



universität  
wien

# MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Versioning and Evolution in Process-aware Information  
Systems“

verfasst von / submitted by

Michal Tkáčik, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of  
Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2018 / Vienna 2018

Studienkennzahl lt. Studienblatt /  
degree programme code as it appears on  
the student record sheet:

A 066 926

Studienrichtung lt. Studienblatt /  
degree programme as it appears on  
the student record sheet:

Masterstudium Wirtschaftsinformatik

Betreut von / Supervisor:

Univ.-Prof. Dipl.-Math. oec. Dr. Stefanie Rinderle-Ma



## Abstract

In this thesis, I present a semantic merging algorithm for Business Process Models (BPM) that enables a widespread use of version control systems in the field of business process modeling. The proposed algorithm is not just a linear textual merge (as most generic merge tools are) but makes use of many semantic aspects of BPM in order to maximize the number of automatically resolved conflicts and thus to improve the efficiency of the algorithm.

In a publicly available<sup>1</sup> testset of 5792 test cases, my semantic merging algorithm has shown 81% of correctly merged instances and only 19% with unresolved conflicts (that need manual human resolving) compared to a generic merging algorithm with only 31% of correctly merged instances and 68% unresolved conflicts (and 1% of other cases).

## Zusammenfassung

In dieser Arbeit präsentiere ich einen semantischen Merging-Algorithmus für Geschäftsprozessmodelle (BPM), der eine weit verbreitete Nutzung von Versionskontrollmechanismen in dem Gebiet der Geschäftsprozessmodellierung ermöglicht. Der vorgeschlagene Algorithmus ist nicht nur ein linearer textueller Merge (wie die meisten generischen Merge Algorithmen), sondern nutzt auch viele semantische Aspekte der BPM, um die Anzahl der automatisch gelösten Konflikte zu maximieren und somit die Effizienz des Algorithmus zu erhöhen.

In einer öffentlich zugänglichen<sup>1</sup> (letzter Aufruf: 9.8.2018) Testsammlung mit 5792 Testfällen hat mein Algorithmus eine Quote von 81% richtig vereinigten Instanzen und nur 18% ungelösten Konflikten aufgezeigt (die manuelles Lösen durch Menschen brauchen) - verglichen mit einem generischen Merging Algorithmus mit nur 31% von richtig vereinigten Instanzen und 68% ungelösten Konflikten (und 1% anderer Fälle).

---

<sup>1</sup><https://github.com/misuliak/testset-process-models> (last access: 9.8.2018)



# Contents

<b>Acknowledgement</b>	<b>5</b>
<b>Disclaimer</b>	<b>7</b>
<b>1 Introduction</b>	<b>9</b>
1.1 Motivation . . . . .	9
1.2 Research Questions . . . . .	10
1.3 Structure of Thesis . . . . .	10
<b>2 Related Work</b>	<b>12</b>
2.1 Representations of Process Models . . . . .	12
2.1.1 Declarative . . . . .	13
2.1.2 Imperative - Graph . . . . .	15
2.1.3 Imperative - Tree . . . . .	20
2.2 Diff & Merging Algorithms . . . . .	22
2.2.1 Diff Overview . . . . .	23
2.2.2 Merge Overview . . . . .	28
2.3 Process Evolution . . . . .	30
<b>3 Conceptual Design</b>	<b>34</b>
3.1 Preparing Sample Instances . . . . .	34
3.1.1 Information Contained in Instances . . . . .	36
3.1.2 Relevant Information . . . . .	39
3.1.3 Regrouping Instances by Students . . . . .	40
3.2 Determining Significant Instances . . . . .	42
3.3 Proposed Diff Algorithm . . . . .	47
3.3.1 Diff of Structure . . . . .	47
3.3.2 Diff of Endpoints . . . . .	53
3.3.3 Diff of Parameters . . . . .	53
3.3.4 Complete Diff Results . . . . .	54
3.4 Proposed Merge Algorithm . . . . .	55
3.4.1 Desired Merge Algorithm . . . . .	56
3.4.2 Semantics of BPM Merging . . . . .	57
3.4.3 Handling Conflicts . . . . .	59
3.4.4 Proposed Merge Algorithm . . . . .	71
<b>4 Implementation and Evaluation</b>	<b>76</b>
4.1 User Interface and Usage . . . . .	76
4.2 Functional Evaluation . . . . .	80
4.3 Test Cases . . . . .	81
4.3.1 Testing Semantic Algorithm . . . . .	84
4.3.2 Testing Generic Algorithm . . . . .	85
4.4 Qualitative Evaluation Based on Test Cases . . . . .	85
4.4.1 Result Types . . . . .	87
4.5 Quantitative Evaluation . . . . .	90

<b>5</b>	<b>Discussion and Outlook</b>	<b>91</b>
5.1	Contribution . . . . .	91
5.2	Conclusion . . . . .	91
5.3	Integration with Existing BPM Environment . . . . .	91
	<b>List of Figures</b>	<b>93</b>
	<b>List of Tables</b>	<b>95</b>
	<b>References</b>	<b>96</b>

## Acknowledgement

I wish to thank my supervisor and tutor for constructive feedback, my parents for the patience and my girlfriend for the support with writing this thesis.





## Disclaimer

Throughout this thesis, I am mostly using the we pronoun - e.g. "we will implement" or "we have shown that". This is purely for the purposes of better readability, since it is very natural and convenient to read such a written text. Nonetheless, I have performed all the work for this thesis myself.



# 1 Introduction

Version control is a widespread feature used in software development around the world. It allows multiple developers to work at the same piece of code at the same time, each of them submitting their changes and downloading changes from other developers (these changes get merged together using merging algorithms). By tracking changes made by each stakeholder, it allows for the merging of concurrent changes. It is a very convenient system that allows a desired number of people to work at the same document at the same time without any worries and is so widespread that it is used basically in all software development projects involving more than one developer.

However, in the field of Business Process Modeling (BPM), concurrent work is not possible. The versioning of BPM projects works solely on the basis of overwrite. If multiple workers want to work at the same BP model, they either have to send each other the most updated version every time they make an update (which is extremely inconvenient), or each time one person opens a model it needs to be locked for others to use, or when saving an updated version of the same file at the same time, they may remove the changes made by the other person.

In this thesis, I want to bring the advantages of version control systems and of concurrent change tracking to the field of business process modeling. This way, it will be possible for multiple people to work at the same business process model simultaneously, submitting their changes and downloading changes made by others.

This thesis proposes a semantic merging algorithm specially designed for BP models that will allow for automatic merging of concurrent changes. It will handle the specific semantic aspects of BP models and make use of them in order to correctly merge them and especially to maximize the number of conflicts that can be resolved automatically, without the need of human intervention. A conflict occurs every time two people make changes at the same place of the document (e.g. the same line of a text file) and where the merge algorithm can usually not determine, whose change should be incorporated (and needs a human to make the decision). By analyzing the semantics of BPM, a lot of useful information can be extracted that will help the merging algorithm make automated decisions in many conflict cases - so that a human does not need to jump in.

In addition to proposing and deeply describing a semantic BPM merge algorithm, a working prototype will be developed and extensive testing will be performed to ensure its functionality and to compare the efficiency of the proposed semantic algorithm with a generic textual algorithm (that does not take any semantic aspects into consideration).

## 1.1 Motivation

When speaking of merge algorithms, one of the most important aspects is handling conflicts (i.e. when both people make changes at the same location of the document). Implementing merging of two different spots of a document is relatively simple - you just apply the first change at the first location and the second change at the second location. However, when there are changes at the exact same place, it is hard to decide which of the two changes should be

applied. Therefore, conflict resolving mostly needs to be performed by human workers.

This is especially true for the case of version control in the field of software development - especially in terms of generic programming languages such as C or Java. Programming code is a very complex structure where a line might be neglectable (such as a comment line or a declaration of a variable that is never actually used - if such lines were accidentally removed, the program would still work as expected) but might also perform very important tasks essential to the program (if such lines were accidentally removed, the program would not work properly). It is therefore extremely hard to determine, whether and how two changes can be merged at the same line of a programming code. Naturally, some academic work has been done and certain syntactical and semantic aspects may in certain cases be considered [1] but in real life, automated conflict resolving for generic programming languages is very limited.

As opposed to generic programming code, the complexity of business process models is very limited. It has a precisely defined structure with a limited number of grammar constructs (such as activity or decision) with each of them having a certain number of possible parameters. Due to the low complexity of the grammar, it is very easy to predict what each entity performs and it is known how each of them should be structured. Also, one can make use of a lot of semantic information stored within the instances. By focusing merely on merging business process models, a merging algorithm can be very specialized and make use of all the semantic information available.

Because of the much smaller complexity of BPM, it should therefore be possible, without extreme effort, to perform a semantic analysis of context and to achieve an improved automated merging. In other words, it should be possible to analyze all the entities of a business process model instance, how they relate to each other and in which ways they affect each other (e.g. whether one activity produces some artifact essential to another activity). Consequently, it should also be possible to improve the automation of merging, especially in terms of automatically resolving conflicts, so that fewer human decisions on how to resolve conflicts are needed.

## 1.2 Research Questions

Based on the motivation (see chapter 1.1), we can define following research questions:

- Can a semantic BPM merge be built?
- How much can it improve the automation of conflict resolving?

## 1.3 Structure of Thesis

The thesis is organized into a number of chapters. Chapter 1 *Introduction* contains an introduction to the topic, the motivation for this thesis and this short description of the structure of the thesis.

Chapter 2 *Related Work* contains theoretical information about the important academic fields for this thesis. Three topics are analyzed: Representations of process models (the ways in which process models are represented and stored),

Diff & merging algorithms and Process evolution. Three general ways of representing business process models are described. Declarative modeling paradigms specify constraints and restrictions for models without explicitly defining each step of a process model. This, on the other hand, is what imperative languages do. This thesis describes both graph based imperative modeling languages which define a process model as a graph (such as BPMN and Petri nets) and tree based languages which define a process model as a structured tree (such as Refined Process Structure Tree).

The thesis then describes the fundamentals of diff algorithms (which look for differences between two versions of a file or between two files) and merge algorithms (which try to merge an original file with changes from two different people made to that file), important aspects as the LCS (Longest Common Subsequence) and makes an introduction into the topic of Process Evolution and the aspects of change in process models.

Chapter 3 *Conceptual Design* is the main part of the thesis. It describes not only the conceptual design of the proposed semantic diff & merge algorithms, but also all the proceeding steps made to the sample data. At first, the test process model instances are prepared - describing what information is stored within them, which of that information is relevant for us and how the script regroups the instances, so that they are stored for each student separately. The next step is determining significant instances, where another script goes through all the instances of a student and finds out, which of them introduce any kind of difference (compared to the previous instance).

Based on the list of significant instances, the proposed semantic diff script is coded that loops through them and performs the diff for each significant instance - both a diff of structure, endpoints and parameters. Finally, the proposed semantic merge algorithm is described. Here, thoughts are collected and evaluated on what characteristics the algorithm should have, on all the relevant semantic aspects of BPM that can be made use of in the algorithm and on handling conflicts. Handling conflicts is a very important topic for this thesis, since the main purpose is to provide a merge algorithm that will be much more effective in automatically resolving conflicts, compared to a generic merge algorithm.

Chapter 4 *Implementation and Evaluation* describes the User interface of the scripts and evaluates whether the semantic merge algorithm does indeed improve the automation of conflict resolving. A functional test is applied, as well as a high scale qualitative evaluation (based on a large number of real process models). During the execution of the large scale tests, also performance data is collected, which is used for quantitative evaluation of the algorithm.

The final Chapter 5 *Discussion and Outlook* describes the contribution of this thesis, provides a conclusion and a description of the possible future extensions of this work, especially in connection with standard formats.

## 2 Related Work

Let me first elaborate the scientific work related to the topic of this thesis. As this thesis is named Versioning and Evolution in Process-aware Information Systems I want to make a brief introduction, as well as state-of-the-art description of topics highly relevant to the topic.

First, I will describe the theoretical background of various types of process model representations and describe each of them in an understandable way. I will write about the main differences between declarative and imperative representations, their purpose, advantages and disadvantages and try to make a clear differentiation. As of the imperative representation, I will also explain the differences between graph and tree and describe the most important representatives.

Second, I will give an introduction into diff and merging algorithms, which are essential to this thesis. Since the purpose of this thesis is to introduce efficient diff and merging prototypes for process models, examining the theoretical fundamentals that lie beneath such algorithms is crucial. Merging algorithms, as will be shown, are widely being used in software development, online office collaboration and other areas; and the purpose of this thesis is to also introduce them into the field of business process modeling.

### 2.1 Representations of Process Models

Business process models are a powerful way to represent, analyze and optimize processes that occur daily in businesses - both in corporate and governmental context. A business process is an abstraction of business interactions between agents represented by a specific flow of work activities across time and place, "with a set of business goals and a set of physical and informational inputs and outputs." [2]

In order to visualize or represent models, a certain modeling language is needed. Such a language defines the elements which can occur in a process (e.g. activity or decision in a BPMN), their relationships and all possible constraints. In broader terms, a business process modeling language defines the whole way, how a certain business process can be written down, so that it is understandable, measurable, reproducible and analyzable for further improvements.

As such, the modeling language is very important, because it comes with a set of features and constraints and each of the languages has certain advantages and disadvantages and is more suitable for certain use cases.

Generally, business process modeling languages can be divided into two main categories: Declarative and Imperative [2, 3, 4], with various representatives in each category.

**Declarative** modeling paradigms focus on what should be done in a process, without actually stating how it should be executed [3, 2, 5]. They specify constraints, business rules, event conditions and other possible expressions that define the relationships between activities in a process, as well as their properties [2]. In effect, all possible paths within a process are implicitly defined by the rules and constraints in a way that cannot violate them.

One could say that declarative approaches focus on the logic of processes, that is represented by business rules and constraints. They enable us to describe key qualities of objects and actions, as well as how they relate to each other [3].

However, declarative approaches do not state a precise definition of the control flow of a process and provide greater flexibility.

**Imperative** modeling paradigms are mainly used in common process modeling tools nowadays and, as opposed to declarative paradigms, they provide an exact definition of the control flow within a process [2]. Thus, there is a definite number of possible states and an exact end state(s).

Imperative approaches allow for a dual view. One is the state space, which defines all the possible states of an object in real world. The other view is the transition space, defining the various actions, events and changes of the respective process [3]. Based on work by Petri [6], Holt formally constructed a mathematical framework, which describes the relationship between state space and transition space in the theory of Petri nets [7]. Put into simplified terms, the two main entities in imperative modeling languages are activities and transitions between them (i.e. nodes, directed arcs).

As Pesic stated in his PhD thesis, " [Imperative] models take an 'inside-to-outside' approach: all execution alternatives are explicitly specified in the model and new alternatives must be explicitly added to the model. Declarative models take an 'outside-to-inside' approach: constraints implicitly specify execution alternatives as all alternatives that satisfy the constraints and adding new constraints usually means discarding some execution alternatives. " [4]

### 2.1.1 Declarative

As has been stated, declarative business process modeling approaches focus on defining what should be done in a business process, without precisely describing, how it should be done [3, 2, 5]. As opposed to imperative paradigms, they don't explicitly specify the exact path(s) within a process, but rather a set of rules that must not be violated - such as constraints, business rules, event conditions or other expressions. As long as none of the applied rules are being violated, any paths and transitions are possible [2].

Every business process can be modeled in by defining its state space and transition space [3]. In the case of declarative modeling, the transition space is represented by the set of business rules constraining the movements in the state space [8, 5].

State space is the discrete set of possible states in a business process, i.e. all the possible distinct states that can occur in a process. A state is a "specific configuration of the facts about entities (e.g. activities) in a state space corresponding to a specific situation of a business proces" [2]. It can also be defined by a start state and a number of end states.

The possible transitions in a process can be defined by activity space transitions. These are changes in the life cycle of an entity, i.e. activity, in a process. Any state transitions are possible only if no business rules are being violated. Thus, declarative modeling approaches define the possible transitions in an implicit way, allowing for any transition that complies with the rules (as opposed to implicit, where each possible transition is explicitly defined). This way, declarative models allow for much higher degree of flexibility than implicit models. Some declarative languages, such as ConDec [5], do also allow modeling of very strict models that behave like implicit.

Numerous declarative approaches can be found in literature:

- Reichert and Dadam [9, 10] describe an advanced Workflow Management System (WfMS) called ADEPT(flex), which allows end-users to change process models at run-time. Working with adaptive WfMS enables changing the process model either at a general level (i.e. all instances of the process model are updated) or at a specific level (the change is only deployed to one instance). ADEPT uses a very complex workflow engine, allowing users inserting, deleting and moving tasks at runtime, while preventing non-permissible structural changes.
- van der Aalst, Weske, and Grünbauer [11] define the formal characteristics of the case handling approach. It is one of the few declarative methods that comes from commercial WfMS, mainly the FLOWer WfMS. This approach enables users to complete, redo and skip activities within a set of constraints. For each activity, the user needs to define whether particular data objects are obligatory for the activity to be completed. In effect, a big part of the semantics of case handling lies in the mandatory constraint.
- S. W. Sadiq, Orlowska, and W. Sadiq [12] define the constraint specification framework with the order, fork, sequence, exclusion and inclusion constraints, as part of the state space. In their work, they prove it can be beneficial to use both declarative and imperative features in modeling business processes. In their approach, a model can contain (apart from predefined activities and control flow) so called pockets of flexibility, consisting of activities, sub-processes and so-called order and inclusion constraints. Whenever they get triggered, they are executed by a human end user through a so called build activity.
- Pesic and van der Aalst [5] define a declarative modeling language ConDec, based on Linear temporal logic (LTL). In addition to basic logical operators, the language introduces temporal operators nexttime, eventually, always, and until. The language uses existence, relationship and negation constraints (also referred to as formulas or templates). Existence formulas define how many times a task can be executed in a process. Relationship formulas specify the ordering and dependencies between activities (e.g. activity B can be executed after task B). Finally, negation formulas specify the existence of one activity excluding another activity. Besides an LTL representation of the process, ConDec also offers a graphical representation.

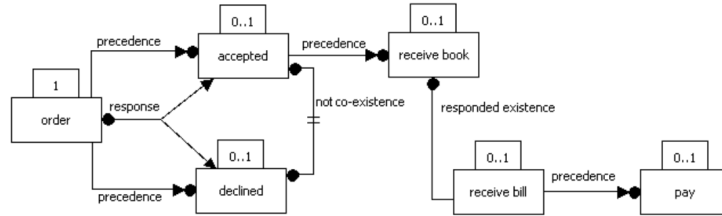


Figure 1: Sample ConDec Process (Purchasing a Book) [5]

- Goedertier and Vanthienen [13] define the language PENELOPE for modeling contracts with so-called temporal deontic assignment rules. These



rules are obligations and permissions of agents to execute activities in bounds of given deadlines. The state space of PENELOPE comprises event history and the system time and there are two transition types: performing activities and deadline violations.

- There are a number of other declarative modeling approaches, such as artefact-centric process modelling [14] or BPCN [15]. The former models preconditions and effects on artifacts by executing services and specifies business rules to define when specific services are to be executed. The latter features graphical notation and defines selection and scheduling constraints.

### 2.1.2 Imperative - Graph

As opposed to Declarative business process modeling, the Imperative modeling approach is much more precise and explicitly specifies all possible paths within a process [2]. As the name already suggests, this approach is about defining all relevant events, constraints, activities and their order and interactions. This approach is widely used in business process modeling tools nowadays and is only gradually being supplemented by the declarative methods [2].

One can divide imperative modeling into graph-based and tree-based modeling languages. The first and more common group of modeling languages define process as a directed graph with a beginning and one or multiple ends, consisting of activities that get executed during the process and with directional arcs connecting the activities in the order that they get executed and other elements, such as decisions or parallelisations [2, 16]. The other group of imperative modeling languages that will be described in the next sub-chapter, are tree-based approaches. Here, a process is depicted as a structured tree going from top to bottom, with every decision represented as a branch growing from the main trunk (since after its execution, it gets back to the main trunk anyways). [17]

There are a number of various imperative graph-based modeling approaches. Among the most important are Business Process Model Notation (BPMN) [18], Event-driven Process Chains (EPC) [19], modeling based on Petri Nets [6] and Unified Modeling Language (UML) activity diagram [20]. Let us now make a brief overview of BPMN, EPC and Petri nets.

**2.1.2.1 BPMN** Business Process Modeling Notation (BPMN) was first developed in 2004 by the Business Process Management Initiative (BPMI) Notation Working Group, as a consensus for a standard business process modeling notation. This Workforce represented many of the important players of the business process modeling community and was an important step towards standardization. Some of the methodologies reviewed during the process were UML Activity Diagram, UML EDOC Business Process, IDEF, ebXML, BPSS, Activity-Decision-Flow (ADF) Diagram, RosettaNet, LOVeM and Event-driven Process Chains (EPC) [16].

The significant heterogeneity of imperative business process modeling approaches before the introduction of BPMN has disabled a wider adoption of interoperable systems, which was one of the reasons to introduce the new standard. Another reason was that before, there was a strong separation of mod-

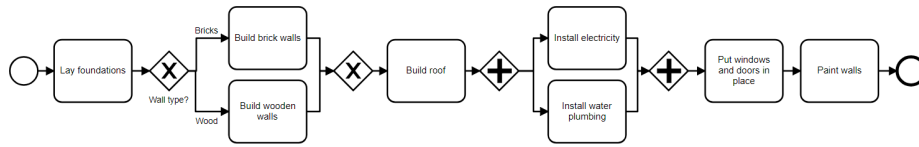


Figure 2: Sample BPMN process - Building a house

eling methods for people and those needed for IT systems to be able to run those processes. With the introduction of BPMN, an automatic translation of BPMN models into BPEL executable processes has become possible [21, 22]. The BPMI was later taken over by Open Management Group (OMG) in 2005 and in 2011, BPMN version 2.0 was released with many new features. [18, 20]

In BPMN, a business process model (BPM) is a graphical representation of the process, describing the precise flow of events within the process. The graphical representation is easily understandable both for business analysts, as well as technical developers. There are four main categories of elements: [20, 18]

- Flow Objects
- Connecting Objects
- Swimlanes
- Artifacts

#### Flow Objects

**Event** is visualized by a hollow circle and represents something that happens



Figure 3: BPMN Sample Flow Objects

From left to right:

Start Event, End Event, Activity, XOR Gateway, Parallel Gateway

during the execution of a process. In most cases, it is the Start and the End of the process, but there is also a third type of events: Intermediate. The center of the circle is empty to allow to visualize additional information about the event.

**Activity** is the main construct of a BPMN model and represents a single coherent unit of work that is done in a process. Activities are the most common objects in a model, they are represented by a rounded-corner rectangle and can either be a single task or a sub-process (an entire process embedded into an activity to allow for greater abstraction of the overview model, with detailed representation within the sub-process).

**Gateway** is represented by a diamond shape and is used as a separator and connector for decisions. The decision criteria, as well as options for each path are usually displayed.

**Connecting Objects** The flow objects within a process model are connected to show their relations. These connectors are directional and include:

**Sequence Flow** (solid black line) shows the order, in which activities are executed - starting in the first activity with the pointed arrow towards the second activity (or gateway, event etc.).

**Message Flow** (dashed line) shows the flow of messages between participants of a process.

**Association** (dotted line) shows associations of data, text and other artifacts with certain activities.

**Pools, Swimlanes and Artifacts** Pools and Swimlanes help to visually assign activities to the responsible participants and show the task and message flow between participants.

**Data Object** show interactions with data in the course of the process. This is a very useful information for further development and execution of Workflow Systems.

**Group and Annotation** don't affect the sequence flow and are used solely for documentation purposes.

**2.1.2.2 EPC** Event-driven Process Chains (EPC; Ereignisgesteuerter Prozessketten (EPK)) were introduced by [19] in 1992 as a development project involving SAP at the Institute for Informations Systems (IWi) at the University of Saarland, Germany [23]. EPC are used in market-leading tools, such as SAP R/3, ARIS, LiveModel/Analyst and Visio. As of 1999, SAP R/3 was the leading ERP system being used in over 7500 companies in 85 countries and ARIS was leading the Business Process Reengineering market with over 7000 licences [24].

