Submitted by
Falk Viktor Görner

Submitted at
Department for Business Informatics - Data & Knowledge Engineering

Supervisor
Assoz.-Prof. Mag. Dr. Christoph Schütz

Co-Supervisor
Simon Staudinger

February 2023

Design and Implementation of a Module for the Alpha Algorithm in the Learning Platform eTutor++

Master's Thesis
to obtain the academic degree of
Master of Science
in the Master's Program
Economic and Business Analytics

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at

Abstract

The expansion of distance learning at universities, accelerated and amplified by the COVID 19 pandemic, has made the provision of e-learning platforms an integral component of teaching. In order to support this development, this master's thesis introduces a new learning module. To better understand the concept of process mining, a learning module for the alpha algorithm was designed and implemented for the e-learning platform eTutor++. Students can use the learning module to enter the required steps of the alpha algorithm into a user interface based on a provided event log and receive automatically generated feedback and grading on their input. The learning module is integrated into the already existing eTutor++ framework and comprises the algorithm implementation, student submission evaluation logic, and back and front end implementation. In order to give students tasks, event logs must be generated to which the algorithm can be applied. This is accomplished by employing a process log generator that has been implemented and adapted specifically for this reason.

Contents

List of Figures

List of Tables

Listings

1

Introduction

"Tell me and I'll forget. Teach me and I remember. Let me do it and I learn."

—Benjamin Franklin

This quote beautifully expresses that the imparting of knowledge and new concepts by faculty members is only the first step in the process of understanding. The finest learning outcomes are ultimately produced by independent, and self-directed learning.
Process mining is becoming increasingly important in both economic and political institutions, given the importance of understanding the processes that define an institution and affect associated decisions. However, it is also a topic that students in the associated courses are yet not necessarily exposed to in depth and thus do not have much to do with. As a result, access and a fundamental understanding of how to apply process mining may be lacking.
Learning platforms can support students in their learning efforts and raise interest in the topic of process mining. The eTutor++ is such a learning platform, which provides interactive modules for students to work on exercises for different topics and to receive feedback, but currently lacks support for process mining.

The goal of the thesis is to integrate a learning module into the eTutor++ platform to provide a first introduction to the topic of process mining. This module should allow students to apply one of the basic algorithms from process mining.
In the context of this master's thesis, a learning module for the alpha algorithm for the learning platform eTutor++ is designed and implemented to support students' autonomous learning on process mining topics. Teachers can assign tasks to their students through the learning module. These tasks are made up of an event log, which varies in complexity and size, based on some configuration. The student then applies the alpha algorithm and enters the according solution steps that are based on the event log in an input mask. Furthermore, students have the option of receiving multiple levels of feedback for their entered solution. The tasks are offered either in German or English to ensure that this module is accessible to as many students as feasible. Because of the evolution and growing relevance of dealing with processes, this learning module provides an excellent opportunity to tackle the topic of process mining on the one hand while also deepening the theoretical information taught at the university course on the other.
The goal from creating and deploying the learning module is to give students the opportunity to work on as many learning tasks as feasible. This is enabled via a process log generator, which is able

to generate a series of randomly created processes and to produce event logs by simulating these random processes. Teachers can modify the complexity of the tasks and allocate them to students based on the learning objectives. This and the various feedback levels for each exercise contribute to the best possible learning outcome. To accomplish this goal, the project is composed of four distinct components that interlock and rely on one another and are written in the programming language Java. The first component includes the implementation of the algorithm itself. The second component is the process log generator, which is implemented and customized to the overall goal's requirement. The third component is the implementation of the related back end for the submission analysis. Based on that, the platform module is built, which also includes a distinct back end and the actual front end. The project is built with JetBrains' IntelliJ IDEA and employs Spring and Angular.

Following this introduction, the second chapter outlines the background of this thesis. The concept 'process mining' is defined, as are the 'process log generator' and its features, which are required for this project. Furthermore, the definition of the alpha algorithm and a brief description of the eTutor++ is given. Following that, in Chapter 3 a brief review of related work is provided, including a comparison with other e-learning platforms. The dispatcher implementation, including the algorithm and the process log generation module, is explained in detail in Chapter 4. The platform modules' implementation is then defined in Chapter 5, along with its back end and front end. In Chapter 6, the work's conclusion and impact are discussed. The theoretical description is then illustrated with a concrete example and corresponding figures in Appendix B.

2

Background

In this chapter the background of the thesis is represented and explained. It should be noted, however, that the literature review is done in the context of this work. Additional information can be obtained from the extensive available literature.

2.1 Process Mining

This section will explain the concept and theory of process mining in order to situate this work within a theoretical framework. In terms of achieving this, an explanation of process mining, event logs and Petri nets is provided.

