4259</td>
</tr>
<tr>
<td>Wrong Activity Corresp.</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Wrong Act. contain. Frag. Corresp.</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Complete Matching</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Corr. Node Corresp.</td>
<td>7324</td>
<td>11694</td>
<td>13039</td>
</tr>
<tr>
<td>Wrong Node Corresp.</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Corr. Frag. Corresp.</td>
<td>13894</td>
<td>18910</td>
<td>4508</td>
</tr>
<tr>
<td>Wrong Frag. Corresp.</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.2: Results for matching workflow graphs of processes from reference libraries with themselves.

The lower part of Table 6.2. Our algorithm computed the correct correspondences for all processes in the library for the self-matching case. Since all the correspondences were correct, the derived matchings were optimal with respect to the Definitions 10 and 11.

Despite the strong results the table reveals a potential problem of the matching algorithm. For the third library the table shows that there are process models that contain activity names more than once. This did not affect the matching of activities in our evaluation. However, it indicates that in order to match the corresponding activities to each other, the matching of activities might need to be extended with techniques that go beyond the comparison of names.

Change Detection

In this evaluation scenario we manually derived models by consecutive changes to existing models. The changes were non-overlapping such that they did not cancel out (e.g. insert and delete of same element) or influence (e.g. insert a move of same element) each other. The models were not manipulated by ourselves, but by people who are not directly involved with our work. Because we kept track of the changes (and hence the change operations), we can again use the confidence function defined in Definition 11 to evaluate the matchings and the quality of our matching algorithm with respect to an optimal matching.

We let three people apply a series of non-overlapping changes to three different models from the first library. We then manually determined the corresponding compound operations (see Section 2.5.3). Tables 6.3 and 6.4 give an overview of the applied change operations and their effect on the fragment structure as well as the expected unmatched nodes and fragments. The first table focuses on fragments that contain activities. The second table captures all nodes and fragments. $V'_i$-$V'''_i$ denote intermediate versions.
Chapter 6. Evaluation

Table 6.3: Overview of derived workflow graphs regarding activity containing fragments.

<table>
<thead>
<tr>
<th>Model</th>
<th>Additional Node Operations</th>
<th>Implied activ. contain. Fragment Changes</th>
<th>Exp. unmatched activ. contain. Frags</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_i'$</td>
<td>Insert Activity</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>$V_i''$</td>
<td>Insert Decision</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>$V_i'''$</td>
<td>Insert Decision</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>$V_i''''$</td>
<td>Insert Activ., Delete Activ., Move Activ.</td>
<td>Insert Fragment</td>
<td>1</td>
</tr>
<tr>
<td>$V_2'$</td>
<td>Move Activity</td>
<td>2x Insert Fragment</td>
<td>2</td>
</tr>
<tr>
<td>$V_2''$</td>
<td>Insert Loop Nodes, Move Activ., Insert Start, Insert Termination</td>
<td>Insert Fragment</td>
<td>3</td>
</tr>
<tr>
<td>$V_2'''$</td>
<td>Delete Start, 2x Delete Termination</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>$V_3'$</td>
<td>2x Delete Activity</td>
<td>2x Delete Fragment</td>
<td>2</td>
</tr>
<tr>
<td>$V_3''$</td>
<td>Insert Fork, Insert Join, 3x Move Activity</td>
<td>2x Insert Fragment</td>
<td>4</td>
</tr>
<tr>
<td>$V_3'''$</td>
<td>2x Insert Activity</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>$V_3''''$</td>
<td>Insert Subprocess Nodes, Insert Start, Insert Termination, Insert Activ.</td>
<td>Insert Fragment</td>
<td>5</td>
</tr>
</tbody>
</table>

Using the confidence function we rated the created matching by our algorithm. Looking at the elements of a workflow graph there are two cases in general: the element is matched correctly (1) or it is not (-1).