There are three main building blocks of an EPC [19, 23, 24]:

- **Functions**

A function in EPC is equal to an activity in a BPMN - it is a task or a working package that needs to be executed. It is represented by a rounded-corner rectangle.

- **Events**

Events happen before or after an activity as a condition of the environment or a result of an activity. Usually an event happens at the end of an activity (e.g. Goods delivered) and serves as a precondition to the following activity. An event is represented by a hexagon.

- **Logical connectors**

Connectors are used for branching the process and control its flow. There are three types of connectors:  $\wedge$  (and), XOR (exclusive or) and  $\vee$  (or). They are represented by a circle with the respective symbol inside.

These elements are connected using directed arcs, just like in BPMN. Each EPC starts and ends with one or more events. An EPC can also consist of multiple embedded EPCs. An event can never be followed by an event, just like an activity cannot be followed by another activity. It is also an important rule that each event and activity have only one incoming and one outgoing edge.

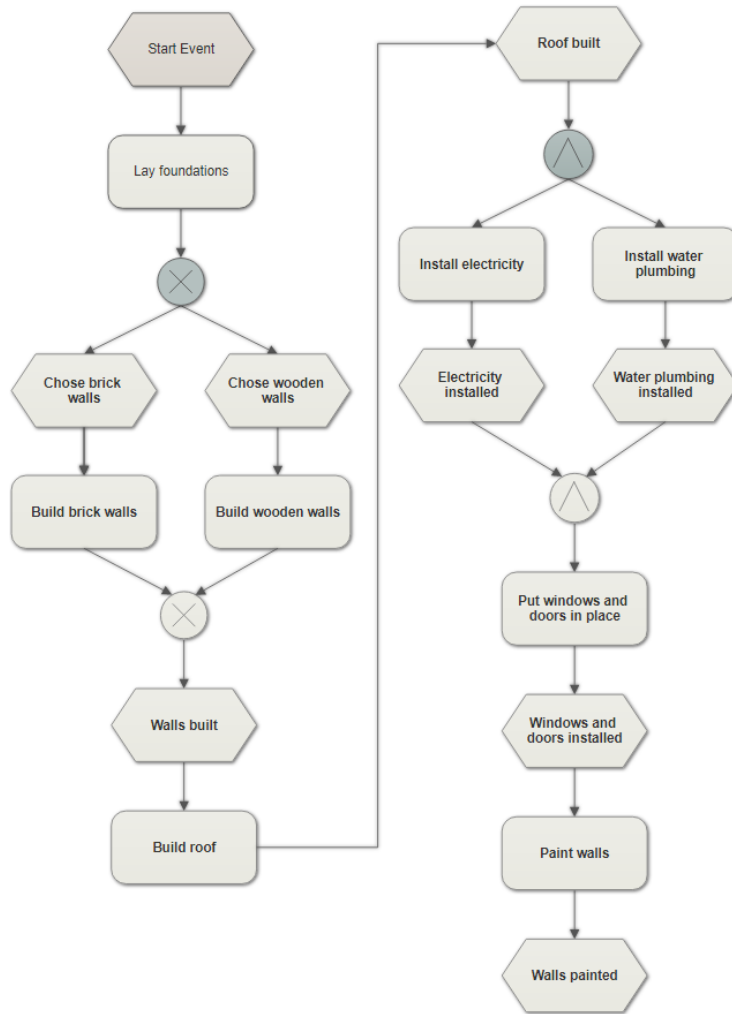


Figure 4: Sample EPC process - Building a house  
The sample process from Figure 2 (BPMN) translated to EPC notation

**2.1.2.3 Petri Net** Petri Nets are another useful method for visualizing processes that also offers great methods for analyzing the concerned business processes [25]. Their theoretical foundations were laid in Carl Adam Petri's 1962 dissertation *Kommunikation mit Automaten* submitted to the Faculty of Mathematics and Physics at the Technical University of Darmstadt, West Germany and worked on as a scientist at the University of Bonn [26]. His work came to the attention of A. W. Holt in the United States, who in 1980 developed a mathematical model for Petri Nets [7] and led many early developments and applications of Petri Nets.

Petri Nets are also referred to as Place Transition Nets [27] due to their main components - places and transitions. As opposed to previous modeling methods described in this thesis, they have a strong mathematical model as a basis and are thus not only a visual modeling tool, but also a powerful analytical tool applicable to many areas. They are useful for modeling and analyzing systems



Figure 5: EPC Building Blocks  
From left to right:  
Function, Event & Connectors - XOR, OR, AND

that are 'concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic' [25]. However, one needs to be careful with the level of details in a Petri Net - their common problem is the complexity and even the model of a mid-size system can become too large [25].

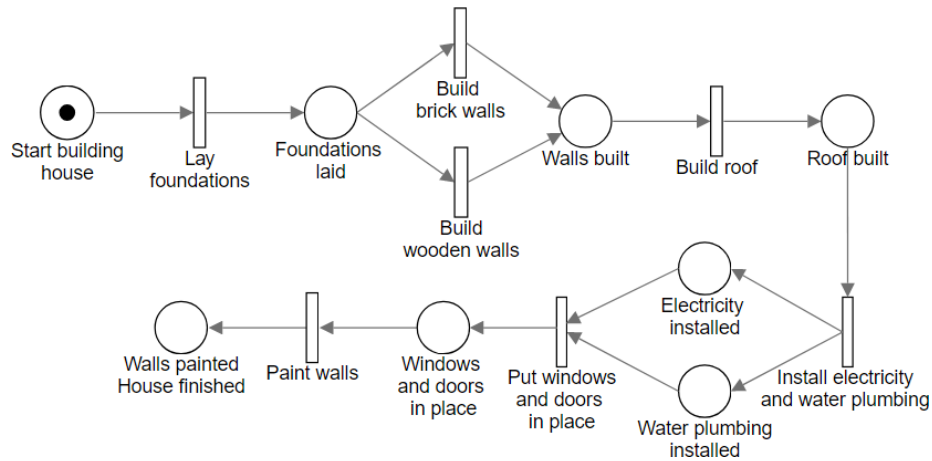


Figure 6: Sample Petri network - Building a house  
The sample process from Figure 2 (BPMN) translated to Petri net notation

A Petri Net consists of following components: [27, 25]

- **Places**

Places are represented by circles and they are possible states of a system. A place can accommodate token(s), where a Petri Net can be with or without capacity constraints - if so, a place has a capacity of a discrete number of tokens. Places situated before a transition are called Input places and places situated after a transition are Output places.

- **Transitions**

Transitions are certain events that cause a change of state and they consume tokens from the input places and produce tokens for the output places (this change of state is called firing of a transition). Due to this conversion, a transition is always preceded by one or more places and followed by one or more places. They are represented by rectangles.

- **Arcs**

Arcs in a Petri Net are directed and go either from a place to a transition or from a transition to a place. There can be a number next to the arc (weight) indicating the number of tokens consumed/produced by that

transition, which are either used up in the input place, or added to the output place.

- **Tokens**

A token is visualized as a small black dot. Each state is determined by the configuration of tokens, i.e. how many tokens are situated in each place. In a model, there can be just one token at a time, as well as multiple tokens.

A firing of a transition in a Petri Net depends on the input conditions and token availability in the input places. Thus, a decision (also called conflict, choice) in a Petri Net can be made by having a place followed by multiple transitions where only the relevant one fires. On the other hand, a transition followed by multiple places indicates a parallel execution, or a concurrent system. Synchronization in a concurrent system can also be modeled by making multiple places end up in a single transition.

With the use of Petri Nets, it is possible to analyze following properties and problems of concurrent systems: [25]

- **Reachability**

Reachability describes whether a particular state can be reached from another one. Each possible state of a net can be denoted as  $M_n$  with  $n=0,1,2,\dots$  and  $M_0$  being the initial state. A state  $M_n$  is reachable from  $M_0$  if there is a firing sequence, which will lead to the state  $M_n$ .

- **Boundeness**

A graph is  $k$ -bounded (or just bounded) if the number of tokens in each place does not exceed a finite number  $k$  for any marking reachable from  $M_0$ . A 1-bounded Petri Net is safe.

- **Liveness**

A Petri Net is live if, whichever marking has been reached from  $M_0$ , it is possible to eventually fire any transition of the net. This property guarantees a deadlock-free operation of the system. Since it is usually too complicated to prove liveness of complex systems, the liveness condition is being relaxed and classified into  $L_0$  to  $L_4$  live (with  $L_0$  being dead and  $L_4$  being fully live) [25].

### 2.1.3 Imperative - Tree

The imperative modeling methods described in previous subchapter are all graph-based methods. There are, however, numerous use cases where a block-based modeling method is desired. Probably the most important case is the conversion of graph-based BPMN models into block-based executable BPEL (Business Process Execution Language) processes [17]. It is also, e.g., possible to use this conversion to significantly speed up control-flow analysis of process models and in effect reach linear time [28]. In a case study with 340 real business processes in the IBM WebSphere Business Modeller in 2007, they reached a time under one second for many of the processes and so concluded, that such a control-flow analysis can be performed in real time, to eliminate any errors while the process is still in modeling phase. [29] have shown that it is also possible to

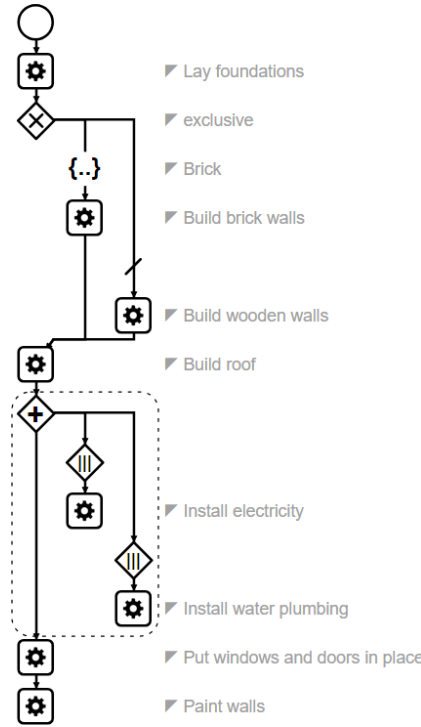


Figure 7: Sample CPEE process (based on RPST) - Building a house  
The sample process from Figure 2 (BPMN) translated to RPST notation

use the graph-to-block conversion to discover differences between two business process models without the previous presence of a changelog.

[17] have developed a graph-to-block based conversion called Refined Process Structure Tree (RPST), which represents a business process model as a tree. It is based on previous work by [30] of triconnected components for structural analysis of biconnected flow graphs (also known as SPQR tree) and extended by providing a specification of the tree structure (i.e. the canonical fragments), providing modularity and finally providing a more refined parse tree [17]. Later, [31] have improved the RPST by providing a simpler way to compute the RPST and by lifting the restriction of RPST of only being applicable to graphs with a single source and single sink.

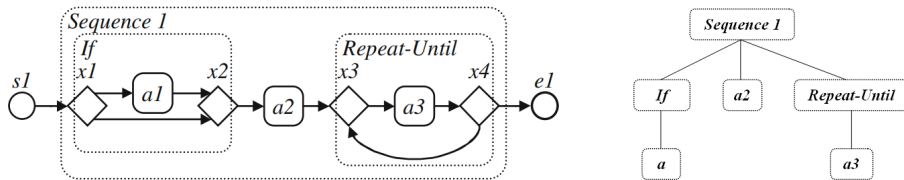


Figure 8: Decomposition of a sample BPMN into blocks [17] and the resulting RPST

The RPST is based on a decomposition of a workflow graph into smaller sub-workflows (also called blocks) which have a single entry and single exit

of control (SESE), using SESE *nodes* (as opposed to *edges* in previous work). We can see such a decomposition in Figure 8. Depending on their syntactical function, these blocks are assigned a category, such as sequence or repeat-until. The decomposition to RPST has following characteristics:

- **Unique**

The same BPMN always translates to the same BPEL process.

- **Modular**

It is desirable that a local change in a BPMN model also leads only to a local change in the BPEL process. In other words, changing a block in BPMN should result in only changing the single block in the BPEL process. Such a decomposition is called modular and the RPST fulfills this criterion.

- **Fine**

It is desirable for the parsed tree to discover as much structure as possible. In other words, the decomposed blocks should be as fine as possible. This allows more BPMN models to be parsed into BPEL in a structured way, creates more readable BPEL processes and makes debugging the resulting process easier, since a potential error can be detected in a small local block instead of a big one. RPST delivers finer blocks and is hence also called *Refined* PST.

Another non-functional characteristic of the RPST is that it can be computed in linear time [17], which means it can be computed in real time any time needed.

## 2.2 Diff & Merging Algorithms

A crucial topic to this thesis is finding differences between different documents (e.g. text files, models), detecting the changes made to them and merging together multiple changes of the same original document into a new version. The first part of this process, finding differences in files, is handled by so-called **difference algorithms** (abbreviated as *diff*), which have been studied thoroughly, especially in the 1970s and 80s [32, 33, 34, 35, 36]. The most famous representative of textual difference programs is the Unix-based utility *diff*, whose first version was shipped in the 5th edition of Unix in 1974 and the respective academic paper published in 1976 by [32].

There are numerous applications for the use of difference search, i.e. file comparison [33, 35]:

- Difference programs show how different versions of text files differ, which is essential to the versioning of files. This is used daily by teams of programmers worldwide working simultaneously on the same project, mainly with the use of subversion or git systems [37, 38, 39, 40]. Subversioning also helps programmers trace the evolution of code and create test cases for the changed fragments of code.
- Frequently changed documents can be economically stored as one base version and a set of changes for each of the following versions, instead of a whole new copy of each version of the document. In practice, a typical



changeset takes up less than 10 per cent of the whole document, so one can store 11 revisions of a document in less space than would be taken for 2 whole revisions [33].

- Distribution of changes to files and data is most economically performed by transferring only changes instead of entire revisions. So called *update decks*, *deltas* or *patches* are applied to transform the old version of data into the new version. An important application are screen drivers, which process the graphics and only transform changes to the display to rerender them, as well as video compression, which makes big use of only storing visual differences between the frames.
- Spelling correction systems use difference algorithms to match words and find differences
- Genetics makes great use of difference algorithms by comparing long molecules consisting of nucleotides or amino acids.

In this thesis, I will use difference algorithms to find differences between versions of business process models and, in the next step, use merge algorithms to merge changes into a common version.

### 2.2.1 Diff Overview

Detecting differences between two files is a necessity for many applications, as was shown in the previous sub-chapter. There are byte-oriented approaches comparing every single byte (useful for comparing binary documents, such as compiled programs [34]) and approaches regarding a file (e.g. document, a code repository, a DNA chain) as a series of lines, where it is basically only necessary to compare lines with one another (such as [32]). In this thesis, I will only deal with the latter approach.

When comparing two files with each other, it is desired to obtain a minimal list of changes necessary to convert one file into the other. This minimal cost of a sequence of editing steps is also referred to as the edit distance [36, 34]. The most common programs and algorithms work with three operations: insert (or append), delete and change. Insert means that at a certain point of the old file, some new line(s) have been added, delete means that some line(s) of the old file were deleted and change means that some line(s) in the old file have been changed or replaced - basically change is a shortcut for a combination of insert and delete, which was introduced for better overview and more efficient use of storage (instead of storing 2 operations, only 1 needs to be stored). The first step of efficiently comparing two files is finding a so-called Longest Common Subsequence (LCS; see next subchapter) and in turn, determine the list of change operations.

In most cases, there is not a single correct answer to the question how two files differ, but for a feasible and reliable solution, it is important to indeed obtain a *minimal* list of changes. Simple algorithms can often determine a set of changes necessary to convert one file into the other, but in an inefficient way, that means with too many unnecessary operations. The simplest method would go through the two files matching equal lines and as soon as it finds a different line, just look forward to a pair of equal lines, connect them and continue. The

example below [34] demonstrates that such a straight-forward approach does not produce optimal results.

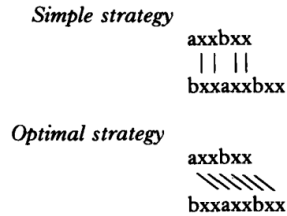


Figure 9: Example of comparing two files [34]

In this example, only three lines are appended to the original file at the beginning (bxx), so a good algorithm would only report three line changes, whereas a very simple algorithm might detect fewer common lines and in effect report more unnecessary change operations.

The standard program for file comparison is the Unix-based utility diff. This was developed in the 1970s by Douglas McIlroy, with the respective scientific work known as Hunt–McIlroy algorithm and published by James W. Hunt and Douglas McIlroy in 1976 [32]. The original diff utility was shipped in the 5th edition of Unix in 1974. Later, especially in 1980s, the original algorithm was optimized for time and memory requirements by [34, 35] whose improvements were used in latter versions of the program and by [36], who came up with similar optimizations independently at the same time. An alternative way for converting one file into another was also presented by [33], who however doesn't take into account the order of lines and also doesn't mind using the same line multiple times. For these reasons, this approach is not suitable for our intentions of comparing two business process models.

**2.2.1.1 LCS** Finding a shortest edit script for transforming a file A into a file B goes hand in hand with the problem of finding a Longest Common Subsequence (LCS) [35]. LCS is the longest list of lines in their right order present in both investigated files. In other words, it comprises the most lines that can be found both in A and B in the right order and these lines don't need to be changed. As has been said, multiple common subsequences can usually be found at the same time, but it is important to find the *longest* common subsequence. [35] shows that this task of finding a LCS is equivalent to finding a shortest/longest path in an edit graph.

For a better overview, the two strings A and B can be visually shown in a graph. Let us consider an example where file A consists of *abcabba* (with each letter representing an entity, such as a line of a text file or acid in an RNA strain) and file B of *cbabac*. This can be graphically represented as follows [32]:

File A is displayed at the x axis and file B at y axis. In this graph, the contents of files A and B go from left bottom to right top, there are also often graphs going from left top to right bottom. The numbers 1 to 6 (or 7, respectively) represent the position of a letter in a file. The goal in finding a LCS is to start moving from 1,1 to the opposite corner and find as many common subsequences (in this case letters) as possible. The dots represent all matching

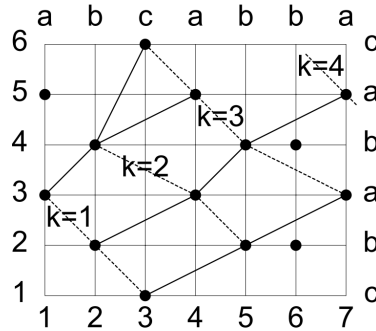


Figure 10: Common subsequences [32]

letters. It is possible to move right, up and diagonally, where a diagonal move can only be made when a matching pair is found.

It can be e.g. seen that at 1,3 both files have the letter *a* and at 2,4 they both have the letter *b*. It can also be seen that there are multiple variations of common subsequences possible, shown by the solid lines connecting the dots. Therefore, dotted lines diagonally cutting through the dots, visualize the number of elements in the common subsequences. This way, it can easily be seen that there is only one common subsequence with as many as 4 elements. This is the Longest Common Subsequence: *baba*.

In the next stage, this pair of files A and B can be converted into a matrix of edit distances (here, the direction is from left top to left bottom): [34]

0	1	2	3	4	5	6	
1	2	3	2	3	4	5	<i>a</i>
2	3	2	3	2	3	4	<i>b</i>
3	2	3	4	3	4	3	<i>c</i>
4	3	4	3	4	3	4	<i>a</i>
5	4	3	4	3	4	5	<i>b</i>
6	5	4	5	4	5	6	<i>b</i>
7	6	5	4	5	4	5	<i>a</i>
	<i>c</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	

Figure 11: Matrix of edit distances [34]

Here, the flow is from left top to right bottom. Numbers in the matrix represent the edit distance at the current position, i.e. how many edit operations are necessary until this point to bring the two files into compliance. The goal is to find a path to the end of both files with a minimal number.

After determining the LCS, the diff program compares all lines of A and B with the LCS and records any differences as the change operations.

**2.2.1.2 Diff and Patch files** The original diff utility, as well as any other file comparison tool, find the differences between an original file A and modified file B and display it in a readable form. The output file is called a diff, or

sometimes a patch file (since it can be applied as a patch to the original file in order to convert it to the new file).

Let us see an example output of the Unix diff utility (Note that the real diff utility output is not colored. It is only colored in this example for a better reading experience):

original:	new:	diff output:
1 This text is the same in	1 Some important information	0a1,4
2 both documents and should	2 is added to the top of the	> Some important information
3 not appear in the diff	3 document.	> is added to the top of the
4 file.	4	> document.
5	5 This text is the same in	>
6 This is some legacy text	6 both documents and should	6,9d9
7 that will be deleted in	7 not appear in the diff	< This is some legacy text
8 the newer version.	8 file.	< that will be deleted in
9	9	< the newer version.
10 Here is some more sample	10 Here is some more sample	<
11 text that contains some	11 text that contains some	12c12
12 spelling mistakes which	12 spelling mistakes which	< spelling mistakes which
13 shall be corrected in the	13 shall be corrected in the	---
14 latter version.	14 latter version.	> spelling mistakes which
	15	14a15,18
	16 One last paragraph is also	>
	17 added to the end of the	> One last paragraph is also
	18 document.	> added to the end of the
		> document.

Table 1: Sample diff output

The first column shows the original file, the second column the modified file and the third column shows the diff utility output when comparing the two files. Note that the diff utility doesn't provide colorful output, it is only shown here in a better readable way.

The numbers in diff output represent lines in the two files, where the respective change has occurred (be it addition, change or deletion). Numbers before the a/d/c are line numbers in the first file and after the a/d/c are line numbers of the second file. If there is a comma separated number, this indicates multiple lines between the two lines separated by comma (e.g. 6,9d9 means that lines 6 to 9 from the original file were deleted). The a/d/c symbol represents the type of operation that is in place (append, delete, change).

The symbol greater than means added lines in the new file, whereas smaller than indicates lines that were removed from the original file. Lines that are the same in both files, are usually not shown. For the change operation, both the original and new file contents are shown.

This is the basic output of the Unix-based diff utility. However, there are numerous other programs and IDE plugins showing file differences in different ways. The diff utility itself also features other output formats.

In 1981 as part of 2.8 BSD, the Berkeley distribution of Unix added the **Context format**. Here, any changed lines are shown along with a number of unchanged lines before and after the change - thus offering a context for the change. This way, the change operations are better understandable for humans and also offer a reference for the exact position of the change in case the line numbers no longer correspond. The number of unchanged lines can be chosen, with a default value of 3 lines.

The context format can be run with the -c parameter (i.e. diff -c original

new). In the output, the use of greater than and smaller than symbols is omitted and instead, a ! represents a change operation and a + represents an addition.

A **Unified format** was later developed, which is based on the context format, but produces smaller output by showing changed lines from both files directly adjacent to each other. This format can be requested with the -u command line option. The unified format has become the most popular and is requested by most patch programs. Also, most of the state-of-the-art software IDEs show code differences in a similar way as the unified format (basically the main difference being, not showing only 3 previous and following unchanged lines, but the whole file with changes indicated inside).

In the unified format, any added lines are marked with +, deleted lines with - and any change operation is split to its primitive suboperations of - and +.

Context format:	Unified format
** old.txt 2017-09-12 11:54:37..... --- new.txt 2017-09-12 11:56:52..... ***** ** 1,14 **** This text is the same in both documents and should not appear in the diff file.	--- old.txt 2017-09-12 11:54:37..... +++ new.txt 2017-09-12 11:56:52..... @@ -1,14 +1,18 @@ +Some important information +is added to the top of the +document. + This text is the same in both documents and should not appear in the diff file.
- This is some legacy text - that will be deleted in - the newer version. - Here is some more sample text that contains some ! spelling mystakes which shall be corrected in the latter version. --- 1,18 ---- + Some important information + is added to the top of the + document. + This text is the same in both documents and should not appear in the diff file.	-This is some legacy text -that will be deleted in -the newer version. - Here is some more sample text that contains some -spelling mystakes which +spelling mistakes which shall be corrected in the latter version. + +One last paragraph is also +added to the end of the +document.
Here is some more sample text that contains some ! spelling mistakes which shall be corrected in the latter version. + + One last paragraph is also + added to the end of the + document.	