The field of process mining (PM) is a practice that evolved from the area of data mining (Tiwari et al., 2008). In contrast to data mining methods, that are typically data-focused and mainly used to analyse a single step in an end-to-end process, PM is process-focused (van der Aalst, 2012). It thus serves "as a bridge between data mining and business process modeling" (van der Aalst, 2012, p.1) and is regarded as a new type of business analytics (Zerbino et al., 2021). As the authors van der Aalst and Weijters, 2004 state, process mining is a "method of distilling a structured process description from a set of real executions" (p.232). Business process mining techniques utilize event data as generated by information systems (Reijers, 2021) by bringing together machine learning and business process management (Zerbino et al., 2021) and are particularly useful for analysing business process event logs and to keep track of business processes (van der Aalst & Weijters, 2004). These techniques are becoming increasingly popular among businesses because of the possibility to gain a wide range of insights (Reijers, 2021). It allows businesses to understand more about their operations and how they function in practice (Tiwari et al., 2008) and enables them to generate process improvement ideas (van der Aalst, 2012).

The three most common uses for process mining techniques are as follows: discovery, conformance and enhancement (van der Aalst, 2012). The difficulty of identifying a process model from a collection of process cases is addressed by the process mining type discovery (Tiwari et al., 2008) with the goal to extract the best process model from the corresponding event logs (van der Aalst & Weijters, 2004) in order to describe the set of process instances without using any previous knowledge (van der Aalst, 2012). It is also the most widely used process mining technique (Zerbino et al., 2021). As a result, the category also includes some practical use cases that place a strong emphasis on the 'control-flow perspective' of a given process (Ailenei et al., 2012). A frequent

use case is to discover the structure of a process without prior knowledge about this process or a known process model (Ailenei et al., 2012). It is followed by the application of analysing the most common path through a process most of the cases take and the practice of analysing the likelihoods of choosing one way to proceed over another following a process decision point (Ailenei et al., 2012).
Conformance testing, the second application, contrasts a present process model with the defined process model of the current process event log (van der Aalst, 2012). Thereby the goal is to determine whether the process has the desired behaviour in practice (Ailenei et al., 2012). The techniques employed detect and quantify the discrepancies between the model and the log (vom Brocke et al., 2021). This category has also some use cases in practice. One would be to determine deviations from a standard path in the process to identify process outliers (Ailenei et al., 2012).
The third use is enhancement, which involves extending and enhancing an presently used process model (van der Aalst, 2012) based on the event log's supplementary information (Zerbino et al., 2021). It is worth noting that these three applications have not gotten the same amount of research in the literature, but 'discovery' in particular has received the most interest (Zerbino et al., 2021). Since the second and third types of process mining are not part of this work, it is referred to the numerous existing literature.

The essential basis of process mining is an event log, since it is the input of any process discovery algorithm. It is composed of a series of traces, each of which contains the sequence of activities/events associated to a specific case (Augusto et al., 2019). An event is the most fundamental component of a business process (Zerbino et al., 2021). A case in turn stands for the execution of a specific process instance. The concept of an event log is illustrated with the example in Table 2.1. The information of Table 2.1 shows that the log is made up of five cases, each of which has an activity 'A' that starts and an activity 'D' that concludes the case. Case one's equivalent trace is 'A,B,C,D' which is also case three's corresponding trace. The traces in case two and four are 'A,C,B,D'. Only three activities with the appropriate trace 'A,E,D' are carried out for case five. A single character represents an activity, which can have multiple attributes such as time stamps or data attributes (van der Aalst, 2012). These attributes are used for different kinds of process mining analyses, for example bottleneck analysis or organizational mining (van der Aalst, 2012). For the purpose of this work, there is no need for additional attributes to the activities, hence they are also not discussed in detail.

As indicated earlier, the goal of the process mining type discovery is to generate process models by employing specific algorithms. In this work the alpha-algorithm is used. It takes an event log as input and returns a Petri net structure as output to visualize the process model. This is not only done on the basis of performance data, e.g. the average lead time calculated from the corresponding time stamps, but process mining also addresses the causal relationships that exist between activities (van der Aalst & Weijters, 2004). The traditional Petri net has two types of nodes called places (shown as circles) and transitions (shown as rectangles) and is a directed bipartite graph (van der Aalst, 1997). Transitions indicate activities in a certain process in the context of process mining. The nodes are connected by directed arcs, which means that places might be connected to transitions and vice versa. Two nodes with the same kind cannot be connected to one another (van der Aalst, 1997). A Petri net is a very common modelling approach

Table 2.1: Example event log from van der Aalst et al., 2004

case identifier	task identifier
case 1	task A
case 2	task A
case 3	task A
case 3	task B
case 1	task B
case 1	task C
case 2	task C
case 4	task A
case 2	task B
case 2	task D
case 5	task A
case 4	task C
case 1	task D
case 3	task C
case 3	task D
case 4	task B
case 5	task E
case 5	task D
case 4	task D