As for the self-matching we performed two tests. First, we tested the preliminary fragment matching. Here we evaluated the correspondences between activities and activity-containing fragments. Second, we evaluated the complete matching computed by our whole algorithm. Table 6.5 shows the results of the comparison between the optimal matching and the preliminary fragment matching by our implementation. The table shows that the results were good in general. However, we have a fragment matching error for the first set of derived models and two fragment matching errors for the third set of derived models. The reason for the errors are corresponding fragments where only one of the two contains activities (e.g. because an activity was moved into it). Clearly, a correspondence between an activity-containing and a non-activity-containing fragment cannot be created with an activity-based fragment matching. It could be ar-
<table>
<thead>
<tr>
<th>Model</th>
<th>Additional Node Operations</th>
<th>Implied Fragment Changes</th>
<th>Exp. unmatch. Nodes</th>
<th>Exp. unmatch. Fragments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V'_1$</td>
<td>Insert Activity</td>
<td>—</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$V''_1$</td>
<td>Insert Decision</td>
<td>—</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$V'''_1$</td>
<td>Insert Decision</td>
<td>—</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>$V^{'''}_1$</td>
<td>Insert Activ., Delete Activ., Move Activ.</td>
<td>Insert Fragment</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>$V'_2$</td>
<td>Move Activity</td>
<td>3x Insert Fragment</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>$V''_2$</td>
<td>Insert Loop Nodes, Move Activ., Insert Start, Insert Termination</td>
<td>Insert Fragment</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$V'''_2$</td>
<td>Delete Start, 2x Delete Termination</td>
<td>3x Delete Fragment</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>$V'_3$</td>
<td>2x Delete Activity</td>
<td>2x Delete Fragment</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$V''_3$</td>
<td>Insert Fork, Insert Join, 3x Move Activity</td>
<td>2x Insert Fragment</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$V'''_3$</td>
<td>2x Insert Activity</td>
<td>—</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$V^{'''}_3$</td>
<td>Insert Subprocess Nodes, Insert Start, Insert Termination, Insert Activ.</td>
<td>Insert Fragment</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 6.4: Overview of derived workflow graphs regarding all nodes and fragments.
Chapter 6. Evaluation

Table 6.5: Matching results for preliminary fragment matching of derived models.

<table>
<thead>
<tr>
<th>Model</th>
<th>Unmatched Activ. Fragments</th>
<th>Add. (activ. cont.) Fragments Matching Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V'_1$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$V''_1$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$V'''_1$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$V'_4$</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$V''_2$</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$V'''_2$</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>$V'_3$</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$V''_3$</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>$V'''_3$</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>$V''''_3$</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.5: Matching results for preliminary fragment matching of derived models.

argued that this kind of correspondences should be excluded from the inspection (like fragments and fragment correspondences that do not contain activities at all). The results of the complete matching indicate that the reported missing fragment correspondences are not really a problem as they are created later during the matching process (compare results of complete matching).

The results for the complete matching are shown in Table 6.6. The results are in general very positive; however, the table shows an error in the node correspondences. For the matching of the last of the models derived from model $V_2$ there is an error in the matching for a start node. The reason is that the matching algorithm creates a correspondence for a newly inserted start node $s_2$ to a start node $s_1$, whose counterpart $s'_1$ was deleted earlier. As the newly created start node $s_2$ is located close to the position of $s_1$, it could be argued that with Requirement LR 7 this correspondence is not really indicating an error, because we want to match start nodes that have a common neighborhood.

In this section we have described two matching approaches from related literature and presented our own third evaluation approach. The results of our evaluation are promising as a proof of concept. More (real world) test models would be required to show the capabilities of the algorithm and the matching framework for the daily practice.