Table 2: Context and Unified format of the sample diff output  
(some passages replaced by dots)

### 2.2.2 Merge Overview

Merging can be described as the next step to diff, where we want to merge changes done by multiple people to the same document in order to obtain a final document with all the changes included. This allows multiple people to work simultaneously on the same software fragments, textual documents or other forms of data and thus enables a new level of productivity. [41] have shown that software merging is an absolute necessity for large-scale software development. To date, merge is also a common feature of the major version control mechanisms [37, 39].

A big issue with merging is that there is not a single correct solution, only different approaches that are getting better over time (and still need improvement) [1]. The easiest solution for guaranteeing consistent version control is the *pessimistic version control* - here the file is locked by the currently editing worker until changes are saved and the file is unlocked again. However, this approach doesn't allow for collaboration of multiple workers simultaneously. Thus it is much more desirable to use the *optimistic version control* [1].

The goal of effective merging is to merge as much as possible in an automated way, without human intervention. When two people edit a different part of the same document or make the same change, this is no problem. However, when two people edit the same part of the document in different ways, a so-called *conflict* arises, and brings many questions as to how to resolve the conflict. The simplest way is to ask the human worker in case of a conflict, which change he wants to keep, or how he wants to resolve the conflict manually. This is especially the case with textual merging, which is the simplest method of merging that doesn't consider the syntax or semantics of software (or other types of documents). However, there are other approaches that make use of the syntax and semantics and allow much higher automation of conflict resolution.

Imagine an example sequence 1: *I see a cat* and two changed versions 2a: *I see a big cat* and 2b: *I see a black cat*. What would be the result of merging these two sequences?

There are different classifications of merge techniques [1].

**2.2.2.1 Two-way, Three-way merging** A two-way merging only looks at two files and merges them together, it does not take into account their common ancestor. Shall file 1 be the ancestor and files 2a and 2b the changed versions of file 1, so does a two-way merging ignore file 1 and only compares files 2a and 2b for differences. As opposed to that, a three-way merging also considers the ancestor file to get more contextual information and thus make better decisions about the changes. As a result, most of currently used merge tools utilize three-way merging [1]. An example of the two-way merging is the Unix-based diff utility, whereas the utility diff3 is an example of three-way merging.

To illustrate the difference between two-way and three-way merging, let us see a pseudo-code example:

With two-way merging, only files 2a and 2b are compared and thus, it is not decidable whether the line for(...) version 2a or 2b is to be used. Mostly, one would simply choose, which file should be merged into the other and that would define the preference. With three-way-merge on the other hand, we don't have this problem. Version 1 clearly shows us that the line for(...) in ancestor is the same as in version 2b, which means that it was only changed in version 2a and

File 1:	File 2a:	File 2b:
<pre> for (x=0;x&lt;10;x++) {     do_some_stuff;     output_some_stuff; } </pre>	<pre> for (x=0;x&lt;20;x++) {     do_some_stuff;     output_some_stuff; } </pre>	<pre> for (x=0;x&lt;10;x++) {     do_some_stuff;     do_some_more_stuff;     output_some_stuff; } </pre>

Table 3: Sample file versions

therefore, this change from 2a should be incorporated.

**2.2.2.2 Textual, Syntactic, Semantic or Structural merging** **Textual merging** regards documents (be it software artifacts, process models, textual documents, XML files or any other document) only as text files. In effect, they mostly split up a text file into a series of lines that can be compared and merged. This method is widely used in practice - examples include the RCS (Revision Control System) rcsmerge tool [42], Sun’s filemerge tool [43], the DOMAIN Software Engineering Environment (DSEE) [44] or the Concurrent Version System (CVS) [45].

Textual merging is advantageous in its efficiency (generates a small change-set), superb scalability (it offers high speed, only rising in linear manner) and also flexibility (can be used to practically anything that can be split into lines or linear entities). Also, it was shown by [41] that in normal conditions, about 90 percent of the changed lines can be automatically merged. The problem with textual merging is the remaining 10 percent, which usually need to be merged manually and can’t be automated. Hand in hand with this goes the disadvantage of textual merging, which is the lack of using syntactical or semantic information to assist the conflict resolution.

**Syntactic merging** also considers the syntactical aspect of the artifacts and therefore brings better results [1]. It is usually bound to a specific programming (or modeling) language, or some other form of syntax of the artifacts. Thus, it is not as flexible as pure textual merging and some basic syntactical rules are needed for the concrete case.

With the syntactical information in hand, syntactic merging can avoid unnecessary merge conflicts arising from e.g. code comments modified simultaneously by multiple developers or from adjusting line breaks and tabs for improved code readability. It can handle such cases and only issue a conflict if there’s a syntactical error. Consider the following example with a code fragment[1]: *if (n mod 2)=0 then m:=n/2*.

If one developer changes the fragment to *if (n mod 2)=0 then m:=n/2 else m:=(n-1)/2* and another developer changes the entire line to *m:=n div 2* (both results are equal), then a textual merge would produce an incorrect mixture of both changes, whereas a syntactic merge will handle the situation right by issuing a syntax error.

Syntactic merging methods can further be classified into those that are tree-based and graph-based.

**Semantic merging** extends syntactic merging by also considering semantic aspects of a program that may occur when multiple developers code simulta-

neously. There are many cases when two concurrent changes to a code may change e.g. the variable name, so that it becomes incompatible and the code would not work in run time. From the syntactical perspective, however, it is absolutely fine, so a syntactic merge tool would not detect an error. This is where semantic aspects are taken into consideration to detect such errors.

Tree-based merge approaches cannot detect such conflicts, but graph-based approaches can [1]. There are so-called *static* semantic conflicts (such as a different variable name in the modified pieces of code that can be detected statically, i.e. before executing the code) and *behavioral conflicts* (such that are very hard to predict, since they usually only arise during run time). Most methods for detecting behavioral conflicts utilize denotational semantics, program dependence graphs, program slicing and other mathematical formalisms [1].

**Structural merging** conflicts occur when the semantics of a program remains unchanged, but the underlying structure changes. This is often the case in object-oriented programming (OOP), where classes can be reorganized, generalized or defined in a more refined way. Unfortunately, most of the existing merge tools cannot detect such conflicts [1].

**2.2.2.3 State-based, Change-based or Operation-based merging** Another significant classification of merge tools can be made into state-based and change-based merging. **State-based** merging only considers the static information in the original file and its revisions. Two-way merges are thus always considered state-based merging.

**Change-based** merging, on the other hand, also uses information about the exact changes performed between the respective revisions of the file. **Operation-based** merging extends change-based merging by standardizing changing as explicit operations. These are also referred to as transformations and can be split into primitive transformations [1, 46, 47].

Operation-based merging can improve the detection of conflicts and help with their resolution [46, 47]. It is not always necessary to compare whole revisions to handle conflicts, but merely the transformations performed in the evolution of the document. These changelogs are referred to as deltas, with the distinction being made between symmetric and directed deltas. Symmetric deltas show the difference between 2 versions of a file, while directed deltas show the exact sequence of operation to transform one version into another. Symmetrical deltas are used in the context of two-way merging, whereas directed deltas are used for three-way merging [1].

Operation-based merging is general by enabling the detection of syntactic, structural and also some kinds of semantic conflicts. Also, with operation-based merging, it is easier to implement undo/redo mechanisms [1].

## 2.3 Process Evolution

Process-aware information systems (PAIS) are information systems that are organized in a process-oriented way, meaning they align the focus strongly on the process that needs to be executed. As opposed to data- or function-centered IS, they show a strict separation of process logic and execution code [48, 49]. Through the separation of process logic and execution code, they allow for a separation of concerns, a well established principle in information technology that increases the maintainability and reduces costs of change [48]. In other



words, it is possible to edit the process model without touching the execution code and vice versa, it is possible to change the methods and interfaces that are executed for each activity without changing the flow of the process model.

One of the most important success factors for PAIS has been shown to be the ability to deal with process change [50, 51, 9, 52, 53]. This is also due to the fact that the ability to adapt to changes (e.g. changes in laws or customers' behavior) quickly is essential to the economic success of businesses in today's world [54].

Besides build-time flexibility (i.e. modeling the process flow of a business process in advance, before it gets executed), run-time flexibility also needs to be considered (i.e. adapting the process model during its execution, when additional information has been discovered) [51]. Being unable to change process models once the execution starts can pose a big threat to flexibility and adaptability and often freezes process models in the inaccurate state. Therefore, PAIS must also be flexible at run-time and allow users to adapt their process models in response to environmental changes [48].

To address the discussed requirements, PAIS have evolved to enable changes at the process type level and process instance level [55, 56, 57, 52, 58]. Changes at the process instance level affect only the single process instance that is being executed, which mostly involves exceptional situations that may occur. E.g. the process model states that a patient shall be investigated by MRT (Magnetic resonance tomography), X-ray and sonography, however it turns out that one patient has a pacemaker, which doesn't allow him to undergo MRT [59]. Changes at the process type level, on the other hand, allow the process designers to change the underlying process model, which will change all future instances. E.g. by the introduction of GDPR regulations in the European Union in 2018, the business process model of investigating a patient may now need to be enhanced by adding an activity of asking the patient for permission of dealing with his personal data at the beginning of the process model. This change shall not only be applied to a single process instance, but to the entire process model, so that all future instances of this model implement the change.

[49] have suggested a set of change patterns that help compare existing PAIS in regard of their ability to adapt to process change (based on their previous work in [53]). The suggested change patterns are scenarios that may occur in business process models and that have been identified from a large set of input data from the healthcare and automotive domain [49].

A total of 14 adaptation patterns have been suggested:

#### **Adding / Deleting Fragments**

1. Insert Process Fragment
2. Delete Process Fragment

#### **Moving / Replacing Fragments**

3. Move Process Fragment
4. Replace Process Fragment
5. Swap Process Fragment

#### **Adding / Removing Levels**

6. Extract Sub Process

7. Inline Sub Process

### Adapting Ctrl Dependencies

8. Embed Process Fragment in Loop

9. Parallelize Activities

10. Embed Process Fragment in Conditional Branch

11. Add Control Dependency

12. Remove Control Dependency

### Change Transition Conditions

13. Update Condition

14. Copy Process Fragment

As can be seen, the 14 patterns are divided into 5 groups. Some of the patterns can further be classified into deeper granularity, such as Insert Process Fragment. Let us see a graph of this pattern:

1. X is inserted between two directly succeeding activities (*serial insert*)
2. X is inserted between two activity sets (*insert between node sets*)
  - a) without additional condition (*parallel insert*)
  - b) with additional condition (*conditional insert*)

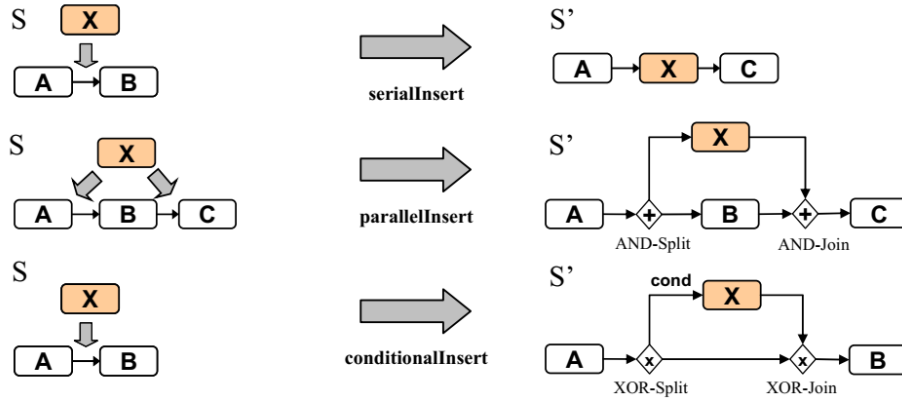


Figure 12: Pattern 1 - Insert Process Fragment [49]

In the above figure, we can see the three heterogeneous cases of an Insert operation. An activity X is inserted 1) between two activities 2) parallel to an activity and 3) between two activities with a condition.

Changes performed to process models can also be logged to a change log, in order to keep track of all the changes (and thus versions) of the respective process model. It is further possible to work with such change logs and to perform data mining on them [60], if desired, to compare two versions of the same process model based on the evaluation of the change logs. An entry of such a changelog (i.e. a change operation) is characterized by [60]:

- type of change (e.g. Insert, Delete, Move, Replace)
- subject (which has been changed)
- syntactical context of the change (pre and postset, i.e. where the change occurs inside the model)

### 3 Conceptual Design

In order to propose a highly automated semantic BPM merging algorithm, we need to prepare the available test instances gathered from CPEE, so that they can be used for merging. We also need to analyze the semantic characteristics of BPM and CPEE in specific, so that we can develop a highly optimized merge algorithm.

First, in chapter 3.1, we will develop a script that regroups the instances (so that they are ordered by students) and extract only important information for our case. Next, in chapter 3.2, we will develop another script that goes through all instances of a student and determines, which of the instances introduce any change (in structure, endpoint or parameters) - we call such instances significant.

With this information available, in chapter 3.3 we can propose a diff algorithm that will perform an exact diff of two instances - making a complete diff of structure, endpoints and parameters. This diff algorithm will be used as a basis by the proposed merge algorithm.

Finally, in chapter 3.4, we will propose a semantic BPM merge algorithm whose main objective is to resolve as many conflicts as possible in an automated manner.

#### 3.1 Preparing Sample Instances

In this thesis, we will be working with a large number of business process instances in order to compare them, show how they evolved over time (in other words, make a diff of them with specific syntactical and semantic aspects) and in the end, merge various changes together.

As a basis for sample data to work with, we will use real-world business process instances from the Cloud Process Execution Engine (CPEE). CPEE is a modular, service oriented workflow execution engine developed by the Workflow Systems and Technology research group at the University of Vienna. It is modular in that it doesn't have an internal API (dependent on a concrete framework), but instead allows building custom solutions on top of a simple REST API - supporting multiple execution languages (e.g. BPEL, YAWL, BPMN) and interaction protocols (e.g. SOAP, REST, XMPP). It is a REST service with very little base code, strongly optimized for speed, memory consumption, multi-core architectures and can be easily scaled with load-balancing. In 10 years, the engine has been downloaded about 80000 times, used in 15 national and international research projects and used to teach thousands of students in various courses about workflow management and service orchestration. Due to these huge numbers of real data, we can easily extract thousands of process instances to work with in our application and perform extensive testing.

In its basic form, CPEE is a REST service, depicting process models as Refined Process Structure Trees [17]. A process model instance describes a state of a process model and each saved change to the model (such as a change of structure, parameters of activities, parameters of the model itself) as well as execution of the model creates a new instance. In other words, every time the user saves changes to the model, or executes the model, a new instance is created.

Let us start by running a demo in CPEE:

Figure 13: CPEE Demo Step 1: Selecting an instance

In the first step of the demo, you can either choose an already existing instance (by typing its url. i.e. id) or create a new instance. By clicking *monitor instance* you open the instance and can modify it. We have created a sample instance and filled it with the same sample business process (of building a house) as was demonstrated for the various modeling languages in subchapter 2.1.

The general CPEE workspace layout is divided into three parts. In the section on top, you can create new instances, perform actions on instance level (such as load testsets/models or export the instance and set model type) and execute the instance. In the mid-section, you can see and change data elements, endpoints and attributes associated with the instance. And in the bottom section, you can see and modify the actual content of the process model instance, i.e. all the activities, decisions, loops and other elements, along with its attributes.

As has been said before, CPEE logs a huge amount of real data from students enrolled in courses. For the primary development of this project, we will be using about 6000 sample instances gathered between December of 2016 and January 2017 as part of the Workflow Technologies <sup>2</sup> course. In latter stages of the project, we might use a substantially higher number of instances for testing purposes, but as for now, about 6000 instances are sufficient for the early stage of development. The sample instances are contained in 6 folders with about 1000 instances per folder.

As we open one of the folders and further explore the structure, we find that each instance folder contains a file named *properties.xml* and another subfolder named *notifications*:

Only relevant for us is the file *properties.xml* which contains all the important information about a process instance. In the following subchapters, we will explore all the information stored within the instances and discuss which information is relevant for the purpose of the thesis. In effect, we will regroup instances by students who created them to obtain a list of all instances for each student with only the relevant information included. Later, by having the data well prepared, we will continue by traversing through all instances of a student to find which instances introduce any significant changes. And in the next step, we will determine the exact changes made to the instances in each step by using various diff algorithms.

<sup>2</sup><https://ufind.univie.ac.at/en/course.html?lv=052513&semester=2016W> (last access: 7.11.2017)

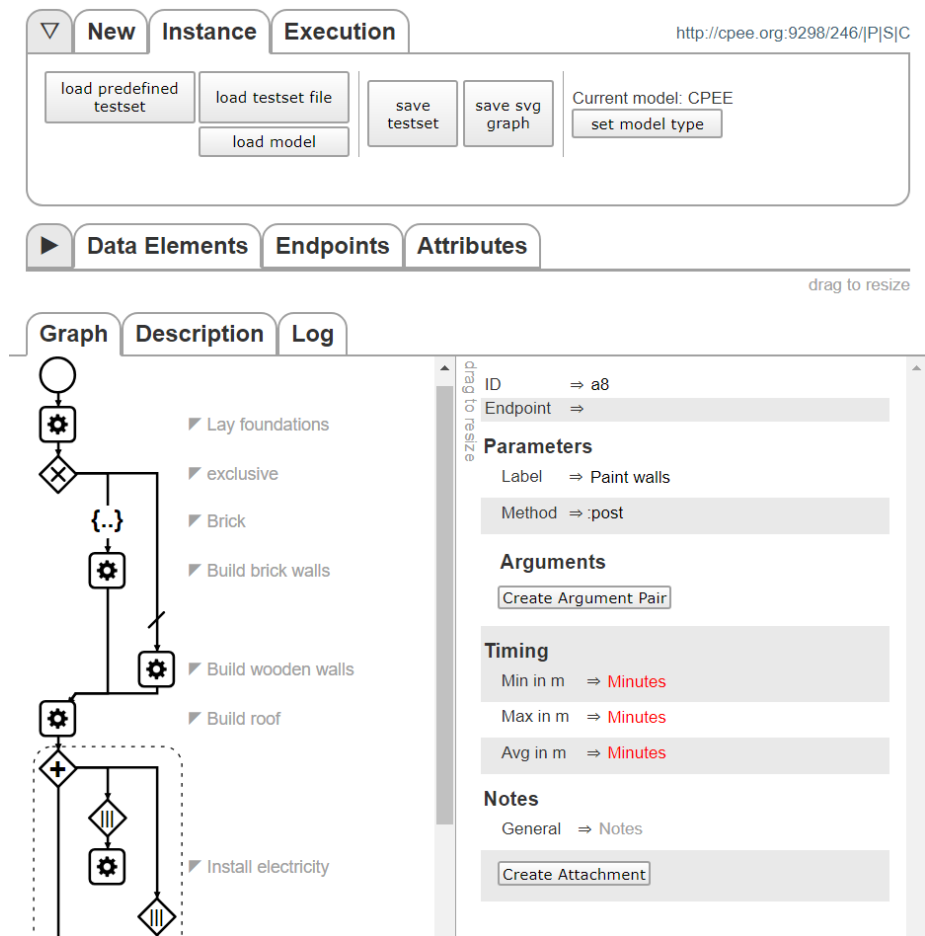


Figure 14: CPEE Demo Step 2: Creating a sample instance

Name	Date modified	Type	Size
instances	21.01.2017 0:32	File folder	
instances.2016_12_08.17_20	20.01.2017 11:05	File folder	
instances.2016_12_11.03_34	20.01.2017 23:21	File folder	
instances.2016_12_15.15_36	21.01.2017 0:30	File folder	
instances.2016_12_22.16_10	20.01.2017 11:05	File folder	
instances.2017_01_09.22_54	20.01.2017 11:05	File folder	

Figure 15: CPEE sample instances

### 3.1.1 Information Contained in Instances

There is a lot of information stored for each instance in the *properties.xml* file, much of which is not relevant to us. All the information is evidently stored in xml format. Let us have a look at a sample *properties.xml* file from the delivered CPEE sample. In order to keep an overview and limit the length of the sample,

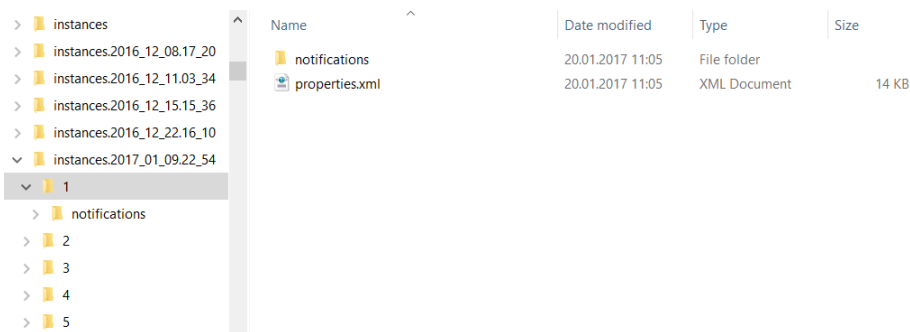


Figure 16: CPEE sample instances (inside a folder)

certain passages were cut out and replaced with dots. This is instance 13 from the first folder of instances and was created by student *a1246269*:

```
<properties xmlns="http://riddl.org/ns/common-patterns/properties
/1.0">
  <attributes>
    <uuid>81b78262-37da-4d0a-9f08-8aab69225d82</uuid>
    <info>a1246269</info>
    <modeltype>CPEE</modeltype>
    <theme>default</theme>
  </attributes>
  <state>finished</state>
  <handlerwrapper>DefaultHandlerWrapper</handlerwrapper>
  <positions/>
  <dataelements>
    <loginStatus>true</loginStatus>
    <allProductsSelected>true</allProductsSelected>
    .....
  </dataelements>
  <endpoints>
    <timeout>http://gruppe.wst.univie.ac.at/~mangler/services/
    timeout.php</timeout>
    .....
  </endpoints>
  <dsl>call :a2, :getLoginForm, parameters: { :label => "system_
    shows_login_form_to_customer", :method => :post,
    :arguments => [(:name => :role, :value => "customer"
    ), (:name => :organisation, :value => "privateCustomer"
    )] }, finalize: <&lt;&&<-END
    data.loginForm=result['loginFormUrl'];
    data.loginFormEntries=result['entries'];
    data.userRole="customer";
    data.userOrganisation="privateCustomer";
  END
  .....
</dsl>
<dslx>
  <description xmlns="http://cpee.org/ns/description/1.0">
    <call id="a2" endpoint="getLoginForm">
      <parameters>
        <label>system shows login form to customer</label>
        <method>:post</method>
        <arguments>
```

```

        <role>"customer"</role>
        <organisation>"privateCustomer"</organisation>
    </arguments>
    </parameters>
    <finalize output="result">data.loginForm=result [ '
        loginFormUrl ' ];
data.loginFormEntries=result [ 'entries ' ];
data.userRole="customer";
data.userOrganisation="privateCustomer";</finalize>
    <update output="result" />
</call>
    .....
</description>
</dslx>
<status>
    <id>0</id>
    <message>undefined</message>
</status>
<description>
    <description xmlns="http://cpee.org/ns/description/1.0">
        <call id="a2" endpoint="getLoginForm">
            .....
        </call>
    </description>
</description>
<transformation>
    <description type="copy" />
    <dataelements type="none" />
    <endpoints type="none" />
</transformation>
</properties>

```

Figure 17: Sample XML instance file from CPEE

The base element is *properties* with an XML namespace defined.

*Attributes* define general attributes of the instance. Aside from the CPEE-specific *UUID*, they also contain the important *info* attribute, storing the student's number. This will help us group instances by students in further steps.

*State* defines the state of execution for each instance. In this case it's finished, meaning that the instance was fully executed and came to an end state.