(Tiwari et al., 2008) used to represent the process instances in the context of the alpha algorithm. Places can contain so called tokens (black dot), and the status of the Petri net is "the distribution of tokens over places" (van der Aalst, 1997, p.3). The state of the net can be changed by the transitions in the net, based on the so called 'firing rule': It is stated that a transition is enabled and capable of firing if all of its input places — those having a directed arc leading from it to the transition — contain "at least one token" (van der Aalst, 1997, p.3). If this transition fires, it "consumes one token from each input place [...] and produces one token for each output place" (van der Aalst, 1997, p.3). The following example, as shown in Figure 2.1 is intended to reinforce the concept of Petri nets.

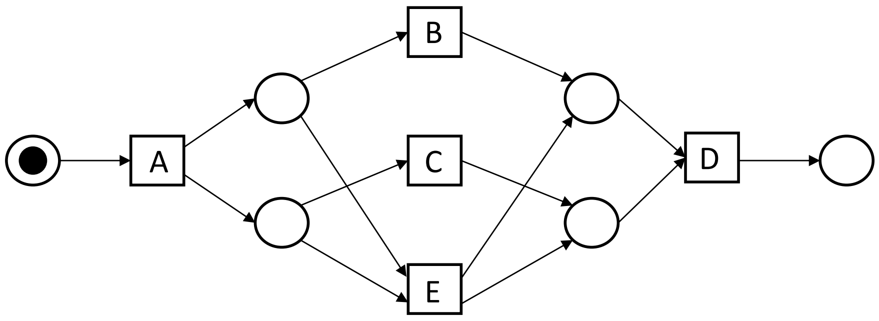


Figure 2.1: Example Petri net (Own representation derived from van der Aalst et al., 2004)

A simple illustration explains the firing rule: In the present state of Figure 2.1, a token can be found before transition 'A'. This is the source place. As a consequence, transition 'A' is enabled because its input place includes one token. As a result, 'A' can fire, consuming the token from the start place while also producing two new tokens for the subsequent places. Three additional transitions, denoted by the letters 'B', 'C' and 'E' are thereby enabled and there is an option to execute transition 'B' and 'C' concurrently or just transition 'E'. It is important to note, that transition 'D' is only enabled, if both of its input places contain at least one token. The place following transition 'D' is called sink place. Furthermore, it is evident that the Petri net in Figure 2.1 can reproduce all traces from the example log in Table 2.1.

2.2 Process Log Generator

As already stated in Section 2.1, the input of the alpha algorithm is an event log. Considering that one of the goals of this thesis is to provide as many learning examples as possible for students in order to practice the process mining concept of the alpha algorithm, business processes and their execution logs are required. There is also the requirement to enable event logs of varied complexity. To achieve these needs, this work emphasizes a tool known as 'Processes and Logs Generator 2' (PLG2) developed by the author of Burattin, 2016. Because just a fraction of this tool is used, this section's purpose is to describe and show its core functioning to the reader in the context of this work. Please refer to the cited literature for further and more extensive information on this instrument and its supplementary functionalities (e.g. data and time perspective, noise, evolving process models, simulation).

Since an event log is the end result of a process simulation, one has to generate a random process first. These process models' underlying structure is based on Business Process Modelling and Notation (BPMN) modelling language which will be briefly explained. A business process (BP) is a collection of some connected actions or tasks that are completed in a certain order to achieve a company objective (Chinosi & Trombetta, 2012). The process of describing an enterprise's processes is known as 'business process modelling', which enables analysis and improvement of the current ('as is') process (Chinosi & Trombetta, 2012). The graphical representation of these business processes is done using the graphical notation BPMN. This notation includes five graphical elements for creating business process diagrams: 'Flow Objects', 'Connecting Objects', 'Swimlanes', 'Artifacts' and 'Data' (Chinosi & Trombetta, 2012). Since only the components 'Flow Objects' and 'Connecting Objects' are relevant for this work, the reader is referred to the existing literature for more information on the other components. Flow objects are events, gateways (e.g. XOR, AND) and activities. They describe all of the actions, possibly occurring inside a business process to determine its behaviour (Chinosi & Trombetta, 2012). Connecting objects link multiple objects together and include 'Sequence Flow', 'Message Flow' and 'Association' (Chinosi & Trombetta, 2012), however only sequence flows are of interest for this work. In essence, a process is a combination of components (Burattin, 2016).