Test with Test Model Collection

As an additional test and to compare the results by our algorithm to an ID-based approach we tested our algorithm on a collection of about 300 test model pairs. We compared the detected operations based on the matching by our algorithm (without the ID-based sub-algorithm) with change logs that were derived by detecting operations with an ID-based matching approach. Such a comparison bears problems for several reasons. The change logs might not necessarily denote the optimal matchings (see Section 6.2.1) and if they do, they might not show the only possible optimal matchings. In addition many of the models do not adhere to the restrictions we imposed on the models we want
Chapter 6. Evaluation

<table>
<thead>
<tr>
<th>Model</th>
<th>Unmatched Nodes</th>
<th>Unmatched Fragments</th>
<th>Add. Node Matching Errors</th>
<th>Add. Fragment Matching Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V'_1$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V''_1$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V'''_1$</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V''''_1$</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V'_2$</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V''_2$</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V'''_2$</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$V'_3$</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V''_3$</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V'''_3$</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V''''_3$</td>
<td>8</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.6: Matching results for complete matching of derived models.

to consider for our matching algorithm (e.g. that every node is on a path from a start node to an end node and that every activity has one predecessor and one successor, see Section 2.2.2). Nevertheless, this evaluation gives an insight on how close a matching using the heuristics gets to an ID-based matching.

The manual comparison of several samples showed that the operations detected using our matching were often very close or identical to the results using the ID-based approach. A detailed comparison of all results requires more time and remains for future work.

6.3 Summary

In this chapter we discussed the evaluation using industrial process models from reference libraries. We first showed different process model properties that we assumed for our matching algorithm. Afterwards, we discussed evaluation approaches to model matching in related literature and developed our own evaluation method. The evaluation results are positive and support our matching approach. The first results from the comparison with an ID-based approach support the assumption that matching heuristics can achieve comparable results without using ID information.
Chapter 6. Evaluation
Chapter 7

Summary and Future Work

7.1 Summary and Contributions

In this thesis we presented an approach for matching business process models in order to detect differences between them. Metrics to measure the similarity between different types of model elements were introduced and an algorithm that establishes a matching between process models were developed. We now summarize the results and draw conclusions.

In Chapter 1 Business-driven Development was introduced, a model-centric approach for developing business-related IT solutions. As process models are continuously refined by different people and across different levels of abstraction, there is a need for quickly detecting (and resolving) changes between different models. Changes can be computed based on a matching of two process models. Creating such a matching is difficult if it cannot be based solely on identical identifiers (IDs) for corresponding elements. Furthermore, many known approaches do not consider a matching of coherent model element groups, i.e. fragments. As a consequence, only low-level differences among elements can be computed, but not among fragments.

In Chapter 2 the main concepts of BPMN, a language that allows us to model process models, were discussed. We have also defined the workflow graph, an abstract representation of such languages. The terminology around matching was introduced and we explained that a matching can be established by measuring the similarity between model elements from the two models under comparison. We also presented the process structure tree, a decomposition technique that is known from compiler theory, and which we use to identify process Single-Entry-Single-Exit (SESE) fragments.

In Chapter 3 three different use cases were described. First, the Version Control scenario was introduced, for which all corresponding elements share the same ID except for recreated elements. Second, the BPEL-2-BPMN Feedback use case was presented, in which a partial ID mapping between model elements is preserved. Finally, we explained that in the Reference Model Customization scenario, models do not share a common modeling history and consequently also no IDs. We defined the requirements for the matching based on these use cases and showed that related work does not fulfill
Chapter 7. Summary and Future Work

all requirements. Similarity metrics, including a metric for process fragments (Chapter 4), were presented and a matching algorithm that incorporates the previously defined metrics and fulfills the defined requirements was developed. It is capable of matching process models in all three use cases, thus also in the absence of IDs.

In Chapter 5 the flexible matching framework that we designed to implement the matching algorithm was described. It is an Eclipse plug-in that is modeling language independent and configurable in terms of used similarity metrics and their parameterization. This plug-in was integrated into an existing compare and merge mechanism for the IBM WebSphere Business Modeler that was restricted to ID-based matching before.

The implementation was used to evaluate the matching in two scenarios in which an optimal matching can be defined. First, a matching between identical models was established and we showed for three reference model libraries that the matching algorithm works correctly in this scenario (Chapter 6). Afterwards, it was shown that the algorithm can also be applied for differing models. This was exemplified for the case of models that were derived from each other via non-overlapping changes.