There are a few, for our use case irrelevant attributes, such as *handlerWrapper* and *positions*. Also *dataelements*, containing some data used in procedures during the execution of the instance, is another element without importance to us.

*Endpoints* define urls to be accessed by procedure calls during the execution of the instance. The meaning of a workflow engine is to automatically execute processes and in the various steps (i.e. activities) of the process to execute certain work fragments. In CPEE, you can call web services in activities by setting the endpoints and calling them in the respective time point.

The process itself is stored at three locations:

- *dsl* defines the structure of the process model in a textual representation.
- *dslx* defines the structure of the process model in XML format.
- *description* also stores the structure in XML format.



For our processing of the process models, we will be using the `dslx` element. Within the process model, there are a few noteworthy elements:

- `call` defines an activity
- `choose` defines a decision
- `alternative` represents a path in the decision
- `loop` is a repetitive sequence (i.e. `loop`) until a terminating condition is fulfilled
- `parallel` defines parallel paths

These elements have a number of different attributes and parameters, many of which will be of interest for our work. Since calls are the most common elements, I will now focus on them. Each call has an `id` and `endpoints`, which defines the endpoints described earlier. The parameters include a `label` (i.e. a call's name), `method` (HTTP method), `arguments` (to be sent via HTTP request to the Web service) and finally an execution sequence to be performed after the HTTP request called `finalize`.

### 3.1.2 Relevant Information

Of all the information contained in instance files, only a fraction is relevant for our further work. For this purpose, along with regrouping the instance files (instead of having all instances along each other in purely chronological order to having them grouped by students and ordered chronologically), we will also be trimming all of the superfluous information and do some minor tweaking of the instance files. This way, they will be better suited for further processing.

In order to compare instances with each other and make semantic diffs of them in the next steps, we will examine the structure of the process model (i.e. the order of activities, decisions, loops and other elements), the endpoints assigned to activities, as well as parameters of the elements. For this purpose, we will first reduce the instance files only to elements that are relevant for us.

Relevant for us are mainly the endpoints and `dslx` elements, as well as parameters of the `dslx` elements (such as activities, decisions or loops). We will also use the `attributes/info` element containing the student's number, but only to correctly assign the instances and won't store it in the processed instance file. We will also keep the `dsl` element in our instance file, however only for the case that a need of using the textual DSL rises later.

Consequently, we will delete all of the following elements, including all of the children nodes:

- `attributes`
- `state`
- `handlerwrapper`
- `status`
- `description`

- transformation

We will also copy the content of endpoints to the respective spots in dslx where they are addressed, and afterward delete them.

Thus, the processed instance file shall look this way:

```
<properties xmlns="http://riddl.org/ns/common-patterns/properties"
  /1.0">
  <dsl>
    .....
  </dsl>
<dslx>
  .....
</dslx>
</properties>
```

Figure 18: Processed Instance File (Excerpt)

### 3.1.3 Regrouping Instances by Students

After having analyzed the content of instance files in previous subchapter and having proposed a resulting design of processed instance files, we can now proceed to carry out the transformations needed. We will write a small one-purpose script to regroup all instance files by students and remove all irrelevant information.

The script will take following actions:

1. Traverse through all folders and instance files in them; and for each instance file:
2. Check if instance is valid and usable for our purpose, i.e. has valid XML structure and contains a student's number
3. If this student doesn't have a folder yet, create it
4. Remove irrelevant elements:
  - attributes
  - state
  - handlerwrapper
  - status
  - description
  - transformation
5. Copy endpoint values to respective positions in dslx
6. Remove endpoints element
7. Save resulting XML document into the student's folder (with automatic incremental file name, i.e. starting with 1,2,3 and so on)

```

Grouping instances from folder: instances/
785 instances found, 543 valid. 24 student dirs created, 543 files
  copied
Grouping instances from folder: instances.2016_12_08.17_20/
1000 instances found, 414 valid. 15 student dirs created, 414 files
  copied
Grouping instances from folder: instances.2016_12_11.03_34/
1001 instances found, 734 valid. 1 student dirs created, 734 files
  copied
Grouping instances from folder: instances.2016_12_15.15_36/
1006 instances found, 325 valid. 0 student dirs created, 325 files
  copied
Grouping instances from folder: instances.2016_12_22.16_10/
999 instances found, 339 valid. 0 student dirs created, 339 files
  copied
Grouping instances from folder: instances.2017_01_09.22_54/
1001 instances found, 692 valid. 1 student dirs created, 692 files
  copied

Total:
15.9708 s ... 5792 instances found, 3047 valid), 41 student dirs
  created, 3047 files copied

```

Figure 19: Executing the group instances script

During the whole process, we want to have some statistics for a better overview. The statistics will be shown for every source folder (of approximately 1000 instances as imported from CPEE) and in total. Let us run our script:

As can be seen at the terminal output, the grouping script shows statistics for every folder separately and a total sum. In total, running the whole script took 15,9 seconds and found 5792 instances, out of which 3047 are valid. Valid here means not only with a valid XML structure and without any flaws, but also containing a student's number in a valid format, so that we can assign every single instance to a student. A total of 41 students folders were created (meaning that there are process instances from 41 different students in the data sample). When looking at the more detailed statistics of each folder, we can see an interesting detail, that the first folder created 24 student folders and the further we go, the less new student folders are created. This is very logical and to be expected, since we always add only new student folders. Also, it is noteworthy that in some of the folders, only about a third of the process instances were actually valid (and suitable for our purpose).

Let us now have a look at the file system to see the changes performed:

In the above figure, we can see a fraction of the student folders created (there are 42 in total, as was said before). When we open one of them, we see following files:

We only see the top 10 instances from a student here, which is just a little fraction. Just to give an idea, e.g. student a0006750 has 156 instances, a0106650 has 18 instances and a1246269 has 178 instances. Now that we have successfully regrouped instances by students, we have finished preparing the sample instances. In the next steps, we can pick a sample student and start evaluating all of their instances - first by determining significant instances (i.e. instances where any noteworthy change has occurred) and later by determining the exact











Name	Date modified	Type	Size
 a0006750	22.10.2017 14:16	File folder	
 a0106650	22.10.2017 14:16	File folder	
 a0552191	22.10.2017 14:16	File folder	
 a0648739	22.10.2017 14:16	File folder	
 a0651676	22.10.2017 14:16	File folder	
 a0853550	22.10.2017 14:16	File folder	
 a0867923	22.10.2017 14:16	File folder	
 a1004500	22.10.2017 14:16	File folder	
 a1026133	22.10.2017 14:16	File folder	
 a1125776	22.10.2017 14:16	File folder	

Figure 20: Folder structure created by the *group instances* script











Name	Date modified	Type	Size
 1.xml	22.10.2017 14:16	XML File	19 KB
 2.xml	22.10.2017 14:16	XML File	19 KB
 3.xml	22.10.2017 14:16	XML File	19 KB
 4.xml	22.10.2017 14:16	XML File	17 KB
 5.xml	22.10.2017 14:16	XML File	19 KB
 6.xml	22.10.2017 14:16	XML File	17 KB
 7.xml	22.10.2017 14:16	XML File	17 KB
 8.xml	22.10.2017 14:16	XML File	19 KB
 9.xml	22.10.2017 14:16	XML File	19 KB
 10.xml	22.10.2017 14:16	XML File	19 KB

Figure 21: Instance files in a student's folder

changes performed between the versions.

### 3.2 Determining Significant Instances

After having prepared the process model instances and grouped them by students in chronological order, we now want to see which of the instances differ, compared to the previous one. We will in effect call instances that make a significant difference significant instances.

In common document editing software (be it office documents all the way to software development IDEs), one document is usually a self-enclosed entity that gets saved over and over, so that you can at any time identify which document it is (identified usually by name); and possibly extended by versioning control, allowing you to see how this one document evolved. In CPEE, however, there is no such thing as a document with its different versions, there are only instances. Thus, one needs to be cautious, not to misinterpret instances as heterogeneous documents. Instances may or may not evolve from each other; an instance can be created from scratch, from its direct predecessor (with id by 1 lower) or from any previous predecessor. Numerous instances are often created while editing the same document and during the sole process of creating a desired version of

a process model. Also, a number of instances get created during the execution. For these reasons, it is important for us to differentiate between instances that do make a significant difference to the previous ones and between instances that are only some kind of byproduct.

We define following significant changes to instances (meaning that the instances will be marked significant):

- **Changed structure.** Whenever there is a change in the structure of a process model, it is an important change for us that deserves further investigating. A change of structure means that some elements, such as activities, loops, decisions or parallels have been added, removed, shifted or renamed.

We will be using the name of elements as an identifier in order to compare them, even though there is an id provided for each activity by CPEE. When changing structure of models, people often take an old activity and make a (semantically) entirely new activity out of it, ruining the sense of its id. By sticking to names, we can differentiate it more properly. The downside is, naturally, that every change to an activity name (e.g. correcting spelling mistakes) will appear as a structure change.

- **Changed parameters of elements.** Another significant change in instances for us is a change to the parameters of the various elements. These parameters define the properties of elements and are also important for the identity of an instance. Attributes may include some of the following: label of the element, method (HTTP request method), arguments (used for the HTTP interaction).
- **Changed endpoints.** Endpoints define the URLs to which scripts should be making HTTP requests in the defined activities. In other words, they define the Web service locations which are to be used in the course of the process model. Changing them also counts as a significant change to the model.

In order to detect any of the three significant changes described above, we will write another one-purpose script. In short, the script will go through all of a student's instances, check for any differences in structure, parameters and endpoints and produce both a textual (suitable for further processing, thus in a common data exchange format) and graphical output. The textual output will be used by our further scripts as a data source to find all of the exact changes between instances. The graphical output is important for people, since it makes the situation a lot easier to understand. For the graphical output, we will be using GraphViz <sup>3</sup>, an open source multi-platform graph visualization software.

The script will perform following steps:

1. Traverse through all of a student's instances
2. Check if a structural change has occurred
3. Check if a change in parameters has occurred
4. Check if a change in endpoints has occurred

---

<sup>3</sup><http://www.graphviz.org/> (last access: 7.11.2017)

5. If a structural change has occurred, traverse all previous significantly structured instances and check, if an ancestor with the same structure as the current instance can be found (meaning that the current instance most probably derives from that ancestor)
6. Export the results into a json file with all significant instances (we chose json, however XML or any other format is possible, too)
7. Export a graph showing all the instances, highlighting any of the three kinds of significant changes made and highlighting structural ancestors of instances with changed structure.

Let us run this script for the sample student *a1246269* with detailed output (and with long passages replaced by dots):

```

Changed endpoints 4
Changed parameters 4
.....
Changed structure 16
15 => 16
Changed structure 17
16 => 17
.....
Changed structure 31
Found ancestor: 28 (of 31)
Parameters & endpoints match
28 => 31
.....
Changed structure 67
Found ancestor: 62 (of 67)
Found ancestor: 16 (of 67)
62 => 67
.....
Total:
2.7136 s ...

```

Figure 22: Executing the *find\_significant\_instances* script

In Figure 22, we can see that in instance 4, both endpoints and parameters have changed, however the structure remained the same. In instance 16, the structure has changed and no ancestor was found with the same structure, so the graph will display solely an arrow from 15 to 16 (with a special kind of arrow for changed structure). The same is the case for instance 17.

We can, however, see that in instance 67 the structure has changed and an ancestor with the same structure was found - instance 62. Since before, an ancestor to 62 was found being 16, both of these are written out here as also being possible ancestors of 67. In this case, the latest of the ancestors (62) will be shown in the graph with a special arrow showing the relation.

Now let us see a fraction of the json file of all significant instances that will be used later for further processing:

```

{
  "1": {
    "change_type": "none",

```

```

    "activities": [
      "system shows login form to customer",
      .....
    ],
    "endpoints": [
      "http://wwwlab.cs.univie.ac.at/~a1246269/workflow/ws/getLoginForm.php",
      .....
    ],
    "parameters": [
      {
        "role": "\"customer\"",
        "organisation": "\"privateCustomer\""
      },
      .....
    ]
  },
  "4": {
    "change_type": [
      "endpoints",
      "parameters"
    ],
    .....
  },
  .....
  "16": {
    "change_type": "structure",
    .....
  }
  .....
}

```

Figure 23: All significant instances json file

Now let us see the resulting graph of the evolution of instances (due to its size, only a fragment will be shown):

The thin grey arrow represents no significant change, so e.g. nothing important has changed between instance 14 and 15. Any other uninterrupted line means a significant change of some kind, whereas dashed lines mean equality (to a previous ancestor). The thick black arrow means a structure change (when no previous ancestor is found with the same structure), e.g. from instance 15 and 16. A red line means parameter change and a green line means endpoints change.

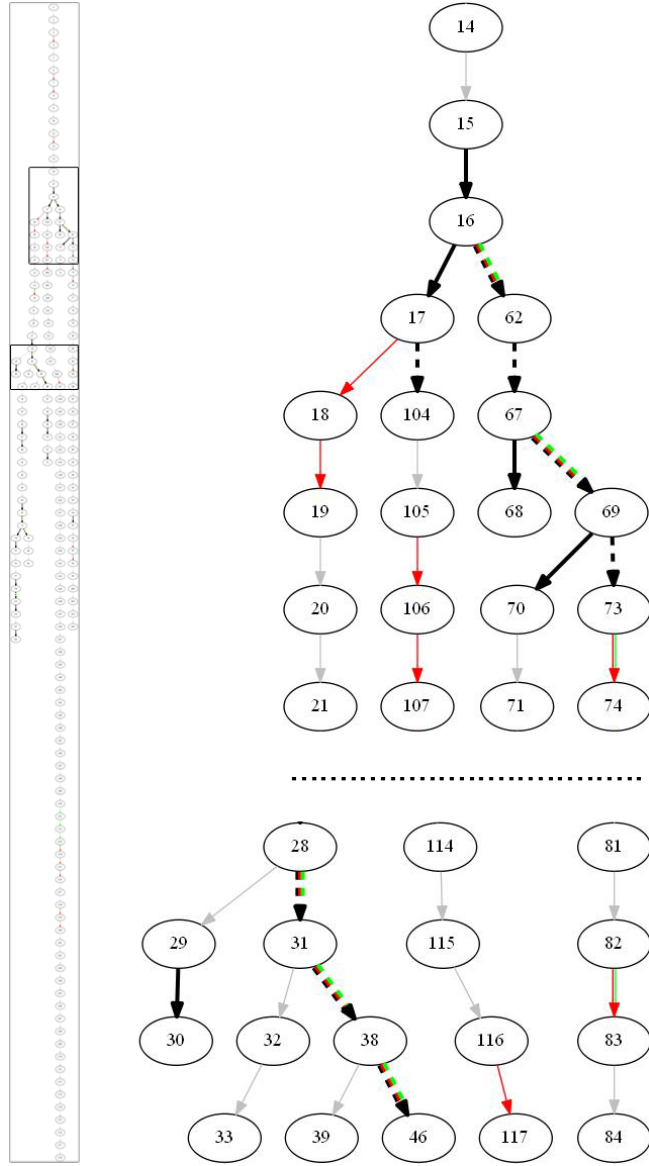


Figure 24: Graph of significant instances and changes (showing fractions and a scale of the whole graph)

If an ancestor is found (e.g. instance 16 being ancestor of 62), we show a dotted thick black line. This means that the structure of 62 has changed (as compared to 61) and it was discovered that 16 equals to that of 62. If also the parameters match, we add a dashed red line and if endpoints match, we add a dashed green line. You can e.g. see that from 16 to 62, structure, parameters and endpoints match, whereas from 62 to 67, only the structure matches.



### 3.3 Proposed Diff Algorithm

After we have determined all the significant instances of a student, we can continue by determining the exact diffs between each of the significant instances. Our goal is to obtain an exact diff of all the structure between each of the significant instances, as well as obtain all changes made to the endpoints and parameters of the respective instances.

In the first step, we will naturally perform a simple diff between two sample instances. We will be examining the quality and performance difference between various algorithms, as we will implement both a very basic diff algorithm (producing correct, but non-optimal results), as well as an LCS-based algorithm. By implementing both kinds of algorithm, we will be able to show the big difference in a very nice way and to demonstrate their efficiency on a large amount of instances. Additionally, we will also incorporate a third diff implementation, an entirely external solution offering out-of-the-box solution for diff with a number of different output options (such as the context format, unified format or side-by-side table view of the original and updated document).

After having implemented the heterogeneous diff algorithms on a small scale (i.e. for a single pair of instances), we will scale it up, so that we can automatically run diff for all the instances of a selected student and produce useful output artifacts, such as an overview graph depicting all differences, as well as textual (json) exports of the diffs, usable for further processing.

Since we have already processed the instances in previous steps and produced nice json exports, we will now be using them as the data basis for our computations. The format of the json file can be seen in Figure 23 and was produced by our *find\_significant\_instances* script. This file only lists significant instances (i.e. instances where changes to structure, endpoints or parameters have occurred) and includes their structure (such as activities, loops or decisions), their endpoints as they belong to the respective activities and finally also all the parameters connected to the respective activities. More details about how each of these entities will be processed and made diff of, can be read in the following subchapters.

#### 3.3.1 Diff of Structure

First of all, we want to make a diff of the structure of process models. By structure we mean the exact composition of activities, decisions, parallels, loops and other elements in their exact order and relations. A big advantage of CPEE (and other tree-based process modeling methods) is that models are recorded as a tree where each split-making element (such as decision or loop, which divide the flow into multiple lines) have their exact children. As opposed to BPMN or other methods, the RPST guarantees to us that we don't need to worry about possible design flaws that can easily occur in BPMN or other languages, such as misplaced closing elements. This way, our input format avoids the possibility of such errors and we can focus solely on our work.

As has been described in Chapter 3.1.1, CPEE encodes process models in an XML-based format. Let us have a quick look at a sample file of a student taken from CPEE, in the XML format (in a very simplified way, just to show the structure):

```

<call id="">.....</call>
<call id="">.....</call>
<call id="">.....</call>
<choose mode="exclusive">
  <alternative condition="data.loginStatus== true">
    <call id="">.....</call>
    <loop mode="post_test" condition="data.allProductsSelected == &
      quot;false&quot;">
      <call id="">.....</call>
      <call id="">.....</call>
      .....
    </loop>
    <call id="">.....</call>
    <call id="">.....</call>
    <parallel wait="-1">
      <parallel_branch pass="" local="">
        <call id="">.....</call>
        .....
        <choose mode="exclusive">
          .....
          </otherwise>
        </choose>
      </parallel_branch>
      <parallel_branch pass="" local="">
        .....
      </parallel_branch>
    </parallel>
    <call id="">.....</call>
  </alternative>
  <otherwise>
    <call id="">.....</call>
  </otherwise>
</choose>

```

Figure 25: Structure of a Sample XML Instance File From CPEE

With our *find\_significant\_instances* script, we have extracted the process structure into a simple array of elements, which allows us to very easily compare the structure between model instances. A single-level array can be compared the same way line-based documents are, since each element of the array equals a line. One possible drawback of this approach is that some nesting information could be lost - some elements are on the highest level, while some are nested underneath an element, such as loop. If you e.g. have a loop followed by three activities in a single-level array, you cannot say, which of the three activities are inside the loop. It may be only the first one, but it may very well be all three of them. However, this issue can be easily resolved. We can keep the single-level array and simply modify the names of elements in such ways, that they reflect their nesting levels. We can e.g. change an activity named *Build brick walls* to *\_Build brick walls*, clearly showing that it's indented by one level. Since this modification can be added very easily (and reinterpreted back) any time later, we will only leave it as a possibility for now and not focus on doing it.

So our json export of the same sample process would look like this (without the indentions):

```

"activities": [
  "system shows login form to customer",
  "customer sends login data",

```

```

    "system checks credentials and responses",
    "choose",
    "system creates session",
    "loop",
    "system shows product list(s)",
    "customer selects product",
    "system adds product to the shopping cart",
    "customer checks if all needed products are saved",
    "system prepares for payment and shipment",
    "system shows payment form to customer",
    "customer enters payment data",
    "system shows adress form to customer",
    "customer enters shipping adress and decide if alternativ billing
      adress",
    "choose",
    "system shows adress form to customer",
    "customer enters independent billing adress",
    "system sets shipping adress as billing adress",
    "banks conforms that products were payed",
    "system finishes order",
    "system ends interaction"
  ],

```

Figure 26: Simplified JSON Version of the Sample Instance File (Figure 25)

We will now be performing diff with these data with two different algorithms - a very basic one which doesn't produce optimal results and with an LCS-based algorithm. This way, we want to clearly show the difference in efficiency between these two algorithms.

**3.3.1.1 Basic diff algorithm** First, we will implement a fairly basic diff algorithm to compare two files (in our case the array of elements in a process model structure). This algorithm is not LCS-based, meaning it does not necessarily find the longest common subsequence and in effect, does not necessarily find the shortest possible diff between two files. It does find a correct difference, but not always the optimal one.

The principle of the first algorithm is quite simple. Basically, we take the first row from file A and try to find the nearest match in file B (nearest meaning from the same line forward). Then, we do the same for file B - we take the first line and try to find the closest match in file A. Then, we compare these two matches and we take the one with smaller difference, i.e. where the two lines are closer to each other. We then mark everything in between as changed, move the pointers to the new lines and continue to do so until the end of file.

Here is a step-by-step summary of the simple algorithm:

1. Take line 1 of file A and find the first match in B (only allowed to go forward)
2. Take line 1 of file B and find the first match in A (only allowed to go forward)
3. Select the match with smaller difference
4. Mark everything in between as changed
5. Mark the match as equal

6. Move pointers to the new positions

7. Repeat 1-6 until end of file

While this algorithm produces correct results (meaning that its diff can be applied to file a in order to obtain file b), the problem is that it doesn't find the LCS and thus doesn't produce the shortest possible diff (though in most normal cases it works well). Let us run the basic algorithm on the example sequences (a,x,x,b,x,x) and (b,x,x,a,x,x,b,x,x) as depicted in Figure 9:

```
- (0) a
- (0) x
- (0) x
+ (6) a
+ (6) x
+ (6) x
+ (6) b
+ (6) x
+ (6) x
```

Figure 27: Results of the simple diff algorithm

The optimal results (as would be delivered by an LCS-based algorithm would, however, look in the following way:

```
+ (0) b
+ (0) x
+ (0) x
```

Figure 28: Optimal results (as would be delivered by an LCS-based algorithm)

As we can clearly see above, the simple diff algorithm can be easily trapped into delivering far from optimal results. This is because of its simple approach, where it compares the lines and tries to find the closest match, but doesn't find the entire LCS. Since it always tries to find a match to the current line of A and B and simply compares, which of the two matches is closer, it doesn't take into account the possibility that it might be more economical to skip the current line and find much closer matches in the following lines. And once it selects the closer match to the current line, it then skips all following lines (up to the match it found), possibly skipping much better matches.

**3.3.1.2 LCS-based diff algorithm** Due to the inefficiency of the simple diff algorithm, we also want to implement an LCS-based algorithm that will deliver optimal results. The principles of LCS and the respective algorithm(s) are described in Chapter 2.2.1.1. Since it is a well-researched problem with standardized solutions, we do not need to write an LCS-finding algorithm from scratch - doing so would be an unnecessary reinventing of the wheel. Thus, we use an available LCS algorithm <sup>4</sup>.

Let us run a quick example with the LCS algorithm with two sample files (file A containing abcdefgh and file B containing bcxdyefg):

<sup>4</sup><https://github.com/eloquent/php-lcs> (last access: 7.11.2017)

```

File A:  a b c   d   e f g h
          | |   |   | | |
File B:   b c x d y e f g
LCS:      b c   d   e f g

```

Figure 29: LCS for two sample files

The LCS-finding algorithm delivers us a correct LCS of the two files (in this case bcdefg) that we can further work with in order to obtain an optimal diff.

Now that we have discovered the LCS, we can make an optimal diff of the two files. We will be traversing the LCS one entity at a time (in the example above an entity is a letter, for a text file it would be a line and in the case of the process models, it is an array value) and comparing it to the files A and B. If we find a value in A that isn't included in LCS, we will mark it as a deletion, if we have a value in B that isn't included in LCS, we will mark it as an addition. We will do this for the whole LCS sequence in order to obtain the complete diff.