In Burattin, 2016 the author employs and combines parts of the workflow control-flow patterns (WCFP) from van der Aalst et al., 2003 to generate a random process. van der Aalst et al., 2003 describe patterns as "conditions that should hold for the pattern to be applicable; examples of business situations" (p.7) and those address business requirements. Six WCFP are used for the PLG2 tool: The first pattern is 'Sequence', which denotes a sequential connection between two activities (van der Aalst et al., 2003). The second and third patterns, known as 'Parallel split' and 'Synchronization' are closely linked to "parallel routing" (van der Aalst et al., 2003, p.10).

This implies the execution of workflow threads (AND- split operator) and hence actions in parallel. Synchronization occurs when many parallel subprocesses merge into a single thread (AND- join operator) (van der Aalst et al., 2003), and the synchronized thread is only permitted to proceed after all incoming branches have been finished.
Patterns four and five are used to control for conditional routing. These are known as 'Exclusive choice' and 'Simple merge' patterns, respectively. Pattern four defines a moment in the process where a choice must be made for one of various branches (XOR-split operator) whereas pattern five is a stage in the process "where two or more alternative branches come together without synchronization" (van der Aalst et al., 2003, p.12) (XOR-join operator). The last pattern utilized in the tool is known as 'Structured loop' and allows subprocesses to be executed repeatedly (Burattin, 2016). In order to always produce a distinct complete process, and therefore an unique event log, WCFP's are combined according to some specified sets of probabilistic criteria (Burattin, 2016). The formal explanation of generating a random process and its simulation are discussed in the following segment. This concept is implemented using a Context-Free Grammer (CFG) (Burratin, 2015).

A language's grammatical description involves four crucial components: a limited number of symbols that define the language's character set, known as 'terminals'; a set of finite 'non-terminal symbols' each of which represents a language, which is a collection of character sets; a 'start-symbol' that represents the language; and a limited number of 'productions' that represent a language's recursive definition (John E. Hopcroft, 2011). A production rule is of the form $A \rightarrow \alpha$, where 'A' is known as 'head' and is a non-terminal symbol, '$\rightarrow$' is the production sign and 'α' is known as 'body', which consists of a character set of zero or more terminal and non-terminal symbols (John E. Hopcroft, 2011).
According to Burratin, 2015, a CFG that satisfies the necessary conditions is defined as $G_{Process} = \{V, \sum, R, P\}$. The set of non-terminal symbols (V) and the set of terminals ($\sum$) are specified as follows (Burratin, 2015):

$$V = \{P, G, G', G_{\circlearrowright}, G_{\oplus}, G_{\otimes}, A, A_{act}, A_{do}, D\}$$
$$\sum = \{;, (,), \circlearrowright, \oplus, \otimes, \nearrow, \swarrow, e_{start}, e_{end}, a, b, c, ..., d_1, d_1, d_3, ...\}$$

Figure 2.2: Set of terminal and non-terminal symbols (Own representation derived from Burratin, 2015)

with the following set of production rules (R):

$$\begin{aligned}
&P \rightarrow e_{start}; G; e_{end} \\
&G \rightarrow G' \mid G_{\circlearrowright} \\
&G' \rightarrow A \mid (G; G) \mid (A; G_{\oplus}; A) \mid (A; G_{\otimes}; A) \mid \epsilon \\
&G_{\oplus} \rightarrow G \oplus G \mid G \oplus G_{\oplus} \\
&G_{\otimes} \rightarrow G \otimes G \mid G \otimes G_{\otimes} \\
&G_{\circlearrowright} \rightarrow (G' \circlearrowright G) \\
&A \rightarrow A_{act} \mid A_{do} \\
&A_{do} \rightarrow A_{act} \swarrow^{D} \mid A_{act} \nearrow^{D} \\
&A_{act} \rightarrow a \mid b \mid c \mid \ldots \\
&D \rightarrow d_1 \mid d_2 \mid d_3 \mid \ldots
\end{aligned}$$

Figure 2.3: Set of production rules (Own representation derived from Burratin, 2015)

and 'P' as start-symbol for the CFG.
The first production rule $P \rightarrow e_{start}; G; e_{end}$ defines a process in such a way that it requires a start event (e_{start}), an end event (e_{end}) and a sub-graph G in between. With the following rule a sub-graph (G) is defined either as a 'simple sub-graph' (G') or a 'repetition of a sub-graph' (Burratin, 2015). Once more, there is a rule that regulates how these terms are defined. A single activity (A); or the consecutive realization of two simple sub-graphs (G;G); or the sequence of a single activity, a parallel $(A; G_{\oplus}; A)$ or exclusive $(A; G_{\otimes}; A)$ execution between at least two sub-graphs and again a single activity; or an empty sub-graph (ϵ) are all examples of a simple sub-graph (G')(Burratin, 2015).
A single activity (A) can be run as a basic single activity (A_{act}) or as a single activity with a related data object (A_{do}). Data objects are omitted from the tool functions in this thesis because they are not required. As a result, this rule is not utilized in process generation but just the production rule for A_{act}.