Within this thesis we made the following major contributions that support business analysts in the detection of changes between process models:

- The definition of similarity metrics for the matching of business processes, including a metric to measure the similarity between SESE fragments
- A hybrid approach to business process matching that makes use of, but is not restricted to, (partial) ID mappings, that includes the matching of SESE fragments, and that fulfills the requirements developed in this thesis (see Table 3.3)
- A configurable and extensible modeling language independent framework for similarity-based matching and an implementation of the matching algorithm in that framework
- The integration of the matching implementation into the compare and merge mechanism for the IBM WebSphere Business Modeler. The extension allows the matching between arbitrary models created with that tool and enables the compare and merge component to detect and resolve changes between them.

7.2 Future Work

In this thesis our business process matching approach was applied to models in the notation supported by IBM WebSphere Business Modeler. Future work might include the support for other languages, such as Business Process Modeling Notation (BPMN) or Business Process Execution Language (BPEL). Due to the flexibility of our framework this would affect mainly the import of model elements into the framework.

Another possible extension of our work might be the consideration of further properties of business process models during the matching process. Among these properties
are data flow and branching conditions of decision nodes. Using these additional properties could improve the matching of edges and decision nodes as their matching would not be solely based on their neighboring nodes anymore.

A further possibility to extend this work might be to improve the similarity function for activity names by taking into account word lists and ontologies as discussed in Section 4.2.2. This is especially important for the Reference Model Customization use case, for which it is less likely that identical activities have similar names. Furthermore, the name similarity could be extended with a neighborhood-based approach in order to successfully match processes where activities with the same name are used several times.

The proposed matching algorithm uses several weights and threshold values. For the evaluation these values were determined manually. Developing a systematic approach to find the best threshold values and weights for a certain use case could improve the matching quality delivered by the algorithm. Finding the optimal values requires real world data on process models refinement. Unfortunately, it is difficult to obtain such information.

In this thesis we developed an algorithm that matches corresponding model elements based on their similarity. The approach could be extended to compute the similarity of entire processes. This would indicate to which extent the two processes are identical and express the same ideas. The question is whether it is sufficient to add up the similarity scores of corresponding elements or if more advanced techniques have to be used.
Bibliography


## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Two differing processes with matching activities in UML-AD notation.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>A matching of model elements based on unique identifiers.</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>A simple business process for claim management in UML-AD notation.</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>The different node types of BPMN.</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>A collapsed and expanded subprocess in BPMN notation.</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>A business process for claim management in BPMN notation.</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>The different node types of a workflow graph.</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>A subprocess that contains a loop in its workflow graph representation.</td>
<td>12</td>
</tr>
<tr>
<td>2.7</td>
<td>A workflow graph that is not well-formed.</td>
<td>12</td>
</tr>
<tr>
<td>2.8</td>
<td>A workflow graph of a business process for claim management that contains a subprocess.</td>
<td>13</td>
</tr>
<tr>
<td>2.9</td>
<td>A workflow graph decomposed into canonical SESE fragments.</td>
<td>14</td>
</tr>
<tr>
<td>2.10</td>
<td>The process structure tree of the workflow graph in Figure 2.9.</td>
<td>15</td>
</tr>
<tr>
<td>2.11</td>
<td>A workflow graph containing an unstructured SESE fragment.</td>
<td>16</td>
</tr>
<tr>
<td>2.12</td>
<td>Two workflow graphs with correspondences between some of their elements.</td>
<td>17</td>
</tr>
<tr>
<td>2.13</td>
<td>Two activities $n$ and $n'$ and directly connected corresponding activities.</td>
<td>20</td>
</tr>
<tr>
<td>2.14</td>
<td>Two workflow graphs matched using a static identity-based approach.</td>
<td>21</td>
</tr>
<tr>
<td>2.15</td>
<td>Two workflow graphs matched using a signature-based approach.</td>
<td>22</td>
</tr>
<tr>
<td>2.16</td>
<td>Two workflow graphs matched using a similarity-based approach and a threshold of 0.6.</td>
<td>23</td>
</tr>
<tr>
<td>2.17</td>
<td>Two workflow graphs with correspondences between their elements.</td>
<td>24</td>
</tr>
<tr>
<td>2.18</td>
<td>A process structure tree with annotated operations</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>A workflow graph from a repository and the same model with local changes.</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Workflow graphs of a BPMN model that was exported to BPEL, changed and reimported back to BPMN.</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>The workflow graphs of a reference process and a refined process that do not share common IDs.</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Two workflow graphs with corresponding activities.</td>
<td>36</td>
</tr>
<tr>
<td>3.5</td>
<td>Two workflow graphs with corresponding decision&amp;merge pairs that enclose corresponding activities.</td>
<td>37</td>
</tr>
</tbody>
</table>
3.6 Two workflow graphs with corresponding decision nodes that are surrounded by corresponding activities. ........................................... 37
3.7 Two workflow graphs with corresponding opening and closing loop nodes. 38
3.8 Two workflow graphs with corresponding opening and closing subprocess nodes. ................................................................. 38
3.9 Two workflow graphs with corresponding fragments based on contained corresponding activities. ............................................. 39