These are the steps of our diff algorithm, based on the LCS results:

1. Take first element of LCS
2. Compare to first elements of A until match is found
3. Mark all the elements of A before match with LCS as deleted
4. Compare to first elements of B until match is found
5. Mark all the elements of B before match with LCS as added
6. Move LCS pointer to the next element
7. Repeat 1-6 until end of LCS
8. Add outstanding elements at the end of A as deleted
9. Add outstanding elements at the end of B as added

This way, we obtain the following diff for the sample sequences (a,b,c,d,e,f,g,h) and (b,c,x,d,y,e,f,g) (as depicted in Figure 29):

```

- (0) a
+ (3) x
+ (4) y
+ (9) i

```

Figure 30: LCS-based diff results for (abcdefgh) vs (bcxdyefg)

And the following results for (a,x,x,b,x,x) and (b,x,x,a,x,x,b,x,x):

As we can see above, the results are an optimal diff in the sense that it delivers the shortest possible sequence of operations needed to transform file A into file B - as opposed to the results of the simple diff algorithm, which delivered a diff that would indeed transform file A into file B, but in many cases far away from the minimal length of operations.

```

+ (0) b
+ (0) x
+ (0) x

```

Figure 31: LCS-based diff results for (axxbxx) vs (bxxaxbxx)

**3.3.1.3 Results of structure diff** We have examined a few cases where the simple diff algorithm fully exposes its weaknesses and performs significantly worse than an LCS-based algorithm. It needs to be said that in most everyday cases, it performs well, there are only certain cases where it performs really bad.

Due to these quality issues, it is understandable that we will be using the LCS-based algorithm for any further work. As has already been said earlier, our script (from now on using solely the LCS-based diff algorithm) delivers two important resulting artifacts - one of them being a textual export of diff made for all significant instances and the other one a graphical representation, made with the use of Graphviz. As for now, both of these artifacts only include structural diffs, while at the same time show all significant instances (also those where only the endpoints or parameters have changed), so some instances appear to have no (structural) changes.

Let us have a quick look at a fraction of the produced graph (showing the diff of file A to B next to the edge between those two files):

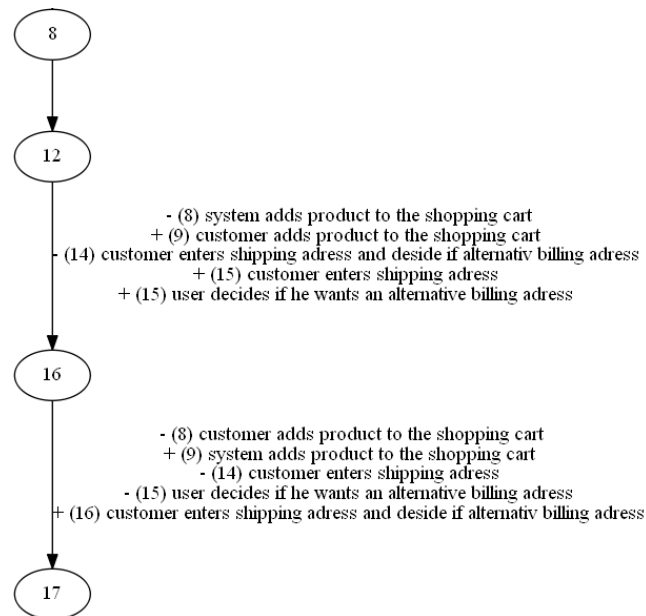


Figure 32: Graphical export of structural diff

A fraction of the textual artifact looks the following way:

As we can see at both listings above (graphical and textual), there are no structural changes between process instance 8 and 12, while there are a number of structural changes between process instances 12 and 16, as well as 16 and 17.

```

.....
8 => 12

12 => 16
- (8) system adds product to the shopping cart
+ (9) customer adds product to the shopping cart
- (14) customer enters shipping address and decide if alternativ
    billing address
+ (15) customer enters shipping address
+ (15) user decides if he wants an alternative billing address

16 => 17
- (8) customer adds product to the shopping cart
+ (9) system adds product to the shopping cart
- (14) customer enters shipping address
- (15) user decides if he wants an alternative billing address
+ (16) customer enters shipping address and decide if alternativ
    billing address
.....

```

Figure 33: Fraction of sample structural diff

### 3.3.2 Diff of Endpoints

Aside from the structural changes, we also want to discover any changes in endpoints or (activity) parameters in the process instances. If an instance doesn't introduce any structural changes, we simply want to compare all activities' endpoints and parameters between the two instances. However, if any structural change has occurred (e.g. an activity was removed), we want to compare the endpoints and parameters only of those activities that remained untouched. For this purpose, we enhanced the structure diff algorithm with a function `get_same_lines`, which gives us the numbers of all unchanged lines in both instances. With this information in hands, we can now make a diff of the endpoints (and parameters).

In CPEE, each activity can only have (none or) one endpoint, so in our json export, we simply have a single-level array of all the endpoint values. We will simply compare the endpoints of each unchanged line in order to obtain a diff of the endpoints. Each endpoint value that has changed, will be marked as changed (for a simple value comparison, we do not need to utilize the operations add or remove, as it is the case with textual diff).

### 3.3.3 Diff of Parameters

Parameters, as opposed to endpoints, are not stored as a single-level array with 0 to 1 values per activity. There can be any number of parameters per activity, meaning that each activity can have 0..n parameters. For this reason, the parameters array will contain an array of 0..n values for each activity in a process model.

Since we don't compare only single values with each other, but rather a list of values, we will be determining the changes made by three operations:

- add (value)
- delete

- change

By using these three primitive operations, we will determine a list of 0..n changes per activity for the whole process instance.

### 3.3.4 Complete Diff Results

By executing the diff of structure, endpoints and parameters, we will obtain a final diff result of all the significant instances.

Let us have a look at a fraction of the diff graph:

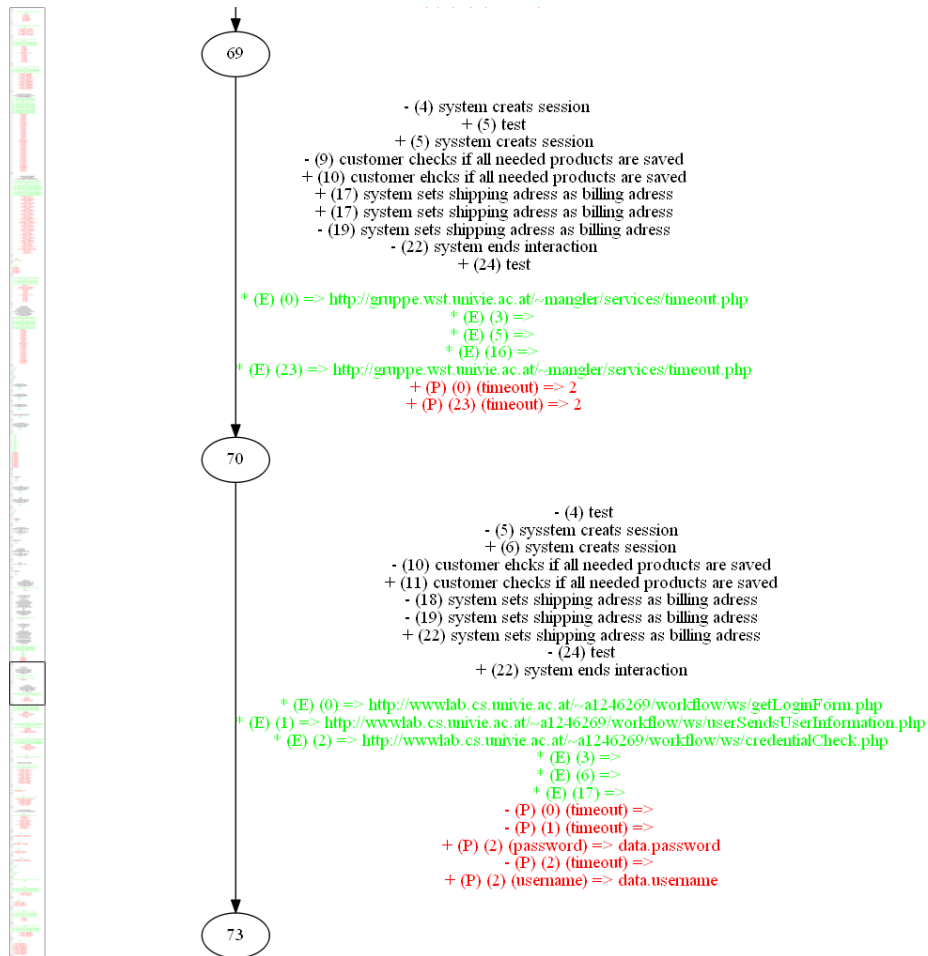


Figure 34: Graphical export of complete diff results (scale on left)

The fraction of the graph shows the differences in instances 70 (from 69 to 70) and in 73 (from 70 to 73). Both these instances are nice for demonstration, as they contain all three kinds of changes. The black lines represent structural changes, with the plus or minus at the beginning of each line indicating an addition or deletion. After the operation, the line numbers are shown and after them comes the new content of the line.



Green lines represent endpoint changes, which is further indicated by a capital E. Since endpoints have a single value, we do not consider multiple operations, only simply a change. Thus, a change of an endpoint means that it has changed, regardless of whether its value was added, removed or updated.

Parameters are represented by red lines and also indicated by a capital P. Here, multiple values may have changed for a single line number, as the line number represents the respective activity, while at the same time one activity may have multiple parameters. We do distinguish here between adding, deleting and updating a parameter.

### 3.4 Proposed Merge Algorithm

The ultimate motivation of this work is to propose a merging algorithm for business process models that will work as automated as possible - mainly by handling any possible conflicts in an automated way, making use of syntactic and semantic aspects of BP models. In order to provide such an optimal algorithm, we first want to discuss all relevant aspects and thoughts on the specific features of business process models. First, we want to make a thorough brainstorming on how the desired merging algorithm should work - what class of merging it should provide, what it should take into consideration and what optimizations in relationship to BPM-specific situations it should undertake.

Next, we want to examine more deeply how merging can be optimized in regard to business process modeling. We want to compare the complexity of generic programming languages vs. the complexity of BPM languages. This will offer us a lot of insight into how to develop the BPM-specific merge algorithm. We also want to take an in-depth look at process models in the context of CPEE, which is our source of sample models taken from real students in a university course (for more details see Chapter 3.1). We will investigate how CPEE handles activities in a BPM instance, how they are executed, what and how is done to the data and how the order of activities may affect the overall process. All this information is important for us to make assumptions on different aspects of the merge algorithm.

As has been shown, the challenging part in improving merging algorithms are the merge conflicts - when the 2 versions to be merged collide at the same spot, each of them introducing a change at the same spot [1]. Thus, we want to focus strongly on resolving the different kinds of conflicts and to provide an exact solution with an underlying argumentation for the possible conflicts. First, we will examine in which specific situations a conflict may be resolved automatically and when it needs to be resolved manually by the human user(s). Next, we will define a series of simple heterogeneous conflict scenarios that may occur when merging BPM instances - scenarios in connection to the change operations Add, Remove and Change parameter. For each such scenario, we will provide a desired solution, making use of the lower complexity of BPM (as opposed to generic programming languages) and its semantic aspects.

Finally, we will propose a working merge algorithm based on all the previous assumptions and thoughts. It will be using the previously defined and implemented diff algorithm (see Chapter 3.3), as well as the preprocessing scripts, described in chapters 3.1 and 3.2. It will fulfill the functional requirements as defined in 3.4.1, make use of all the relevant and useful semantic aspects of BPM as described in Chapter 3.4.2 and handle conflicts in a manner as automated

as possible. It will distinguish between automatically and manually resolvable conflict types and handle the various conflict types as discussed in 3.4.4.

This proposed merge algorithm is the ultimate goal delivered by this thesis and, as such, is of great importance. We will precisely describe the way it works, show a pseudocode representation, list the possible list of arguments to adapt some of its settings and also show some results of real executions of the implemented algorithm.

### 3.4.1 Desired Merge Algorithm

In order to develop a BPM merging algorithm, let us first define the requirements we have to such an algorithm. We want to develop a BPM merging algorithm with following characteristics:

- 3-way
- Semantic
- Automated merge
- Merging not just linear text, but a multidimensional tree

Let us go through each of the points.

It is very clear that we want to develop a **3-way** merge that has a number of advantages in comparison to a 2-way merge [1]. We do have the previous instances of BP models available, so we can perform a 3-way merge. It offers us more insight into how the changes have occurred in the newer instances and into how to resolve conflicts.

We want to suggest a **semantic** merging algorithm in order to automate the merging (and mainly the conflict handling) as much as possible. As has been stated in [1], with the right use of syntax and semantics of the certain domain language (in our case the business process modeling domain represented by the CPEE refined process structure tree [17]) it is possible to significantly improve merging and conflict resolution. This way, we can leverage a simple generic merge (that doesn't utilize any syntactical or semantic aspects) to a smarter merge with higher efficiency rates (in terms of successfully automated merges).

As has already been said, we want to develop a BPM merging algorithm that will try to merge the highest possible number of instances **automatically**, i.e. without the intervention of a human worker. Merging changes from diffs that do not interfere with each other, is easy. The tricky part is merging changes from diffs at the same location, i.e. conflicts. In such cases we need to rely on the knowledge we have of the specific domain language and utilize it to the maximum. In case of programming languages, e.g. we can utilize the syntax of a programming language to determine a comment and conclude that it will not affect the real code, or semantics to see the relationship of certain blocks. In our use case, we will make use of the relevant knowledge we have on BPM in general, and just as important, the knowledge we have on the BPM concepts in CPEE. We will sum up this knowledge in the following subchapter 3.4.2.

Last but not least, our algorithm will not solely merge linear textfiles (like a generic merge algorithm), but a multidimensional tree. A BPM instance in CPEE is stored as a **RPST** (Refined Process Structure Tree), making it a more

complex structure than just a multitude of lines. Also, we do not compare (and merge) only the names of the activities, but also the whole underlying information stored within these activities - mainly the endpoint and parameters.

### 3.4.2 Semantics of BPM Merging

In order to make a highly efficient merging algorithm, we want to utilize the semantics of BPM to its maximum. We want to examine the general characteristics of business processes, of BP modeling, as well as specific CPEE aspects that we can make use of.

First, we will compare and contrast the complexity of generic programming languages vs. BPM. By doing this, we want to show that BPM is far less complex and limited in the things we have to consider.

Next, we will discuss the syntactical and semantic characteristics of BPM, especially in regard to useful facts for improving and automating our merging algorithm. We will not only elaborate the general syntax and semantics of BPM, but also the specific aspects in CPEE that define activities, work flow, data flow and that will help us optimize merging.

**3.4.2.1 Complexity of generic programming languages vs. BPM** In the world of software development, diff and merge have been around for a long time. It is an essential tool in any teams of more than 1 developer working on the same project and enables not only small, but also big groups of developers working at the same resources at the same time.

In the field of software development, the diff and merge algorithms mostly work as simple textual tools comparing line by line. There is hardly any deep semantic optimization underneath - the reason being that IDEs (and version control tools) support a variety of different generic programming languages (also within the same project) and that generic programming languages offer almost endless possibilities of what the programmer can accomplish.

Since we are developing a BPM-specific merging algorithm (and implementing a prototype), we do however want to focus solely on BPM and make the most use of its specific semantics. BPM is much less complex and has a limited number of entities, operations, and other aspects to consider.

Let us make a quick comparison of generic programming languages vs. BPM:

- Programming code
  - Can be very complex
  - Offers almost endless possibilities of what can be accomplished
  - When merging (especially trying to automatically resolve conflicts), we need to be very careful
  - Conflicts, when incorrectly resolved, may cause errors
  - We need to consider many different aspects to automate merge:
    - \* variables being defined/changed
    - \* function calls
    - \* write to data or db
- BPM

- significantly less complex
- significantly less aspects to consider

=> possible to resolve conflicts automatically to a great extent

**3.4.2.2 Using semantic knowledge to optimize merging** We want to make use of any syntactical and semantic knowledge we have in order to optimize the merging algorithm - especially to automate resolving conflicts, which is the decisive part of merging. Thus, we will examine all the various aspects of BPM that are of use to us.

First, we will briefly discuss the syntactical characteristics of process instances in CPEE. Next, we will examine aspects of BPM in general and finally, the specific characteristics of process models in CPEE.

From the syntactical point of view, a process in CPEE is stored as an XML file - for a detailed description of its format see chapter 3.1.1. For the syntax of the eXtensible Markup Language (XML) see [61].

**Semantics of BPM** As has been said, the complexity of BPM is much lower than that of generic programming languages. Thus, we want to use this fact and utilize the semantics of BPM to collect useful knowledge to help us optimize our merging algorithm.

In a business process model (as e.g. opposed to programming code), there are a limited number of entities, such as:

- activity
- decision (split and join)
- parallelisation (split and join)
- loop

This limited number of heterogeneous entities simplifies our task significantly, since we only need to consider and handle a small variety of elements.

In a BPM, there is always an implicit control flow, starting in the start point, going through the defined relations (marked by lines or arrows) towards an endpoint. This also defines which activity comes after which activity and can strongly influence the behaviour and changeability of activities.

Further, there is also a certain data flow, when certain activities may change certain data - that once again, other activities may use. This is another aspects to consider during merging.

**Specific semantics of CPEE** There are certain specific semantic aspects of CPEE instances that will help us in optimizing the merging algorithm.

An activity in CPEE:

- is a procedure call that gets executed by the WF (workflow) engine
- has a name but generally, that is not really important. What really makes up the activity, is its place in the process plus the Web service endpoint that it accesses and the data it processes (and how it processes the data).

- usually accesses the defined endpoint (i.e. the URL of the desired Web service)
  - sends data to the endpoint
  - receives data from the endpoint
  - performs some manipulation with the received data
- => Order of activities in CPEE only defines what should be done to the data and at which endpoints (so the names of the activities aren't of great importance)
- => Therefore, as long as the data isn't affected, one may exchange the order of (conflicting) activities

This is a very important discovery because it allows us to automatically merge conflicting activities, as long as one activity doesn't directly affect the data of the other conflicting activity. Let us describe the exact way of handling conflicts in our merging algorithm in the next subchapter.

### 3.4.3 Handling Conflicts

Resolving conflicts is an essential part of an efficient merge algorithm. The more we manage to optimize the conflict resolution, the less human workers will have to interact and decide. In this thesis, we will be elaborating the possible conflict types and their resolving in a very detailed way.

First, we will make assumptions and conclusions about when it is indeed possible to resolve a conflict automatically and when, on the other hand, it needs to be manually resolved. This decision making will also be an important part of the final algorithm and implementation proposed.

After determining, which situations can be automatically resolved, we will take a close look at different kinds of conflicts that may occur. We will distinguish between three kinds of conflicts, based on the change operations involved:

- Add
- Remove
- Change endpoint/parameters

Based on these three types of conflicts, we will define possible conflict scenarios. These scenarios shall be very elementary conflict types - by combining them it is always possible to achieve more complex conflicts, but we want to look at the very roots of the conflicts.

For each of these conflict scenarios, we will deliver an insightful graph, precise description, thoughts on how to handle the situation and final decisions on how to exactly resolve the conflict. These conflict scenarios make big use of our semantic knowledge of the BPM paradigm and will also play a key role in the proposed merge algorithm - including the latter implementation.

**3.4.3.1 Automated vs. Manual Merge** The first important decision that we will make in a conflict is, whether we can resolve it automatically or not. The fundamental reasoning is described in the previous Subchapter 3.4.2 and is based on the way activities work in CPEE. As has already been stated, an activity is basically a procedure call that gets executed by the workflow engine. As such, it may (or may not) send some data to a defined Web service, get back some data and post process it. In other words, the names of the activities are not that important, the only important things are

- Which Web service endpoint is used
- What data is sent
- What data is received
- What is done to the data afterwards

In a real-world BPM, the name and order of activities would seem to be of great importance - e.g. an activity called *Wash windows* would need to come before an activity called *Dry windows*. With the knowledge of human language, we can agree on that. However, in case of CPEE, what really defines the activities, are the points mentioned above - mainly how the data is manipulated. So also if the order of activities gets changed, it is theoretically alright, as long as the data flow and manipulation doesn't get disrupted.

For the above reasons, we can make a clear distinction of when a conflict can be automatically merged and when it needs manual intervention.

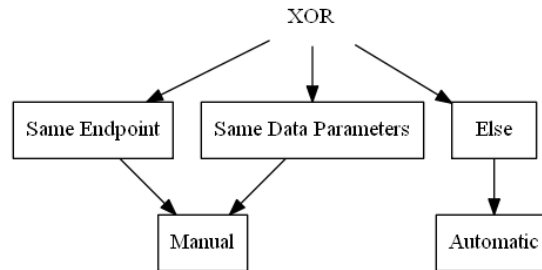


Figure 35: Manual vs Automatic Merge

Let  $a1$  and  $a2$  be the conflicting activities of the two processes to be merged (with  $a1$  being merged first). Then we distinguish:

- If  $a1$  and  $a2$  use the same endpoint, manual conflict resolving
- If  $a1$  changes some data parameters that  $a2$  will use, manual conflict resolving
- Else use automatic conflict resolving

Let us discuss these situations in more detail.

If  $a1$  and  $a2$  use the same endpoint, it can be assumed that they might collide in some way. Accessing the same endpoint means they send/receive data to the same Web service, which will perform some operations on the respective data.

It is therefore to be expected that if both of the activities send and manipulate their data at the same Web service, some data misuse may very well be in place - and it is necessary for a human worker to investigate their relationship to each other, as well as whether it would be safe to merge the two colliding activities.

There are a number of parameters sent by CPEE during the execution of an activity that can be defined by the user for each activity. They get sent as part of the HTTP Request to the defined Web service endpoint that is supposed to perform some work on them. Some of these parameters are so called data parameters - they start with *data...* . Let us have a look at a small example of parameters of an activity within our sample:

```
"parameters": {
  "type":      "login",
  "role":      "data.userRole",
  "organisation": "data.userOrganisation",
  "form":      "data.loginForm",
  "entries":   "data.loginFormEntries"
}
```

Figure 36: Parameters and data parameters (highlighted red) of a sample activity

These data parameters not only get sent to the defined Web service, but are also used by the activities in CPEE. When the activity gets executed, these data parameters are read or written and used to alter the outcome of the process.

Therefore, in case of a conflict, we have to examine whether the data parameters of the two conflicting activities a1 and a2 are not affecting each other. First, we will extract only the data parameters from the whole list of parameters. Then, we will see any data parameters that get changed in a1 and see if they are sent along to the Endpoint in a2. If so, it means once again, we need to have a human worker to intervene and make manual decisions on whether these conflicting can be merged.

If neither the endpoint of a1 and a2 is the same, nor do the data parameters collide, we can safely perform automatic conflict resolving.

Considering the order of the conflicting activities to be added to the resulting process instance, we will simply take a1 first and a2 second - with a1 being the activity from the first diff file and a2 from the second diff file. As has been said, the order of the activities matters only if the endpoints or data collide.

**3.4.3.2 Add operation** At first, we will examine possible conflict scenarios, where the conflicting change in diff 1 and diff 2 are both of the operation Add. We have determined two scenarios where the second one is a more complicated version of the first one.

**Scenario 1** Let us see an insightful graph of the first conflict scenario, where an activity is added at the same spot:

First, let us describe the format of the graph (and all subsequent graphs of conflict scenarios). First column shows diff 1, second column shows diff 2 and the third column shows the final merge. The original file is not shown, as it can be implicitly derived from diff 1 and diff 2 - in this case the original file was A

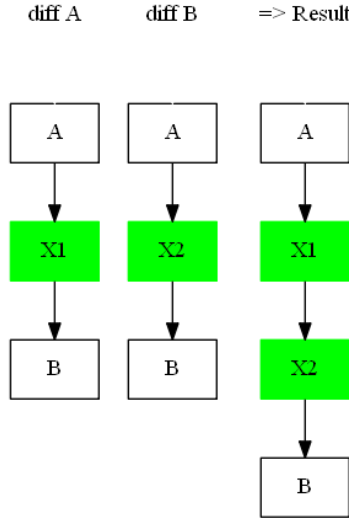


Figure 37: Conflict scenario 1

First column shows diff 1, second column shows diff 2, third column shows final merge. Green means add, red means remove (not applicable here), blue means change endpoint/parameter (not applicable here).