Given that one of the purposes of this project is to generate as many distinct processes and consequently event logs as feasible, each production rule is assigned a probability. This allows to randomly generate processes, based on user-defined parameters. To accomplish this, for each production rule the various possible productions are assigned to a group of probabilities. The user can then assign a probability to each of the possible productions within each group. It is worth noting that the probability of each production in a group adds up to one (Burratin, 2015). In total, the user may define ten parameters to control the random generation of a process (how this is done is discussed in Section 4.2.2). The procedure of generating a process consequently consists of constructing the corresponding derivation tree based on the CFG and its assigned probabilities and to use this tree to infer the result string. Thus, this string is then derived by $G_{Process} = \{V, \sum, R, P\}$ and depicts a process (Burratin, 2015) that can be illustrated graphically (e.g. with a Petri-net).

The PLG2 tool uses some algorithms to simulate the created processes in order to derive the appropriate event logs to the randomly generated processes. The primary simulation algorithm essentially requires a process model as input and a parameter to specify the number of simulated traces (Burratin, 2015). In order to fill the event log, the production of single traces is then repeated until the number of traces reaches its maximum (Burratin, 2015). The technical details of the process simulation are not discussed further because they are outside the scope of this work.

2.3 Alpha Algorithm

The alpha algorithm (AA) is described in this section along with its use, the underlying assumptions and its limitations. It serves as the thesis' central theme.
This algorithm was developed by the authors of van der Aalst et al., 2004 in the context of the 'rediscovery problem'. The problem is about developing a mining algorithm capable of extracting a process model from an event log. However, the purpose is to investigate the class of workflow models for which the derived process model equals the workflow model that created the workflow log utilized as input for the AA (van der Aalst et al., 2004). Thus, the purpose of the AA is to discover a process model (e.g., a workflow net) based on an event log, where a workflow net is a Petri net that represents the "control-flow dimension of a workflow" (van der Aalst et al., 2004,

p.1131). As previously noted, the AA takes as input an event log. Table 2.1 displays the least acceptable data that is assumed to be accessible. In addition, it is expected that no noise exists in the data, that the event log provides 'sufficient' information, and the completeness of the log (van der Aalst et al., 2004). Furthermore, the frequency of traces, the identity of cases and noise are excluded in the cited study, since they have no effect on the theoretical results (van der Aalst et al., 2004). With these restrictions the workflow log from Table 2.1 can be summarized as follows: $W = \{ABCD, ACBD, AED\}$.

The first step in developing a process model from an event log is to examine the log for causal dependencies (van der Aalst et al., 2004). This is accomplished by the use of the following notation (Definition 1), referred to as 'ordering relations' in the work of van der Aalst et al., 2004:

Definition 1 (Log-based ordering relations)**.** Let W be a workflow log over T, i.e., $W \,\epsilon\, \mathcal{P}(T^*)$. Let a,b ϵ T:

- $a >_W b$ iff there is a trace $\sigma = t_1 t_2 t_3 \dots t_{n-1}$ and $i \,\epsilon\, \{1, \dots, n-2\}$ such that $\sigma \,\epsilon\, W$ and $t_i = a$ and $t_{i+1} = b$,
- $a \rightarrow_W b$ iff $a >_W b$ and $b \not>_W a$,
- $a \,||_W\, b$ iff $a >_W b$ and $b >_W a$, and
- $a \,\#_W\, b$ iff $a \not>_W b$ and $b \not>_W a$

where 'T' is the set of tasks, $\sigma \,\epsilon\, T^*$ is a trace and $\mathcal{P}(T^*)$ is the powerset of T^*, i.e., $W \subseteq T^*$ (van der Aalst et al., 2004, p.1132).

Those four relations are calculated based on the corresponding workflow log W. With the first relationship $>_W$ one indicates the order in which the tasks appeared, e.g. B directly follows A (A>B). The second relation $\rightarrow_W$ suggests a (direct) causal relation between two tasks and is derived from the first relation (e.g. $A \rightarrow_W B$). A causal relationship exists between two tasks if task b directly follows task a but task a does not follow task b.
Given that task b directly follows task a and vice versa, the third relation $||_W$ indicates possible parallelism of two tasks. If two tasks never directly follow each other and therefore have no causal relationship or potential parallelism, they are covered by relation four $\#_W$ (van der Aalst et al., 2004). If every task that possibly follow another does so in the log in some trace, the event log is complete (van der Aalst et al., 2004).