4.1 Two trees with corresponding leaf and inner nodes. ......................... 49
4.2 Example for the similarity flooding algorithm ................................... 51
4.3 Two fragments of two workflow graphs and their fragment similarity values. 58
4.4 Two workflow graph snippets and their fragment correspondences ........ 59
4.5 Two workflow graph snippets with each a gateway and neighboring activities 60
4.6 Two differing workflow graphs for the handling of claims. .................. 69
4.7 Two differing workflow graphs for the handling of claims with the partial matching by the first three sub-algorithms. ............................. 74
4.8 A workflow graphs with two structured fragments $f_Y, f_X$ and a workflow graph with an unstructured fragment $f_U$. .......................... 76
4.9 Two differing workflow graphs for the handling of claims with the partial matching by the first five sub-algorithms. ....................... 78
4.10 Two workflow graph snippets with similarity values and correspondences established by a greedy strategy. ............................... 81

5.1 Screenshot of the compare and merge component frontend. ............... 84
5.2 The component architecture of the matching framework. .................. 87
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Differences in Figure 1.1 expressed with different types of operations.</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>BPMN model elements and their workflow graph equivalent.</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>High-level requirements for matching workflow graphs.</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Low-level requirements for matching workflow graphs.</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Comparison of related work with respect to the requirements.</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Similarity values for leaf nodes of the trees in Figure 4.1.</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>Matching order of element types and applied matching strategies.</td>
<td>66</td>
</tr>
<tr>
<td>4.3</td>
<td>Name-based similarity values of activities from Figure 4.6 that cannot be matched using IDs.</td>
<td>71</td>
</tr>
<tr>
<td>4.4</td>
<td>Order of created correspondences by Algorithm 3 based on the similarity values shown in Table 4.3.</td>
<td>71</td>
</tr>
<tr>
<td>5.1</td>
<td>Requirements for the matching component.</td>
<td>85</td>
</tr>
<tr>
<td>6.1</td>
<td>Characteristics of industrial process libraries.</td>
<td>94</td>
</tr>
<tr>
<td>6.2</td>
<td>Results for matching workflow graphs of processes from reference libraries with themselves.</td>
<td>97</td>
</tr>
<tr>
<td>6.3</td>
<td>Overview of derived workflow graphs regarding activity containing fragments.</td>
<td>98</td>
</tr>
<tr>
<td>6.4</td>
<td>Overview of derived workflow graphs regarding all nodes and fragments.</td>
<td>99</td>
</tr>
<tr>
<td>6.5</td>
<td>Matching results for preliminary fragment matching of derived models.</td>
<td>100</td>
</tr>
<tr>
<td>6.6</td>
<td>Matching results for complete matching of derived models.</td>
<td>101</td>
</tr>
</tbody>
</table>