– > B. In diff 1, an activity X1 was added after A, while in diff 2, an activity X2 was added at the same place. This is where the conflict arises.

The last column shows the merged result. In this case, we can see that it is A – > X1 – > X2 – > B - more details will be explained in a few lines.

The green boxes mean an activity that has been added, red box means removed (is not applicable in this scenario, but will be in latter), blue box means changed endpoint or parameter(s) (again not applicable here).

Now on the conflict scenario itself. We assume (as in all the conflict scenarios listed here) that it has already been determined that we can perform an automatic conflict resolution - there are no endpoint or data parameters conflicts among the conflicting activities. In that case, we can resolve our first conflict scenario by simply putting the added activities X1 and X2 in a sequence behind each other. As has been said, the activity from diff 1 comes first, which is in our case X1, followed by X2 (from diff 2).

This way, we can resolve the conflict in an automated and very elegant way without the need of manual conflict resolving. Whenever there is an activity added at the same place (having the same predecessor) in both diff 1 and diff 2 - and when it has been shown that we can perform automatic merging - we will simply add the colliding activities after each other.

**Scenario 2** This is a second similar conflict scenario where activities were added at the same place:

The second conflict scenario is slightly more complex than the first one. While diff 1 stays the same, diff 2 introduces not only 1 but 2 subsequently added activities at the same place (right after A). It needs to be said that whether this bigger change happens in diff 1 or diff 2 doesn't matter, it would



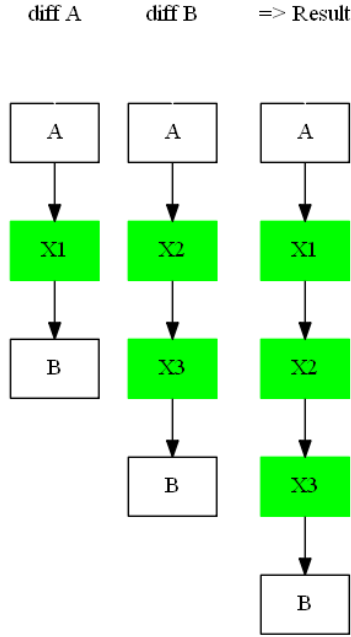


Figure 38: Conflict scenario 2

have the same consequences.

So now we have two diffs adding activity/ies at the same place, where the second diff adds 2 activities in a row. If we were about to make a generic merge assuming we have no background information, one might be in dispute how to merge this conflict:

- A -> X1 -> X2 -> X3 -> B
- A -> X2 -> X1 -> X3 -> B
- A -> X2 -> X3 -> X1 -> B

However, as we do know the specific semantics of BPM and CPEE, we know that we can basically chose the fist variant and perform the automatic merge. In the preceeding step, we have already made sure that there is no endpoint or data parameters conflict, so we can proceed with automatically merging the scenario.

**3.4.3.3 Remove operation** After describing the possible conflicts that may arise using the add operation, we now take a closer look at those conflicts arising from removing certain activities. Here, we can indentify a higher number of different scenarios, so let us discuss each of them.

**Scenario 3** The first remove scenario is very simple:

In this simple scenario, we can see that the same activity (B) was removed in both diffs. This naturally leads to a conflict, since a change is performed at

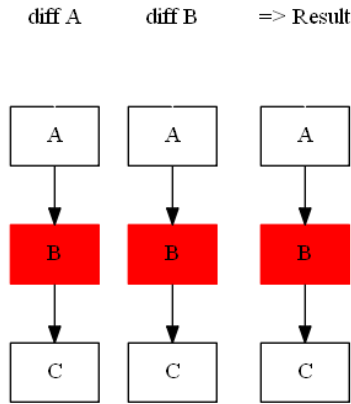


Figure 39: Conflict scenario 3

the same place in the file - right after A. However, we do not require a manual resolution of the conflict, since both diffs basically say the same - remove activity B. Thus, we can proceed by removing the activity B and merging to the resulting file.

**Scenario 4** The next remove scenario is a more complicated situation, where multiple activities were removed in each of the two diffs and they happen to be at a conflicting spot:

In this scenario, both diff 1 and diff 2 introduce a number of remove operations that collide. From the original file  $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$ , diff 1 removes nodes (i.e. activities) B and C. If only this patch were to be applied, it would be leaving us with a changed process  $A \rightarrow D \rightarrow E$ .

Diff 2, at the same time, removes activities C and D from the original file. If we were only to apply this patch, it would result in a changed process  $A \rightarrow B \rightarrow E$ .

As we can see, these two diffs have a conflict with each other, crossing over one another. By just implementing a generic merging algorithm, we might end up having a conflict announced and needing to resolve it manually - which is undesired, since we aim to automate most of the workflow.

Therefore, in our algorithm we will solve this situation by simply incorporating all of the mentioned remove operations. This is very reasonable, since in fact, both of the diffs said to remove the same thing (as in scenario 3), plus each of them also removed another thing (which is not in conflict). So thinking about it, both of the workers who created diff 1 and diff 2, wanted to have the same thing removed - which gives us no reason to hesitate about whether it would indeed be correct to remove it.

As a result, we will remove activities B (which was removed in diff 1), C (which was removed in both diffs, so we actually have a double confirmation and intention of removing it) and D (which was removed in diff 2). The resulting merged file contains  $A \rightarrow E$ .

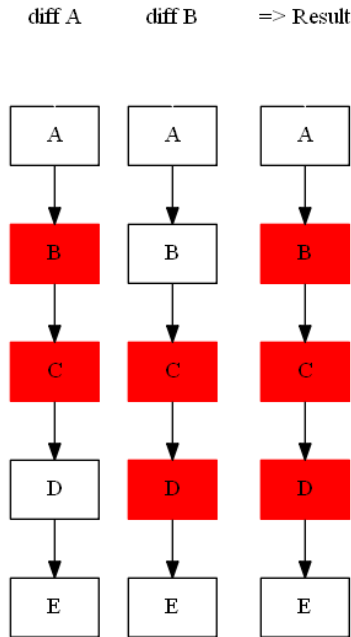


Figure 40: Conflict scenario 4

**Scenario 5** Scenario 5 shows a combined conflict of adding and removing - one diff removes an activity, whereas the second diff adds an activity at the same place:

As we can see in the graph, diff 1 removes the node B - which is the node following right after A. Diff 2, on the other side, adds a new node X1 at the same spot - right after A.

This is a very tricky situation where one might prefer to force manual conflict resolving (in order to avoid any errors that might be caused by the conflict). However, with all the assumptions that we have made in the previous subchapter 3.4.2, we can safely say that these two diffs can be merged. This is where the importance of the underlying analysis and decision making is unveiling in its full power - we now have much more power in terms of automatically merging conflicts, that otherwise would need manual intervention and would unnecessarily waste human resources.

If we have the necessary requirement for an automated merge fulfilled (namely that there is no endpoint or data parameter conflict), we can proceed to automatically merge the presented two diffs. Since diff 1 wants to remove the activity B (following after A) and diff 2 wants to add an activity X1 right after A, we can safely execute both. We will remove B and add X1 right after A - resulting in the merged file A -> X1 -> C, that can be seen in the graph.

**Scenario 6** Scenario 6 shows another combined conflict of adding and removing - with diff 1 removing an activity, after which diff 2 wants to hook an added activity.

This scenario is slightly similar to the previous scenario 5 in that it also

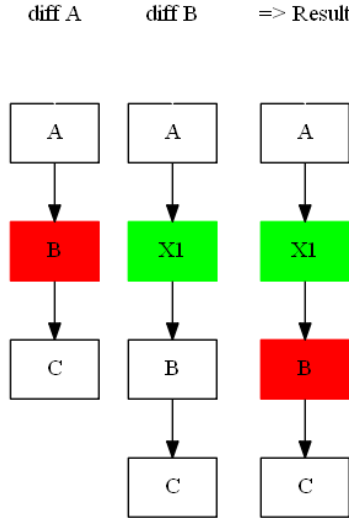


Figure 41: Conflict scenario 5

combines adding and removing one activity in the diffs, respectively. However, while scenario 5 was adding and removing an activity with the same ancestor (activity A), scenario 6 removes an activity which at the same time serves as the ancestor for adding another activity.

In other words, diff 1 wants to remove activity B (diff 1 is the same for scenarios 5 and 6). However, diff 2 doesn't add activity X1 after A (as was the case in scenario 5), but rather after B.

The challenge here is that the activity X1 is supposed to be added after an activity that at the same time gets removed (so we basically lose the hook to which it should be attached). The question is, what should we do in this case, when we want to add a new activity after a non-existent one?

The answer is in fact quite simple: We can easily incorporate both of the suggested changes without any problems. First, we will remove activity B. There is no hesitation about that, since diff 1 removes B and diff 2 ignores it. And then, we will add X1, hooking it right after A (once again, since diff 2 adds X2 to this spot and diff 1 ignores it at this point).

This way, we will obtain the final merge:  $A \rightarrow X1 \rightarrow C$ , just like we did in scenario 5.

**Scenario 7** Scenario 7 is a more complex version of scenario 6 that tries to remove multiple activities from the process, that at the same time serve as hooks for newly added activities.

In scenario 7, we can see a more elaborated version of multiple add and remove operations carried out at the same time. If we only subtracted one half of the changes from both diffs (only the first operation of both respective diff files), we would obtain scenario 6.

In here, we want to elaborate more on the issue of removing nodes that, at the same time, should serve as ancestors, i.e. hooks for newly added activities. We further develop this possible conflict case to decide what should happen in

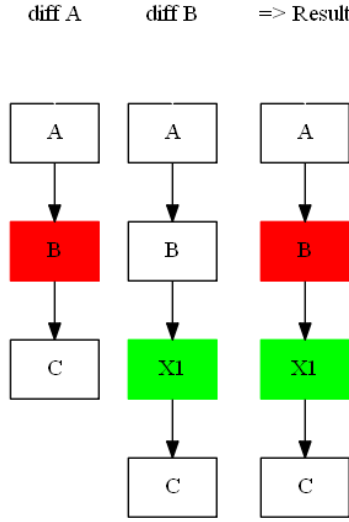


Figure 42: Conflict scenario 6

these cases (and latter in the evaluation to test if it does so).

Diff 1 removes not only activity B, but also activity C. If only diff 1 was to be applied to the original file, we would obtain a resulting file containing only A.

Diff 2 at the same time adds an activity X1 after B (just like scenario 6) and activity X2 after C. If only diff 2 was to be applied, we would obtain A -> B -> X1 -> C -> X2.

When we want to merge both of these diff files, we can use the same decision making as in scenario 6 as our base. In spite of scenario 7 being more complex than scenario 6, we can use the same principle. We will simply remove the node to be removed and afterwards add the node to be added (hooking it up to the node before the removed node). In other words, we will execute the same principle as in scenario 6, as many times as it occurs. Were it about to happen 10, 20 or n times in some test case, we would run it 10, 20 or n times.

So we will remove B (as suggested in diff 1), add X1 after A (as written in diff 2), remove C (as requested in diff 1) and finally add X2 after X1 (as demanded in diff 2).

The final merged file, as you can see in the last column of the graph, has the following content: A -> X1 -> X2.

**3.4.3.4 Change endpoint/parameters operation** A separate category of conflicts is when endpoints or parameters of activities get changed, conflicting with remove operations. Add operations do not really introduce any conflicts, since adding an activity and updating parameters or endpoint of another activity does not lead to a conflict. However, removing an activity that at the same time is supposed to be updated in the other diff file, is something to think about.

We will generally assume here that adding/removing an activity is a superior change operation to just updating endpoint or parameter of an activity. In other words, manipulating the structure of the process is superior to just

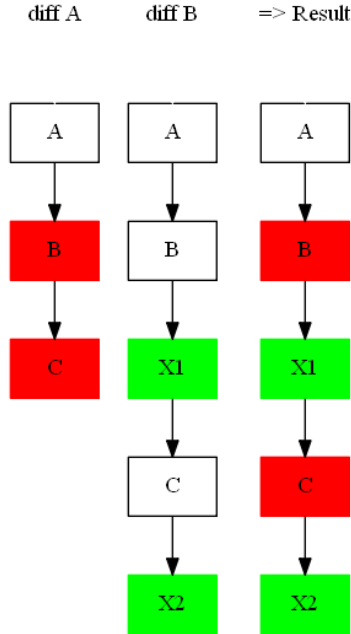


Figure 43: Conflict scenario 7

updating minor details of a single activity. As has been said, add operations do not interfere with the operations of changing endpoint/parameters. As for delete operations, these shall (for the above reasons) be regarded as superior to updating endpoint/parameters and shall thus be executed.

The reason we can make this assumption is that if a human worker has once already decided that an activity may be removed, we don't need to change the minor parameters of that activity any more - a human person has already decided it is ok to remove the activity as a whole.

As a conclusion, once we decide to treat the change endpoint/parameters operation as inferior to the remove operation (or add operation but that does not cause conflicts), we can derive a general rule that (for all scenarios of the operation change endpoint/parameters) to just ignore the change endpoint/parameters operation and not classify it as a conflict. (since we have shown that we will treat the remove operation with higher priority and carry it out, so the change endpoint/parameters operation becomes irrelevant).

For the purposed of reference and for brainstorming, we still do want to examine the cases where a change endpoint/parameters operation may collide with the remove operation. However, as was declared, the outcomes will be defined by the prioritization rule of the remove operation.

It is naturally possible, in the future, to alter this decision making and to adapt the algorithm.

In this subchapter, we will go through three different cases of updating activities (in terms of conflicting remove and update endpoint/parameters operations) and discuss, how they are to be resolved.

**Scenario 8** Scenario 8 is a simple case of two conflicting operations at the same activity: change endpoint/parameter(s) and remove.

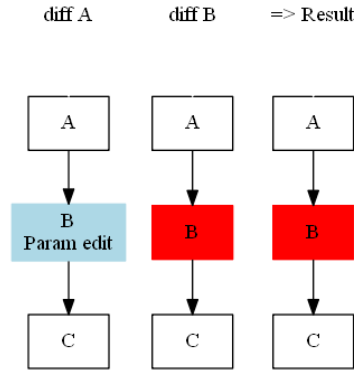


Figure 44: Conflict scenario 8

As we can see in the graph above, scenario 8 depicts a case when both diffs introduce an operation to the same activity: B. First, let us repeat that blue boxes mean a change endpoint/parameter operation. So, as can be seen, diff a updates the endpoint/parameter(s) of activity B. At the same time, diff 1 wants to delete the activity B.

As has been said in this subchapter, we will generally treat a remove operation superior to change endpoint/parameters. This means that whenever we see this kind of conflict, we will simply remove the activity, as a human worker has decide to do (in this case in diff 2) - since a human person already decided that we can remove the whole activity, we don't need to consider smaller operations of updating some details of the activity, it has already been decided it can be removed as a whole.

In this scenario, we can therefore ignore the change endpoint/parameters operation from diff 1 and execute the remove operation from diff 2. If we were to generally define the change endpoint/parameters operation as inferior to remove and to decide to generally not even treat it as a conflict, we could (in all scenarios of the operation change endpoint/parameters) just ignore this operation and not classify all such cases as conflicts.

We will obtain the merged file A → C.

**Scenario 9** In scenario 9, we can see a special case of changing endpoint/-parameters and renaming an activity at the same time.

Scenario 9 is a special case, where we tackle the issue of renaming an activity. As has been stated and argued in 3.2, we handle the name of an activity as its identifier.

It would also be possible to use the id (delivered in the CPEE process file) of each identity as an identifier. In that case, renaming an activity would be similar to a plain parameter change - the parameter that would change is the name of the activity, since id would still remain the same.

However, we chose to use the name of the activity as its identifier, mainly due to the fact that in a business process model, the names define the activities. It is naturally possible that someone carries out a simple misspell correction and

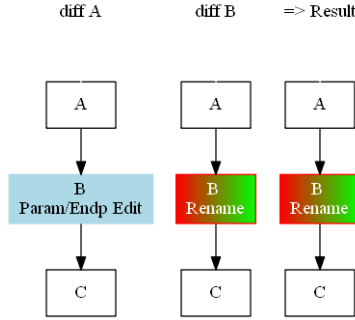


Figure 45: Conflict scenario 9

it will be identified as removing and adding a new activity. At the same time, using ids as identifiers would bring certain disadvantages, mainly that often older versions of BPM instances are taken and the old activities get renamed to new ones - which are semantically entirely different and change the semantic structure of the entire process. However, using ids as identifiers, we would still regard them as the original activities with only minor changes to the name. To read a discussion of this topic in detailed manner, go to 3.2.

Since we are using the name of an activity as the identifier, renaming an activity means removing the activity and adding a new one in its place with the new name. In the graph of scenario 9, we show this by using the box with a gradient filling from red to green (suggesting that a remove and add operation occurs at the same time and spot).

As has been said in scenario 8, we treat a change endpoint/parameters operation inferior to the remove operation, meaning we ignore it and perform the remove operation. This rule is also applied here. The only difference is that renaming is split into two separate primitive operations of removing and adding an activity and that's what happens here.

The activity B is removed and in turn, a new activity is added with the new name (for sample purposes, let us refer to the new activity as B1 now). In the graph, we could proceed to split up the renaming into the two separate primitive operations, however the result would be the same, so it is not necessary.

The final process (referring to the new activity as B1) is  $A \rightarrow B1 \rightarrow C$ .

**Scenario 10** Scenario 10 shows a more advanced version of scenario 8 with a change endpoint/parameter operation conflicting with a sequence of remove operations:

In the final scenario 10, we can see a more elaborated version of scenario 8. It also depicts a conflict of a change endpoint/parameters operation with a remove operation. However, we want to examine the case when the whole surrounding block of activities around the relevant conflicting activity is being removed - including the ancestor.

In other words, we remove the activity, right after which the activity is hooked (along with two more following activities).

Once again, since (one of) the conflicting operation is an endpoint/parameter change, this conflict does not pose a problem. We can imply the rule of treating



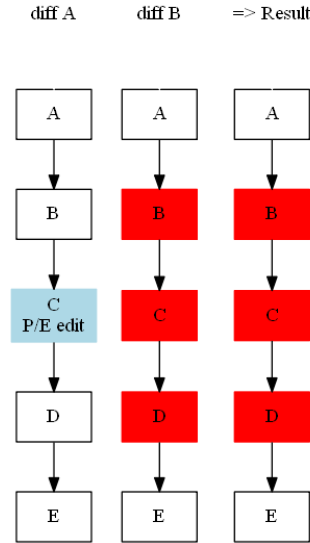


Figure 46: Conflict scenario 10

an endpoint/parameter change as an inferior operation to removing and ignore it.

In effect, we can carry out all the remove operations listed in diff 2 and remove activities B, C and D.

The resulting merged file will be: A – > E.

#### 3.4.4 Proposed Merge Algorithm

After discussing all the important aspects that need to be considered in the merging algorithm, we can finally proceed to design and implement it. In previous subchapters, we have talked about the qualities we want the merging algorithm to have (such as being a 3-way, semantic, automated merge merging not only linear text, but the whole CPEE process instances), we have discussed the various semantic aspects of business processes and CPEE instances that will be useful and crucial to optimizing the conflict resolving and finally, we have talked a lot about how to resolve conflicts in our algorithm - first by analyzing when an automated conflict resolution is viable and second by discussing a number of heterogeneous conflict scenarios that might occur in real situations.

To best describe the merging algorithm, we will list a pseudocode representation, strongly simplifying the real code, but showing the essential parts of the code. The goal is to make a certain abstraction, so that the structure of the algorithm is easily understandable and not too complicated.

At first, we will show the pseudocode of the general part of the algorithm, including the parts when there is no conflict. To achieve better overview, we will then separate the parts of the algorithm handling conflicts into a separate pseudocode listing. This is in order to keep the readability and comprehensibility of the figures at a good level.

**3.4.4.1 Pseudocode** Let us go through the simplified pseudocode of the main parts of the merging algorithm:

Listing 1: ABC

```

1  // Repeat until finished
2  while (not finished) {
3
4      // If end of file x was reached (original file) => finished
5      if (index_x == length_x) {
6          finished
7      }
8
9      // Look for next change in diff A (+ or -)
10     for (a = index_a to length_a) {
11         ...
12     }
13
14     // Look for next change in diff B (+ or -)
15     for (a = index_b to length_b) {
16         ...
17     }
18
19     // Check if a conflict occurred
20     if (a_changes overlaps with b_changes) {
21         *****
22         => See Figure 48 for conflict handling
23         *****
24     }
25
26     // No conflict
27     else {
28         // If no more changes, output rest of file X
29         if (a_changes == 0 && b_changes == 0) {
30             ...
31         }
32
33         // If next closest change is from diff A
34         // => Output unchanged lines from X and the change
35         else if (a_changes <= b_changes) {
36             ...
37         }
38
39         // If next closest change is from diff B,
40         // => Output unchanged lines from X and the change
41         else {
42             ...
43         }
44     }
45 }

```

Figure 47: Pseudocode of the merging algorithm  
(Conflict handling is shown in Figure 48)

The algorithm states at the beginning that it will run until it says it has finished (so there is not a limited number of iterations in the beginning, it will run as long as it doesn't decide it has fulfilled its mission). This is quite understandable, since we cannot say an exact number of iterations at the beginning

- it all depends on how it evolves throughout time and how the 2 diff files will stepwise be merged into original file to create the final output file.

Once the working index in the original file reaches the end of this file (i.e. the length of this file, which represents the number of lines), the algorithm says that it has finished and will shut down.

At first, during each iteration, we want to find the next change that's appearing in both diff files. That's what we do at lines 9-17 in the Figure 47. Generally, we have indexes for both files depicting which lines from both files have already been integrated into the output file and we just look at what lines of file X (original file) the next changes in diff A and diff B influence (e.g. diff A wants to change line 5 of file X, whereas diff B wants to change line 10 of file X).

After having determined the number of lines that the next changes in diff A and diff B want to change in the original file, we check if these changes are in a conflict or not.

If the changes from diff A and B overlap, we have a conflict. Resolving a conflict is a big issue (and in a way the most important part of the entire merging algorithm, since the success rate of automatically resolving conflicts is what really makes up a good merging algorithm) and is quite complex. For this reason, we will show the pseudocode for handling a conflict in a separate Figure 48.

If there is no conflict (i.e. the next changes in the two diff files don't target the same spot in the original file), we differentiate three cases. If there are no more changes (if the procedures to find the next change in both diff files states that it has reached the end of file, it sets the index variables for the diff files to 0), then we output the rest of the original file.

If the next closest change is from diff A (e.g. diff A targets at line 5 of the original file and diff B targets at line 10), we output the unchanged lines of the original file before this change and then output the changes. If the next closest change is from diff B, we do the same, but naturally in regard to diff B.