Following the computation of the ordering relations based on the log, the steps of the algorithm can now be conducted with the purpose of constructing a Petri net. The alpha algorithm (Definition 2) consists of seven steps, which are explained in detail below.
Step one summarizes all of the separate tasks into a single set of transitions T_W. Because tasks and transitions are so closely related, the phrases are used interchangeably (van der Aalst et al., 2004). The initial T_I and final T_O transitions of a trace are included in the second and third steps, respectively. Relation X_W (step 4) and Y_W (step 5) are constructed since some of the places computed by step six must be merged in the event of XOR splits/ joins (van der Aalst et al., 2004). Task pairs (A,B) are part of relation X_W if each element a of A and each element b of B are causally related. In addition, the condition $\forall_{a1,a2 \epsilon A} a_1 \#_W a_2$ and $\forall_{b1,b2 \epsilon B} b_1 \#_W b_2$ must be

satisfied. It says that the elements of A must never follow each other, the same is true for the elements of B. Because it is obtained from the preceding step by "taking only the largest elements with respect to set inclusion" (van der Aalst et al., 2004, p.1135) the result for relation Y_W is straightforward to compute. In practice, this means that all pairs in relation X_W can be deduced from pairs in relation Y_W. Step six summarizes the places in relation P_W based on the pairs calculated in step five. Additionally, this relation also includes the source i_W and sink o_W places. These places can be seen as the first and last circle in Figure 2.1. The place's input and output transitions are indicated by the subscript of $p_{(A,B)}$ (van der Aalst et al., 2004). The arcs of the Petri net are calculated in the final step of the AA. There is an arc connecting the source place and each starting transition (i_W, t), as well as an arc connecting each final transition and the sink place (t, o_W). In addition, an arc is built between each input transition and its associated place $(a, p_{(A,B)})$, as well as between a place and its corresponding output transitions $(p_{(A,B)}, b)$.

Definition 2 (Mining algorithm α)**.** Let W be a workflow log over T. $\alpha(W)$ is defined as follows:

$$\begin{aligned}
1.\, T_W =& \{t \,\epsilon\, T \mid \exists_{\sigma\epsilon W} t \,\epsilon\, \sigma\},\\
2.\, T_I =& \{t \,\epsilon\, T \mid \exists_{\sigma\epsilon W} t \,=\, first(\sigma)\},\\
3.\, T_O =& \{t \,\epsilon\, T \mid \exists_{\sigma\epsilon W} t \,=\, last(\sigma)\},\\
4.\, X_W =& \{(A, B) \mid A \subseteq T_W \,\wedge\, B \subset T_W\\
& \wedge\, \forall_{a\epsilon A} \forall_{b\epsilon B} a \rightarrow_W b \wedge \forall_{a1,a2\epsilon A} a_1 \#_W a_2\\
& \wedge\, \forall_{b1,b2\epsilon B} b_1 \#_W b_2\},\\
5.\, Y_W =& \{(A, B) \,\epsilon\, X_W \vert \forall_{(A',B')\epsilon X_W} A \subseteq A'\\
& \wedge\, B \subseteq B' \Rightarrow (A, B) = (A', B')\},\\
6.\, P_W =& \{p_{(A,B)} \mid (A, B) \epsilon\, Y_W\} \,\cup\, \{i_W, o_W\},\\
7.\, F_W =& \{(a, p_{(A,B)}) \mid (A, B) \,\epsilon\, Y_W \,\wedge\, a \,\epsilon\, A\}\\
& \cup\, \{(p_{(A,B)}, b) \mid (A, B) \,\epsilon\, Y_W \,\wedge\, b \,\epsilon\, B\}\\
& \cup\, \{(i_W, t) \mid t \,\epsilon\, T_I\} \,\cup\, \{(t, o_W) \mid t \,\epsilon\, T_O\},
\end{aligned}$$

$$8.\, \alpha(W) = (P_W, T_W, F_W)$$

(van der Aalst et al., 2004, p.1135). The AA is capable of discovering a broad and relevant class of workflow processes (known as structured workflow nets) based on the given event log (van der Aalst et al., 2004). However, the alpha algorithm still has limitations.
First, it is not able to rediscover a process with 'short loops', i.e., loops with a length below three transitions (van der Aalst et al., 2004). Second, the algorithm is incapable of detecting implicit locations (no PM algorithm can). Since an implicit place has no effect on the process' behavior, it is not displayed in the event log (van der Aalst et al., 2004). Finally, while the algorithm's application is not restricted exclusively to structured workflow nets, it is not possible to assure that arbitrary sound workflow nets may be rediscovered (van der Aalst et al., 2004).
However, it should be noted that it is irrelevant for the sake of this work if the mining algorithm accurately represents the process model utilized for log production. It is far more crucial that students learn how to utilize the method, whether the identified process model conforms to the actual process is secondary.