**3.4.4.2 Pseudocode - Conflict resolving** This is the part of the merging algorithm that handles conflict resolving (for the exact position of this piece of pseudocode within the merging algorithm, see Figure 47):

```
1
2 // Output unchanged lines from X before change
3
4 // Fetch changes from diff A to a variable
5 for(a = index_a to last_change_in_sequence) {
6     ...
7 }
8
9 // Fetch changes from diff B to a variable
10 for(b = index_b to last_change_in_sequence) {
11     ...
12 }
13
14 // If auto resolving conflicts is turned on
15 if (auto_resolve) {
16
17     // Extract endpoints and parameters for A and B
18     a.endpoint = ...
```

```

19     a_params = ...
20     b_endpoint = ...
21     b_params = ...
22
23     // Extract only data params
24     foreach (a_params) {
25         if (startsWith "data") {
26             add to a_data_params
27         }
28     }
29
30     // Check if endpoints or data parameters of A and B collide
31     if (a_endpoint == b_endpoint) {
32         no auto_merge
33     }
34     if (any a_data_params = b_data_params) {
35         no auto_merge
36     }
37
38     // If conflict has to be manually resolved
39     if (no auto_merge) {
40         output_manual += unresolved both alternatives
41     }
42     // Otherwise automatically resolve
43     else {
44         output += auto resolved option
45     }
46 }
47 }
48
49 // For each unresolved conflict, ask the user to manually resolve
50 // (choose option)
51 if (no auto_merge)
52     print "Choose A or B"
53     print (option_a)
54     print (option_b)
55
56     if (read (a)) {
57         output += a
58     }
59     else {
60         output += b
61     }
62 }

```

Figure 48: Pseudocode - conflict resolving

The above listed Figure 48 shows the part of pseudocode, where a single conflict is handled and resolved. The exact position of this piece of pseudocode in the whole merging algorithm can be seen in Figure 47 where the conflict handling piece of code is executed once for every conflict.

Generally, the algorithm checks to see if it can resolve the conflict automatically (based on the criteria described in Chapter 3.4.3.1) or whether it needs to be manually resolved. In the automated case, it resolves the conflict according to the scenarios described in Chapter 3.4.3. In the manual case, there is an option to ask the user to choose how to resolve the conflict, or the option to write a conflict into the resulting merged file.

Let us now go through the pseudocode shown in the above Figure 48. At first, in the case of a conflict, any unchanged lines before the conflict itself are added to the output (i.e. resulting merged file).

If auto merging is turned on (we want to have an option of disabling it), we want to see if we can indeed perform an automated conflict resolution. To do this, we will first extract the endpoint and parameters from both A and B diff files. In turn, we extract only data parameters from the whole list of parameters (those starting with data...).

Next, we check if the endpoints of A and B collide, as well as whether the data parameters of A and B are in a conflict. If any of this is true, we must not perform automatic resolving of this conflict.

As one would expect, if auto merge is evaluated as possible, it will be conducted. Otherwise, both unresolved alternatives will be added for manual resolving.

If desired, we can leave the unresolved conflicts be written into the resulting merged file. However, in most cases, it is more desirable to let the user resolve each conflict on the fly, as they are using the program. For this purpose, we show the user both options (from diff files A and B) and let them choose which option should be selected (starting at line 49 of Figure 48).

## 4 Implementation and Evaluation

In order to evaluate the success of our assumptions and semantic algorithm, we will implement a working prototype of the merge script (and all the other scripts mentioned throughout this thesis) and test it with various test cases. After doing so, we will present a clear comparison of the success rate of a generic merging algorithm and our semantic BPM merging algorithm to see, whether all the optimizations and semantic specializations have indeed lead to a significant improvement - especially of the percentage of automatically merged conflicts.

First, we will implement a working prototype of all the scripts used throughout this thesis - such as for regrouping the instances, finding significant instances (those with changes), finding diff, merging diffs and testing the results. In the subchapter 4.1 we will briefly show the usage and user interface of these scripts.

Next, we will perform three different kinds of evaluation:

- **Functional testing:** Simple automated functional testing with a few scenarios to make sure that the algorithm returns the expected results
- **Qualitative evaluation:** Testing based on a high number of test cases to see, how well the algorithm performs (especially in regard to automated conflict resolving). For this purpose, we will use all the available diffs from the student instances, split each diff into 2 separate diffs and try to merge them together. We will also merge the separate diffs with a generic algorithm to see, whether the semantic optimized algorithm is superior.
- **Quantitative evaluation:** While performing the qualitative evaluation, we will also record the merging execution time to see, whether the performance of the semantic algorithm is any worse than that of the generic algorithm.

In turn, we will present the test results in an easily understandable way with various graphs and statistics.

### 4.1 User Interface and Usage

During the writing of this thesis, a number of different scripts have been described and implemented :

- **group instances:** First script that takes all the available student instances (in our case about 6000), groups them into folders (each student has a separate folder). It only extracts the information from the instances that we need and the rest is ignored.  
*For more information, see chapter 3.1.3*
- **find significant instances:** After regrouping instances, this script iterates through all of them to see whether each instance does introduce any change at all (compared to the previous instance) and marks such instances as significant.  
*For more information, see chapter 3.2*
- **diff:** The diff algorithm implementation, which takes two input files (original and changed file) and returns the diff (i.e. patch) of these two files.

It not only makes a diff of the structure, but also of the endpoints and parameters.

*For more information, see chapter 3.3*

- **diff all:** A script to automatically perform diff on all the significant instances of a student and save the results into respective files. Aside from all the separate diff files, it also produces a nice overview graph containing all diffs of a student, as shown in chapter 3.3.4.

*For more information, see chapter 3.3*

- **merge:** The merge algorithm implementation, which takes the original file and two diff files, merges them and returns the result. It makes use of the diff algorithm to create the two diff files (if they are not yet existent). It is a semantic merge algorithm, which makes use of a number of characteristics of both business process models and CPEE-specific characteristics. The algorithm tries to merge as many conflicts as possible in an automated way, so that human intervention is not needed.

*For more information, see chapter 3.4.4*

And three more scripts will be implemented for testing purposes (with one final script to automate all the important scripts):

- **Functional test:** A script to perform the functional testing based on a limited number of scenarios. If the results are as expected, the tests have been passed.

*For more information, see chapter 4.2*

- **Test semantic merge:** An automated test script, which takes diffs from each significant instance of a student, splits the diff into 2 separate diffs (splitting even and odd lines), merges them together and examines, whether the resulting merged file is the same as expected. It also performs a big number of different statistics during this process, allowing us to have a deep look into the qualitative performance of the semantic merging algorithm.

*For more information, see chapter 4.3.1*

- **Test generic merge:** This script takes (for each significant instance of a student) the two separated diffs and merges them with the original file, using a generic textual merging algorithm. This script also stores a lot of different statistics, so that we can compare the semantic and generic merging algorithms.

*For more information, see chapter 4.3.2*

- **Batch execute:** This is the final script used to automate the execution of all the relevant scripts even more. In general, all the above scripts carry out their work with one student. Thus, the batch execute script executes the desired script for all students. After finishing (since it may take some time), it plays a beep sound to inform the user about having finished.

All of the above scripts are executed from the terminal (i.e. command line) and have a number of obligatory and optional parameters. To make it easier to use, all scripts have a help parameter, which shows the proper usage of the script. Let us have a look at the help mode of all scripts - this way we can easily see the obligatory and optional parameters for each student.

## Group instances:

```
>group_instances help

Usage of the group_instances script:
group_instances

No parameters needed
```

## Find significant instances:

```
>find_significant_instances help

Usage of the find_significant_instances script:
find_significant_instances parameters

Parameters:      Obligatory:  Example:
aMatrikelnr      *           a1234567
```

## Diff:

```
>diff help

Usage of the diff script:
diff parameters

Parameters:      Obligatory:  Example:
aMatrikelnr      *           a1234567
instance_1       *           1 (0 is infinite)
instance_2       *           4 (1 is default)
```

## Diff all:

```
>diff_all help

Usage of the diff_all script:
diff_all parameters

Parameters:      Obligatory:  Example:
aMatrikelnr      *           a1234567
```

## Merge:

```
>merge help

Usage of the merge script:
merge parameters

Parameters:      Example:
aMatrikelnr      a1234567
base_instance    1
instance_a       1a
instance_b       1b
force_diff       true      (optional)
ask_for_resolve  true      (optional)
```



```
merge_parameters  false      (optional)
merge_endpoints   false      (optional)
```

### Functional test:

```
>merge_functional_test help

Usage of the merge_functional_test script:
merge_functional_test

No parameters needed
```

### Test semantic merge:

```
>test_semantic_merge help

Usage of the test_semantic_merge script:
test_semantic_merge parameters

Parameters:      Example:
aMatrikelnr      a1234567
start_id         5 (1 is default)
max_instances    10 (0 is infinite)
force_diff       true
```

### Test generic merge:

```
>test_generic_merge help

Usage of the test_generic_merge script:
test_generic_merge parameters

Parameters:      Example:
aMatrikelnr      a1234567
start_id         5 (1 is default)
max_instances    10 (0 is infinite)
```

### Batch execute:

```
>batch_execute help

Usage of the batch_execute script:
batch_execute parameters

Parameters:      Example:
script           1/2/3/4 or name of script:
                  - find_significant_instances
                  - diff_all
                  - test_semantic_merge
                  - test_generic_merge
                  or 0 (all)
start_student    a1234567 (first found is default)
max_instances    10 (0 is infinite)
```

To perform all the required scripts to compare the semantic and generic merge from scratch - i.e. to take the approx. 6000 instance files and perform all the scripts needed - we need to perform the scripts in the following order:

1. group\_instances
2. batch\_execute 1  
(performs the find\_significant\_instances script for all students)
3. batch\_execute 2  
(performs the diff\_all script for all students)
4. batch\_execute 3  
(performs the test\_semantic\_merge script for all students)
5. batch\_execute 4  
(performs the test\_generic\_merge script for all students)

## 4.2 Functional Evaluation

In order to test the basic functionality of the merge algorithm, we will develop a basic functional testing script. It is possible to run this script at any time (so e.g. before committing any update to the merge scripts to see if it still works as expected) and the included test scenarios can easily be edited - it is very easy to add, change or remove any scenarios at any time.

As the testing scenarios, we will use the 9 various conflict scenarios that are listed in chapter 3.4.3. These are simple situations that may occur, where each single situation defines a potential conflict. For each of the scenarios, we have exactly defined the desired outcome, so it is easy to compare the result to the expected outcome. Also, these 9 scenarios are especially interesting for testing, so that we can see whether our desired way of resolving the conflict in the given situation(s) is indeed put in place.

To see the exact description, graph and the expected outcome of all the test scenarios, go to chapter 3.4.3. Now, let us execute the functional test script (showing only a fraction of the results):

```
***** 1 => 1a,1b *****
OK - Result as expected

***** 2 => 2a,2b *****
OK - Result as expected

***** 3 => 3a,3b *****
OK - Result as expected

.....

Total:
0.0402 s ...
```

Figure 49: Executing the merge functional test (excerpt)

As we can see at the figure above, **all of the functional tests are correct**. As has been said, in the future we can easily add other test scenarios (such as special conflicts at beginning or end of files, much more complex conflicts or other cases) and easily execute the functional test before committing each major update to the code.

Just for the record, let us have a look at a situation where the merge algorithm cannot resolve a conflict automatically (if the endpoint and data parameters are the same):

```
***** 3 => 3a,3b *****
Conflicting Endpoint (same in A and B):
http://wwwlab.cs.univie.ac.at/~a1246269/workflow/ws/credentialCheck
.php

Conflicting Data Params:
[username] => data.username
[password] => data.password
[role] => data.userRole
[organisation] => data.userOrganisation

CONFLICT needs to be manually resolved
```

Figure 50: Unresolved conflict during the merge functional test

And finally, let us have a look at a situation where the merged result is wrong (i.e. not as expected) - we intentionally modified the input files, so these two situations occur:

```
***** 4 => 4a,4b *****
NOT as expected

Merge result:
AX1C

Expected result:
A
X1
C
```

Figure 51: Wrong result during the merge functional test

### 4.3 Test Cases

In order to make a clear comparison of our semantic merge algorithm vs a generic algorithm, we will proceed to the biggest part of the evaluation. We want to do a high scale testing of the merging algorithms and see how they perform - in terms of correctly merged files and especially in terms of automatically resolved conflicts.

For the high scale testing of the success rate of the merge algorithms, we want to make use of the available students' instances from CPEE. We have

about 6000 real business process model instances that students have modeled in their courses, so it is both a representative and real-life sample of instances.

The general idea is that we will split each diff (from one significant instance to another) into two halves - and merge the two half-diffs with the original file to see, whether the result is the same as expected (the next significant instance is the expected result).

So the approach of preparing test cases for the evaluation of merge algorithms is as follows:

1. Iterate through the significant instances (with structural diffs) and for each:
2. Take the diff (of original and next file)

Original file (instance 50)	Next file (instance 52)
0 system shows login form to cust..	0 system shows login form to cust..
1 customer sends login data	1 customer sends login data
2 system checks credentials and ..	2 system checks credentials and ..
3 choose	3 choose
4 system ends interaction	4 asdf
5 test	5 system creates session
6 system creates session	6 loop
7 loop	7 .....
8 .....	
Diff	
- (4) system ends interaction	
- (5) test	
+ (6) asdf	

Table 5: Diff of Original and Next File

Using the previously implemented scripts, it is easy for us to determine the diff of two instances. In fact, the defining of significant instances, as well as defining the diffs in them has already been automatically done by the scripts - so we have all the necessary artifacts available.

For the purpose of explaining the process of preparing test cases, we will have take the instances 50 and 52 from student a1246269 as an example. In the above table, we can see a fraction of the instance 50 (left) and instance 52 (right). In this case, instance 50 is the original file and instance 52 is the updated file with certain (structural) changes. For the purpose of easier reading, these changes have been coloured with green (add) and red (remove).

As we can see, lines 4 and 5 were removed, while at the position of 6 (in the original file), a new line was added. In the bottom of the table, we can see the resulting diff file.

3. Split the diff into two separate diffs (odd lines to diff A and even lines to diff B)

Diff		Diff A		Diff B
- (4) system ends interaction	=>	- (4) system ends interaction		
- (5) test	=>			- (5) test
+ (6) asdf	=>	+ (6) asdf		

Table 6: Diff file split into Diff A and Diff B

Having the diff file available, we can split the complete diff into two halves. For this purpose, we go through all the changes in a diff file (i.e. lines) and put the odd lines into diff A file and the even lines into diff B file.

This can be seen at the listed example where the lines 1 and 3 of the diff were assigned to diff A and line 2 was assigned to diff B.

4. Save diff A and diff B to separate files (as well as patched files A and B):

Diff A	Diff B
- (4) system ends interaction	- (5) test
+ (6) asdf	
File 1 A	File 1 B
0 system shows login form to cust..	0 system shows login form to cust..
1 customer sends login data	1 customer sends login data
2 system checks credentials and ..	2 system checks credentials and ..
3 choose	3 choose
(removed)	4 system ends interaction
4 test	(removed)
5 asdf	5 sysstem creates session
6 sysstem creats session	6 loop
7 loop	7 .....

Table 7: Separate diff and patched files for A and B

We will store the diff A and diff B into separate files, as well as the patched files. These files can later be used by the merging algorithms for the evaluation purposes.

5. Merge the Original file with the Diffs A and B - using the semantic and generic merge algorithms:

Original File	Diff A	Diff B
0 system shows login form to ..	- (4) system ends interaction	- (5) test
1 customer sends login data	+ (6) asdf	
2 system checks credentials ..		
3 choose		
4 system ends interaction		
5 test		
6 sysstem creats session		
7 loop		
8 .....		
Merged Result File:		
0 system shows login form to ..		
1 customer sends login data		
2 system checks credentials ..		
3 choose		
(removed)		
(removed)		
+ (4) asdf		
5 sysstem creats session		
6 loop		
7 .....		

Table 8: Original File, Diff A, Diff B and the Merged Result

We can execute the two merging algorithms for each of the significant instances to see the results. The semantic algorithm will take the two original file and two diffs as inputs, whereas the generic algorithm will take the original file and the two changed files as inputs (since it performs the search for diff itself). The following subchapters 4.3.1 and 4.3.2 offer a little more insight into the specific testing of each algorithm.

#### 6. Evaluate the results

And finally by applying the merge algorithms, we can evaluate the results. The main goal is to see how many instances get merged correctly without the need for human intervention and how many have conflicts that can't be automatically resolved. The results of the evaluation can be seen in Chapter 4.4.

It needs to be said that our test case of about 6000 available process instances with each diff split into two halves and merged back again is not at typical everyday use case for the merging algorithms. Since we separate each line from one another (odd and even lines into separate diffs), there is a high chance of getting many conflicts. E.g. every time two or more add operations are at the same spot, they get split into the two diffs and a conflict situation arises (during the merging). Nevertheless, this high amount of conflicts is an excellent test suite to see how well our semantic merge algorithm performs in automatically resolving conflicts.

#### 4.3.1 Testing Semantic Algorithm

First, we will be testing the proposed semantic merging algorithm described in chapter 3.4.4. For this purpose, we developed a script `test_semantic_algorithm`

which performs the important steps described in chapter 4.3. So it not only splits the diff into two separate diffs, but also saves them and the patched files, so that both merging algorithms can be tested.

The usage of this script is shown in Figure ?? . The parameter `student` defines the student, for which the testing should be performed. Parameters `start_id` and `max_instances` can be used to limit the testing to a certain number of instances (e.g. by setting `start_id` to 5 and `max_instances` to 7, the script will only test 7 instances starting from id 5) and the `force_diff` parameter allows you to turn on/off the forced execution of diff - the diff for all instances can be performed by a separate script, so for performance reasons you need not do it again.

#### 4.3.2 Testing Generic Algorithm

The script for testing the generic algorithm has a very similar usage as the `test_semantic_merge` script. The only difference is the missing `force_diff` parameter, since this script does not perform our own diff any more.

As a generic merge algorithm, we use the `diff3` utility from the GNU Diffutils<sup>5</sup>. This is a generic merge algorithm that can merge an original text file with two modified files.

### 4.4 Qualitative Evaluation Based on Test Cases

Based on the test cases prepared in chapter 3.1, we can now perform the qualitative evaluation based on those test cases. We can run the big scale tests by executing the `test_semantic_merge` and `test_generic_merge` scripts. These two scripts give us a lot of data for analysis:

- A csv file for all students, showing the number of these values (both for each student separately and an aggregated sum):
  - Correctly merged instances
  - Instances with exchanged lines
  - Instances with unresolved conflicts
  - Other instances
  - All significant instances  
(changes in structure, endpoint and/or parameter(s))
  - Significant instances with structural changes
  - Number of structural changes (lines changed)
  - Execution time
  - Merging time
- Graphs made from the csv file
- A separate csv file for each student with the exact outcome, number of changes and merging time for each instance

---

<sup>5</sup><http://www.gnu.org/software/diffutils/diffutils.html> (last access: 7.11.2017)

- A number of log files For each student (one log file containing the whole output and separate log files only for the different result types)

Whenever a merged result is not the same as the expected result (i.e. the next significant instance), the log files contain a nice comparison of the merged and expected files with highlighted differences.

The aggregated statistical numbers are as follows:

- 1217 significant instances (those with any changes)
- 661 with non-structural changes (endpoint and/or parameters)
- 556 significant instances with structural changes - so these are the 556 instances we tested for merging
- containing a sum of 4354 structural (line) changes.

Now let us see the results of the evaluation:

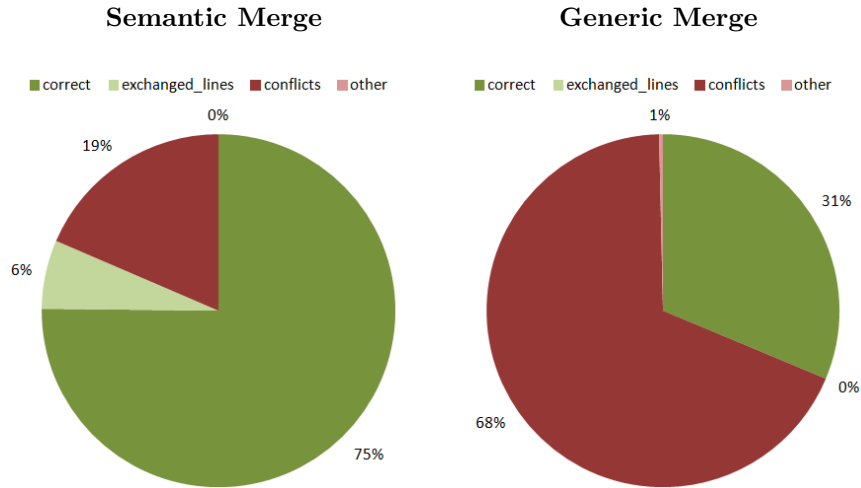


Figure 52: Results of Semantic vs. Generic Merge

The above graph (Figure 52) shows the percentage of successfully merged instances. Correct means the merged result is exactly the same as expected. Exchanged\_lines means that only the order of certain lines (i.e. activities) was exchanged but since it was proven that there is no endpoint or data parameter conflict between those activities, it is also correct. Conflict means an unresolved conflict occurred during the merge, so this is an undesired result. And other means any other results that do not equal the expected result. So all in all, correct and exchanged\_lines are correct, whereas conflict and other are wrong.

We can clearly see that the **semantic merging algorithm has a much higher success rate**. The semantic algorithm can automatically merge  $75+6 = 81\%$  of the test instances, while the generic merging algorithm only merges automatically 31%.

Also, the semantic algorithm only has 19% clearly identified unresolved conflicts (meaning that in those cases the conflicting activities used the same endpoint or the same data parameters and thus could not be automatically resolved)



while the generic algorithm has 68% of unresolved conflicts. This shows that any time there was a conflict, the generic algorithm asks the user to resolve it manually, while the semantic algorithm could decrease the percentage of manual conflict resolutions from 68 to only 19% - more than three times lower. In other words, a human worker needs to intervene more than three times less often and this can save enormous amount of human working time.

The following two graphs show the distribution of the result types among students in a more detailed way:

Also at these two graphs, we can see the big difference in the success rate of the semantic and a generic merging algorithm. All the expertise and use of semantic knowledge that has gone into developing the semantic algorithm has clearly payed off with a significantly higher automation of conflict resolving. With the use of our algorithm, we can significantly reduce the required human intervention in resolving potential conflicts and save an enormous amount of human labour time.

It needs to be added that our test cases are a special scenario where a significantly higher number of conflicts may arise than in every day situations. By the nature that we took entire diffs and split them into half by odd and even lines, whenever there were multiple add operations at the same place, by splitting and merging them, one would naturally get a conflict. In other words, whenever the original diffs contained more than one add operation at the same spot, a conflict arose. However, exactly due to the unconventionally high number of conflicts, it is a really good test case for testing the ability of our merging algorithms to automatically resolve conflicts.

Now let us quickly have a look at samples of the various result types and explain them.

#### 4.4.1 Result Types

There are four different result types in our tests:

- correct
- exchanged\_lines (is also correct)
- conflict (is wrong)
- other (is also wrong)

The correct type is self-explanatory - the merged result equals the expected result. Other is any merged result that does not equal the expected result and does not fall into the defined categories (and we do not need to address this with much focus, since there are only 2 other cases (out of 556) in the generic algorithm). For the exchanged lines and conflict, we will now have a quick look at a sample result.