2.4 eTutor++

This section will provide a brief overview of the platform on which the learning module is implemented. The Department for Business Informatics (DKE) at Johannes-Kepler-University Linz provides the eTutor++, an intelligent e-learning platform. It aids students in the understanding and application of new concepts, algorithms, and query languages. The eTutor++ is designed to give students relevant courses that correspond to university courses in the appropriate semester. Different assignment sheets can then be offered throughout various courses. Because these work sheets are designed to pursue a learning aim, task assignments are made based on the student's success in various difficulties. Section 4.2.5 explains how the task assignment is accomplished in detail. The assignments can then be completed online and either handed in (points can be awarded) or only tested (diagnosed). One feature that stands out in the case of testing is the ability for students to specify different feedback levels for their assignments. This option should aid in the best possible understanding of the relevant task type. The query languages relational algebra and SQL are currently offered. The alpha- and apriori algorithm will be supplied in the near future.

3

Related Work

In order to compare the results of this work with existing e-learning tools, I investigated the literature on other academic e-learning tools, automated feedback and grading of student assignments, and teaching/learning algorithms based on such online tools.
Two points should be made. First, it turns out that most reviews of tools designed for computer science students only create environments for learning programming (Garcia et al., 2018, Robinson and Carroll, 2017) or deal with automatic responses to this kind of programming exercises (Antonucci et al., 2015, Keuning et al., 2016, Odekirk-Hash and Zachary, 2001, Xu and Chee, 2003). Based on e-learning technology, it doesn't appear that there are many resources available for students to independently learn algorithms. Second, the manner in which humans engage with computers in the context of task processing and the feedback provided varies greatly. Below is a brief summary of selected literature in comparison with the tool designed and implemented in this work.

Keuning et al., 2016 give a systematic literature evaluation on automated feedback generation for programming exercises. They discovered that technologies rarely provide feedback for issue solving. Furthermore, teachers lack the ability to tailor learning materials and exercises to their specific needs. The majority of feedback supplied by such tools for learning programming is feedback on a submission's mistakes, which comprises a numeric value for the number of problems and a description of the mistakes. As a result, the authors state that there is a growing desire for greater and more precise feedback, as well as additional task specification options for teachers.
Antonucci et al., 2015 describe a feedback system that uses a methodology which is comparable to the one used in this work. They created an incremental hint system with multiple levels for programming exercises, each of which contains more information than the preceding one. The program can deliver various hints automatically, primarily code exposing cues, although textual hints are also available. The programming learning tool is likewise web-based and can automatically grade submissions. As a result, the authors discovered that students appreciated the step-by-step process leading up to the solution. The authors' implemented solution is more of a step-by-step instruction for the solution than an actual examination of the student's submission, which is another distinction with the tool developed in this work.
Wang and Wei, 2021 have constructed an e-learning platform to support the learning of two algorithms. Since they were specifically interested in the interaction between people and computers, they created a website design comprised of six modules. Among the modules are a watch, learn, discussion and assignment module. The former two contain the necessary algorithm knowledge

as well as algorithm animations. The assignment module provides an algorithm solution, but the article does not specify to what extent this solution can be submitted or whether automated feedback is provided.
Avancena et al., 2013 have created an algorithm learning tool. In this tool, four algorithms can be taught. This is accomplished through lessons and graphical representations of how the algorithms work. The tool can also include lecture notes, pseudocode, and algorithm illustrations. The main difference between this tool and the one developed in this thesis is that while algorithms will be taught, students will not be able to perform actual exercises and must instead follow a graphical representation of the procedure in this tool. As a result, there is no input from the learner, and hence no feedback can be supplied.

In numerous ways, the learning module implemented in this thesis differs from the e-learning tools published in the literature: a large number of tasks can be provided, teachers can influence the exercise generation, and a task processing in terms of utilizing an algorithm is provided for the student. Only in terms of feedback similarities with other e-learning systems may be drawn.

4

Dispatcher

This chapter describes the dispatcher's architecture in detail, including what needs to be implemented, how it was done, and how the back end interfaces function. It should also be mentioned that all GitHub references can be accessed using the link provided **here** (Görner, 2022a).

4.1 Dispatcher Architecture

The dispatcher of the eTutor++ is where various application requirements are handled and incoming HTTP requests are processed. It consists of three modules: the alpha algorithm, the process generator, and the evaluation logic for processing student submissions. It also handles the storage of task configurations, the alpha algorithm solution for created tasks, and the corresponding logs in a relational database. The back end also contains a REST controller for managing incoming HTTP requests from the platform. Figure 4.1 illustrates the architecture. Through various possible HTTP