**4.4.1.1 Exchanged Lines** When the result is almost the same as expected and only certain lines were exchanged, we classify it as `exchanged_lines`. This result is basically still correct, because the merging algorithm made sure that the two conflicting activities are not using the same endpoint or data parameters - otherwise it would give us an unresolved conflict. Since the merging

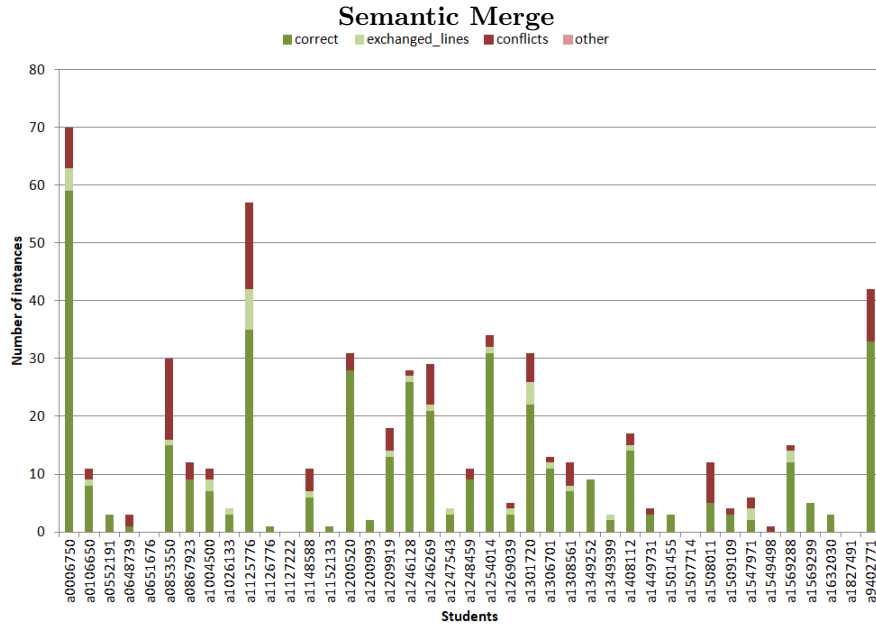


Figure 53: Detailed Results of Semantic Merge

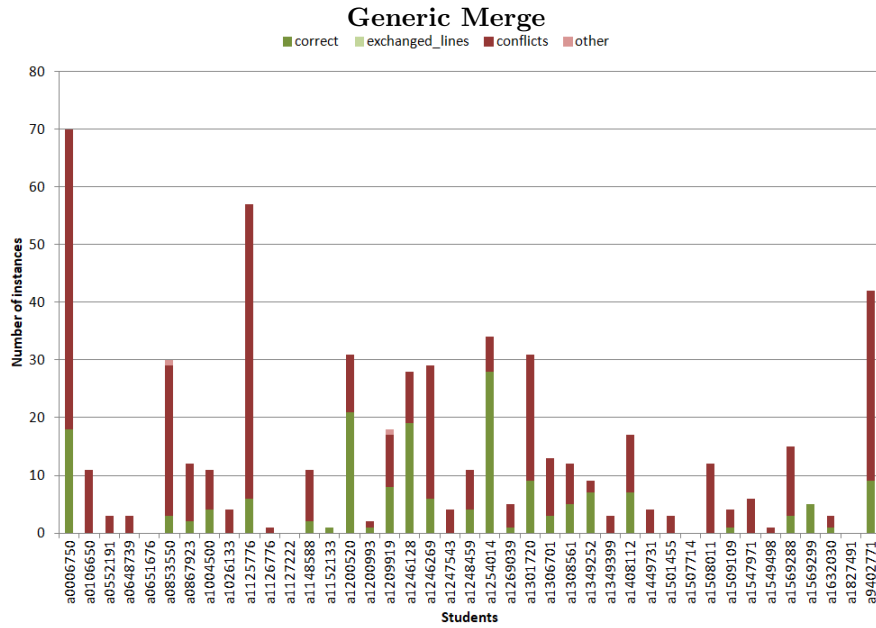


Figure 54: Detailed Results of Generic Merge

algorithm returns a result without a conflict, it means that the activities can be automatically resolved.

The reason why the exchanged lines case occurs is because we split the original diff by odd and even lines - which means that when re-merging the diff,

we may take the changes in the opposite order. It depends on pure chance. Let us see an example:

```
Diff A:
- (8) system adds product to the shopping car..
- (14) customer enters shipping adress and de..
+ (15) user decides if he wants an alternativ..

Diff B:
+ (9) customer adds product to the shopping c..
+ (15) customer enters shipping address

Merged result                                Original next version
.....
12 customer enters payment data                12 customer enters payment data
13 system shows adress form to customer        13 system shows adress form to customer
14 * user decides if he wants an alte..        14 * customer enters shipping adress
15 * customer enters shipping adress            15 * user decides if he wants an alte..
16 choose                                      16 choose
.....
```

Figure 55: Sample exchanged lines  
(Student a1246269, instances 1 and 16)

As we can see, there are two new activities to be added at line 15 (*user decides if he wants an alternativ...* and *customer enters shipping adress*). Whenever there is such a conflict (and can be automatically resolved), the algorithm places the activity from diff A first and from diff B second - so we get the manifested result. The original result had the same lines in reverse order, so the merged result does not equal the expected result. However, since the activities can be auto-merged, it means that the process will run correctly and can thus be classified as correct.

**4.4.1.2 Conflict** When the merge algorithm cannot resolve a conflict, it looks the following way:

```
Diff A:
- (4) system creates session
+ (5) sysstem creates session
- (9) customer checks if all needed products are saved
...

Diff B:
+ (5) system ends interaction
- (8) system adds product to the shopping cart

Merged result                                Original next version
.....
3 choose                                      - choose
4 * ## A                                     - system ends interaction
5 system creates session                     - sysstem creates session
6 * ## B                                     - loop
7 * system ends interaction                  - system shows product list(s)
8 * ## END                                   - customer selects product
9 * loop                                      - customer adds product to the shop..
.....
```

Figure 56: Sample unresolved conflict  
(Student a1246269, instances 62 and 63)

We can see that there are two activities to be added at line 5 (*system creates session* and *system ends interaction*). The conflict could not be automatically resolved, so the semantic algorithm returns an unresolved conflict. The situation with the generic algorithm would be very similar, only the syntax of the output format is slightly different.

## 4.5 Quantitative Evaluation

The most important evaluation is the qualitative evaluation that was presented in the previous subchapter. The quantitative performance evaluation is not too critical, since the merging of business process model instances does not occur too often, however we still want to make a short overview of the execution time of the merge algorithm.

During the qualitative testing, we recorded the merging times for all tested instances into a csv file. So now it is very easy to aggregate them and produce a mean value:

		Semantic algorithm		Generic algorithm	
Student	Instances	Merge time	Average	Merge time	Average
a0006750	70	0,09137	0,001305286	27,12083	0,387440429
a0106650	11	0,01831	0,001664545	3,9318	0,357436364
a0552191	3	0,00266	0,000886667	1,026	0,342
a0648739	3	0,01457	0,004856667	1,34055	0,44685
.....					
.....					
		<b>Average:</b>		<b>Average:</b>	
		<b>0,001777857s</b>		<b>0,360531873s</b>	

Table 9: Execution times for the semantic and generic merge algorithms

As we can see, the average execution time of the semantic merging is 0,001777857 seconds, while that of the generic algorithm is 0,360531873 seconds. This is the average merge time of one instance.

However, it needs to be said that the two times are not equal in what they represent. For the semantic algorithm, we are able to measure purely the time of the merging process (so once we already have the original file and two diffs available and we pass them all over to the merging function), whereas for the generic algorithm, we can only measure the entire execution time of the script - this means the script opens the original file and two updated files, finds the diffs for both files and only then merges them. In other words, the generic time also includes opening the three files and performing two diffs, while the semantic time doesn't.

Despite these facts, we can state that the execution time of the semantic algorithm is definitely satisfying and would in no way pose a problem in a real-life application of the algorithm. The execution times are by no means too high and impracticable and despite not having a precise comparison to the generic algorithm, the times of the semantic algorithm are about 0,0018 seconds, which is excellent.

## 5 Discussion and Outlook

### 5.1 Contribution

The main contribution of this thesis is the conceptual design and prototypical implementation of a semantic BPM merging algorithm, which significantly improves automated conflict resolving. In the test cases, it achieved 81% of correctly merged cases and only 19% unresolved conflicts, compared to a generic merging algorithm, with 31% correct cases and 68% unresolved conflicts.

Along with the semantic merge algorithm, other contributions constitute of the underlying diff algorithm (used by the merge algorithm as an integral part) which performs diffs of BP models (showing not just a diff of the structure, but also of endpoint and parameters) and all the other scripts developed: aside from the script for regrouping the test cases (and extracting only the essential information), a script for looping through all instances and finding the significant ones (containing any diffs) and a script for determining all the diffs between all the significant instances (with the two last scripts producing practical overview graphs of all significant instances and all diffs).

### 5.2 Conclusion

This thesis has shown that a) a semantic BPM merge algorithm can indeed be developed and that b) it is significantly more efficient than a generic test algorithm - in the test cases, it produced 81% correctly merged cases and only 19% unresolved conflicts, compared to a generic merging algorithm, with 31% correct cases and 68% unresolved conflicts.

Business process models, indeed, have a much smaller complexity than generic programming languages and a semantic analysis of process models is therefore possible without extreme effort. Due to the limited number of entities and constraints of a process model, it is possible to analyze the relations and make qualified decisions in the case of a merging conflict. By defining the decision outcomes for various conflict scenarios (and by defining in which cases an automated conflict resolving is possible), it was possible to drastically increase the number of automatically resolved conflicts and to drastically reduce the number of conflicts that still need human decision making.

This can automate a large extent of process merging and save enormous amounts of human labour.

### 5.3 Integration with Existing BPM Environment

For the moment being, the proposed and implemented merge algorithm performs all that is required - it performs a semantic BPM merge with a high amount of automation and so, it achieves its goal. Let us shortly talk about possible future extensions.

The artifacts produced by the merge algorithm at the moment are simple simple text or json files (and a number of graphs in png format, as well as a number of statistical data in csv format). This is absolutely fine for the purpose of demonstrating the principles and the improvements behind the algorithm. However, in real use, the artifacts produced by the merge algorithm should

be expanded to those compatible with standardized formats and widely used programs.

The algorithm would fulfill its real mission by being integrated into the existing BPM ecosystem. It would be possible to use the algorithm in connection with popular BPM programs/platforms, which is where people would actually make use of it. In order to integrate it with the existing ecosystem, the first step is to make sure that the inputs and outputs for the algorithm are compatible.

The input for the semantic merge algorithm is the hierarchical file format of a tree-based modeling notation. As for the outputs, it was said that the current output of the algorithm are simple text and json files. This can be easily expanded to make it compatible with the popular standards and programs.

First, the output can definitely be modified so that it can be integrated back into the Refined Process Structure Tree format. Also, since RPST is an imperative process modeling language, it is easily possible to adapt the output to other imperative modeling languages - such as BPMN.

It is also possible to extend the exporting possibilities of the algorithm to accommodate file formats for any desired standardized and program-specific formats. E.g. standardized XML-based or JSON-based file formats could easily be added. Also file formats for any desired programs could be exported (e.g. Signavio or MS Visio).

As a second stage, an even tighter integration with existing BPM platforms could be done by making the algorithm an integral part of the BPM software by directly executing it. In other words, the algorithm would be coded as an executable library integrated into the software and being executed every time needed.

## List of Figures

1	Sample ConDec Process (Purchasing a Book) [5]	14
2	Sample BPMN process - Building a house	16
3	BPMN Sample Flow Objects	16
4	Sample EPC process - Building a house	18
5	EPC Building Blocks	19
6	Sample Petri network - Building a house	19
7	Sample CPEE process (based on RPST) - Building a house	21
8	Decomposition of a sample BPMN into blocks [17] and the resulting RPST	21
9	Example of comparing two files [34]	24
10	Common subsequences [32]	25
11	Matrix of edit distances [34]	25
12	Pattern 1 - Insert Process Fragment [49]	32
13	CPEE Demo Step 1: Selecting an instance	35
14	CPEE Demo Step 2: Creating a sample instance	36
15	CPEE sample instances	36
16	CPEE sample instances (inside a folder)	37
17	Sample XML instance file from CPEE	38
18	Processed Instance File (Excerpt)	40
19	Executing the group instances script	41
20	Folder structure created by the <i>group instances</i> script	42
21	Instance files in a student's folder	42
22	Executing the <i>find-significant-instances</i> script	44
23	All significant instances json file	45
24	Graph of significant instances and changes	46
25	Structure of a Sample XML Instance File From CPEE	48
26	Simplified JSON Version of the Sample Instance File	49
27	Results of the simple diff algorithm	50
28	Optimal results (as would be delivered by an LCS-based algorithm)	50
29	LCS for two sample files	51
30	LCS-based diff results for (abcdefgh) vs (bcxdyefg)	51
31	LCS-based diff results for (axxbxx) vs (bxxaxxbxx)	52
32	Graphical export of structural diff	52
33	Fraction of sample structural diff	53
34	Graphical export of complete diff results	54
35	Manual vs Automatic Merge	60
36	Parameters and data parameters of a sample activity	61
37	Conflict scenario 1	62
38	Conflict scenario 2	63
39	Conflict scenario 3	64
40	Conflict scenario 4	65
41	Conflict scenario 5	66
42	Conflict scenario 6	67
43	Conflict scenario 7	68
44	Conflict scenario 8	69
45	Conflict scenario 9	70
46	Conflict scenario 10	71

47	Pseudocode of the merging algorithm . . . . .	72
48	Pseudocode - conflict resolving . . . . .	74
49	Executing the merge functional test (excerpt) . . . . .	80
50	Unresolved conflict during the merge functional test . . . . .	81
51	Wrong result during the merge functional test . . . . .	81
52	Results of Semantic vs. Generic Merge . . . . .	86
53	Detailed Results of Semantic Merge . . . . .	88
54	Detailed Results of Generic Merge . . . . .	88
55	Sample exchanged lines . . . . .	89
56	Sample unresolved conflict . . . . .	90



## List of Tables

1	Sample diff output . . . . .	26
2	Context and Unified format of the sample diff output . . . . .	27
3	Sample file versions . . . . .	29
5	Diff of Original and Next File . . . . .	82
6	Diff file split into Diff A and Diff B . . . . .	83
7	Separate diff and patched files for A and B . . . . .	83
8	Original File, Diff A, Diff B and the Merged Result . . . . .	84
9	Execution times for the semantic and generic merge algorithms .	90

## References

- [1] T. Mens, “A state-of-the-art survey on software merging,” *IEEE transactions on software engineering*, vol. 28, no. 5, pp. 449–462, 2002.
- [2] S. Goedertier, J. Vanthienen, and F. Caron, “Declarative business process modelling: principles and modelling languages,” *Enterprise Information Systems*, vol. 9, no. 2, pp. 161–185, 2015.
- [3] D. Fahland, D. Luebke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal, “Declarative versus imperative process modeling languages: The issue of understandability,” *Enterprise, Business-Process and Information Systems Modeling: 10th International Workshop, BPMDS 2009, and 14th International Conference, EMMSAD 2009, held at CAiSE 2009, Amsterdam, The Netherlands, June 8-9, 2009*.
- [4] M. Pesic, “Constraint-based workflow management systems: Shifting control to users,” *PhD thesis, Eindhoven University of Technology*, 2008.
- [5] M. Pesic and W. M. P. van der Aalst, *A Declarative Approach for Flexible Business Processes Management*, pp. 169–180. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.
- [6] C. A. Petri, “Concepts of net theory,” in *MFCS*, vol. 73, pp. 137–146, 1973.
- [7] A. Holt, “A mathematical model of continuous discrete behavior,” *Massachusetts Computer Associates, Inc.*, Nov, 1980.
- [8] S. Goedertier, R. Haesen, and J. Vanthienen, “Em-bra2ce v0. 1: A vocabulary and execution model for declarative business process modeling (pp. 0–74),” *DTEW-KBL0728*, 2007.
- [9] M. Reichert and P. Dadam, “Adeptflex—supporting dynamic changes of workflows without losing control,” *Journal of Intelligent Information Systems*, vol. 10, no. 2, pp. 93–129, 1998.
- [10] P. Dadam and M. Reichert, “The adept project: a decade of research and development for robust and flexible process support,” *Computer Science-Research and Development*, vol. 23, no. 2, pp. 81–97, 2009.
- [11] W. M. Van der Aalst, M. Weske, and D. Grünbauer, “Case handling: a new paradigm for business process support,” *Data & Knowledge Engineering*, vol. 53, no. 2, pp. 129–162, 2005.
- [12] S. W. Sadiq, M. E. Orlowska, and W. Sadiq, “Specification and validation of process constraints for flexible workflows,” *Information Systems*, vol. 30, no. 5, pp. 349–378, 2005.
- [13] S. Goedertier and J. Vanthienen, “Designing compliant business processes with obligations and permissions,” in *International Conference on Business Process Management*, pp. 5–14, Springer, 2006.
- [14] K. Bhattacharya, C. Gerede, R. Hull, R. Liu, and J. Su, “Towards formal analysis of artifact-centric business process models,” in *International Conference on Business Process Management*, pp. 288–304, Springer, 2007.

- [15] R. Lu, S. Sadiq, and G. Governatori, "On managing business processes variants," *Data & Knowledge Engineering*, vol. 68, no. 7, pp. 642–664, 2009.
- [16] S. A. White, "Introduction to bpmn," *Business Process Trends*, vol. 2, 2004. available at: [www.bptrends.com](http://www.bptrends.com).
- [17] J. Vanhatalo, H. Völzer, and J. Koehler, "The refined process structure tree," *Data & Knowledge Engineering*, vol. 68, no. 9, pp. 793–818, 2009.
- [18] B. P. M. OMG, "Notation (bpmn) 2.0," *Object Management Group: Needham, MA*, vol. 2494, p. 34, 2011.
- [19] G. Keller, M. Nüttgens, and A. Scheer, "Semantische prozessmodellierung auf der grundlage ereignisgesteuerter prozessketten (epk). 1992," *Institut für Wirtschaftsinformatik, Universität des Saarlandes: Saarbrücken*, 1992.
- [20] OMG, "Uml 2.0 superstructure specification," *OMG, Needham*, 2004. available at: [www.omg.org](http://www.omg.org).
- [21] J. C. Recker and J. Mendling, "On the translation between bpmn and bpel: Conceptual mismatch between process modeling languages," in *The 18th International Conference on Advanced Information Systems Engineering. Proceedings of Workshops and Doctoral Consortium*, pp. 521–532, Namur University Press, 2006.
- [22] P. Wohed, W. M. van der Aalst, M. Dumas, A. H. ter Hofstede, and N. Russell, "On the suitability of bpmn for business process modelling," *Business Process Management*, vol. 4102, pp. 161–176, 2006.
- [23] A.-W. Scheer, O. Thomas, and O. Adam, "Process modeling using event-driven process chains," *Process-aware information systems*, pp. 119–146, 2005.
- [24] W. M. Van der Aalst, "Formalization and verification of event-driven process chains," *Information and Software technology*, vol. 41, no. 10, pp. 639–650, 1999.
- [25] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.
- [26] C. A. Petri, "Kommunikation mit automaten," *Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM*, no. 3, 1962.
- [27] W. Reisig, *Petri nets: an introduction*, vol. 4. Springer Science & Business Media, 2012.
- [28] J. Vanhatalo, H. Völzer, and F. Leymann, "Faster and more focused control-flow analysis for business process models through sese decomposition," *Service-Oriented Computing-ICSOC 2007*, pp. 43–55, 2007.
- [29] J. Küster, C. Gerth, A. Förster, and G. Engels, "Detecting and resolving process model differences in the absence of a change log," *Business Process Management*, pp. 244–260, 2008.

- [30] R. E. Tarjan and J. Valdes, “Prime subprogram parsing of a program,” in *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 95–105, ACM, 1980.
- [31] A. Polyvyanyy, J. Vanhatalo, and H. Völzer, “Simplified computation and generalization of the refined process structure tree,” in *International Workshop on Web Services and Formal Methods*, pp. 25–41, Springer, 2010.
- [32] J. W. Hunt and M. McIlroy, *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.
- [33] W. F. Tichy, “The string-to-string correction problem with block moves,” *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 309–321, 1984.
- [34] W. Miller and E. W. Myers, “A file comparison program,” *Software: Practice and Experience*, vol. 15, no. 11, pp. 1025–1040, 1985.
- [35] E. W. Myers, “An o (nd) difference algorithm and its variations,” *Algorithmica*, vol. 1, no. 1, pp. 251–266, 1986.
- [36] E. Ukkonen, “Algorithms for approximate string matching,” *Information and control*, vol. 64, no. 1-3, pp. 100–118, 1985.
- [37] F. Lanubile, C. Ebert, R. Prikladnicki, and A. Vizcaíno, “Collaboration tools for global software engineering,” *IEEE software*, vol. 27, no. 2, 2010.
- [38] S. Otte, “Version control systems,” *Computer Systems and Telematics, Institute of Computer Science, Freie Universität, Berlin, Germany*, 2009.
- [39] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*. ” O’Reilly Media, Inc.”, 2012.
- [40] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, “How do centralized and distributed version control systems impact software changes?,” in *Proceedings of the 36th International Conference on Software Engineering*, pp. 322–333, ACM, 2014.
- [41] D. E. Perry, H. P. Siy, and L. G. Votta, “Parallel changes in large-scale software development: an observational case study,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 3, pp. 308–337, 2001.
- [42] W. F. Tichy, “Rcs—a system for version control,” *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.
- [43] E. Adams, W. Gramlich, S. S. Muchnick, and S. Tirfing, “Sunpro engineering a practical program development environment,” in *Advanced Programming Environments*, pp. 86–96, Springer, 1986.
- [44] D. B. Leblang and R. P. Chase Jr, “Computer-aided software engineering in a distributed workstation environment,” in *ACM Sigplan Notices*, vol. 19, pp. 104–112, ACM, 1984.

- [45] B. Berliner *et al.*, “Cvs ii: Parallelizing software development,” in *Proceedings of the USENIX Winter 1990 Technical Conference*, vol. 341, p. 352, 1990.
- [46] M. S. Feather, “Detecting interference when merging specification evolutions,” in *ACM SIGSOFT Software Engineering Notes*, vol. 14, pp. 169–176, ACM, 1989.
- [47] E. Lippe and N. Van Oosterom, “Operation-based merging,” in *ACM SIGSOFT Software Engineering Notes*, vol. 17, pp. 78–87, ACM, 1992.
- [48] M. Reichert, P. Dadam, S. Rinderle-Ma, M. Jurisch, U. Kreher, and K. Göser, “Architectural principles and components of adaptive process management technology,” 2009.
- [49] B. Weber, M. Reichert, and S. Rinderle-Ma, “Change patterns and change support features—enhancing flexibility in process-aware information systems,” *Data & knowledge engineering*, vol. 66, no. 3, pp. 438–466, 2008.
- [50] C. Li, M. Reichert, and A. Wombacher, “Discovering reference process models by mining process variants,” in *Web Services, 2008. ICWS’08. IEEE International Conference on*, pp. 45–53, IEEE, 2008.
- [51] M. Reichert, P. Dadam, and T. Bauer, “Dealing with forward and backward jumps in workflow management systems,” *Software and Systems Modeling*, vol. 2, no. 1, pp. 37–58, 2003.
- [52] S. Rinderle, M. Reichert, and P. Dadam, “Correctness criteria for dynamic changes in workflow systems—a survey,” *Data & Knowledge Engineering*, vol. 50, no. 1, pp. 9–34, 2004.
- [53] B. Weber, S. Rinderle, and M. Reichert, “Change patterns and change support features in process-aware information systems,” in *International Conference on Advanced Information Systems Engineering*, pp. 574–588, Springer, 2007.
- [54] M. Poppendieck and T. Poppendieck, *Implementing lean software development: From concept to cash*. Pearson Education, 2007.
- [55] W. M. Van Der Aalst and T. Basten, “Inheritance of workflows: an approach to tackling problems related to change,” *Theoretical Computer Science*, vol. 270, no. 1-2, pp. 125–203, 2002.
- [56] C. Ellis, K. Keddara, and G. Rozenberg, “Dynamic change within workflow systems,” in *Proceedings of conference on Organizational computing systems*, pp. 10–21, ACM, 1995.
- [57] M. Minor, A. Tartakovski, and D. Schmalen, “Agile workflow technology and case-based change reuse for long-term processes,” *International Journal of Intelligent Information Technologies (IJIT)*, vol. 4, no. 1, pp. 80–98, 2008.
- [58] M. Weske, “Formal foundation and conceptual design of dynamic adaptations in a workflow management system,” in *hicss*, p. 7051, IEEE, 2001.

- [59] B. Weber, M. Reichert, S. Rinderle-Ma, and W. Wild, “Providing integrated life cycle support in process-aware information systems,” *International Journal of Cooperative Information Systems*, vol. 18, no. 01, pp. 115–165, 2009.
- [60] C. W. Gunther, S. Rinderle-Ma, M. Reichert, W. M. Van Der Aalst, and J. Recker, “Using process mining to learn from process changes in evolutionary systems,” *International Journal of Business Process Integration and Management*, vol. 3, no. 1, pp. 61–78, 2008.
- [61] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible markup language (xml).,” *World Wide Web Journal*, vol. 2, no. 4, pp. 27–66, 1997.