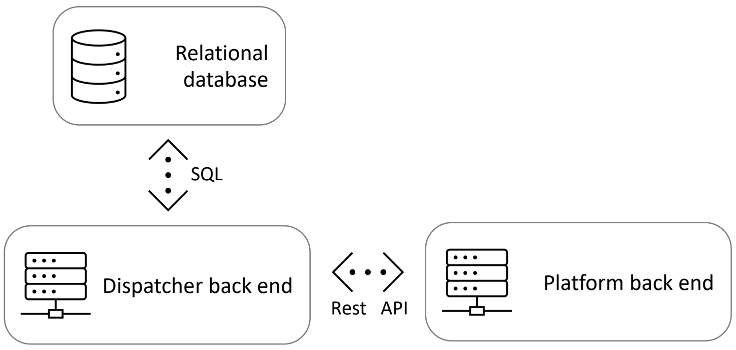


Figure 4.1: eTutor++ dispatcher architecture (own illustration)

requests in the REST controller, the dispatcher back end provides information to the platform back end. This is represented by a two-sided arrow with the description 'Rest API' between the two rectangles 'Dispatcher back end' and 'Platform back end'. This information is saved in a relational database, which can be queried using SQL in the context of Java Database Connectivity (JDBC). This is represented by the two-sided arrow 'SQL' connecting the rectangles 'Dispatcher back end' and 'Relational database'.

4.2 Dispatcher Implementation

As previously stated, the dispatcher back end consists of three modules that have to be designed and implemented. The alpha algorithm is the first module. Based on a certain event log, this module computes the solution of the alpha algorithm. It has to be implemented from the ground up. This will be covered in greater detail in Section 4.2.1. The process log generator is the following module. As mentioned in Section 2.2, this is a tool that must be tailored to the specifications of this project. Section 4.2.2 describes the procedure for implementing PLG2 and the necessary adjustments. The third module to implement is the evaluation logic, which includes submission analysis, grading and the corresponding reporting. This will be covered in Section 4.2.3. A suitable database structure is required to store the results of event log creation, task configuration, and task solution, which will be discussed in Section 4.2.4. A REST controller, in addition to the modules stated above, is required to manage the platform's incoming HTTP requests. As a result, the relevant HTTP requests and logic must be created and implemented. This is detailed further in Section 4.2.5.

4.2.1 Alpha Algorithm

The concept behind this module's implementation is that it just needs three lines of code to define the algorithm and run the solution stages. This is done in order to make it as simple to use when utilizing the algorithm as feasible. This can be also seen in Listing 4.1.

Listing 4.1: Alpha Algorithm (from AlphaAlgorithmApplication.java)

```
1          Trace t1 = new Trace(new String[]{"a", "c", "d"});
2          Trace t2 = new Trace(new String[]{"b", "c", "d"});
3          Trace t3 = new Trace(new String[]{"a", "c", "e"});
4          Trace t4 = new Trace(new String[]{"b", "c", "e"});
5
6          // generate Log
7          Map<String, Object> resultMap;
8          Log l1 = new Log(t1, t2, t3, t4);
9          AlphaAlgorithm simA = new AlphaAlgorithm(l1);
10         resultMap = simA.run();
```

As can be seen, the algorithm is declared and initialized (line 9), and the solution is executed by calling the *run()* method (line 10). The outcome is saved in a map, with the key being a *String* and corresponding to the appropriate solution phase. The outcome of the solution step is saved as the value of the map. Because these variables have distinct types, they are saved as *Object*.

The method *run()* contains all of the solution logic required to solve the various steps of the AA as well as the ordering relations. Before going into detail, the various data types used are explained below.
The following classes are introduced to operate with an appropriate data structure that is required to represent a Petri net structure as well as an event log: Event, Trace, Log, Pair, Place, Arc, Combination as well as OrderingRelation, FootprintMatrix and FootprintMatrixCell. It should be noted, however, that the footprint matrix - a graphical representation of the ordering relations - has been implemented and is functional but it has not been integrated into the learning module.

The 'Event' class is required for the creation of event objects that represent the concept of an event/activity as well as the concept of Petri net transitions. It is sufficient to construct events with a name, there is no need for further attributes.
This class includes a *setName(String name)* method for changing the name of an event and a *getName()* method for retrieving the name information of a specific event. Moreover, the 'Comparable' interface is used in this class, a method *compareTo(Event other)* is introduced. Furthermore, two additional methods are required. Methods *equals(Object o)* and *hashcode()* are implemented because a HashSet in class 'Log' is used and it is required to examine if the set's elements (traces) are equal. As a result, *equals(Object o)* and *hashcode()* must be overridden in class 'Traces' first, and because traces are distinguished by their events, *equals(Object o)* and *hashcode()* must also be overridden in class 'Events'.

A trace, as previously stated, is a series of events/transitions whose ordering inside the trace is important since it correlates to the sequence of executed activities. The class 'Trace' is introduced to reflect the concept of a trace. This class contains three significant characteristics. A class attribute (static) *transitions*, an object attribute *traceEvents*, and the class constructor. There are two ways to use the class constructor to create a new trace object.
Either it is built by passing an array of events in the form of a single string character to the class constructor (see Listing 4.1, line 1-4). After that, these events are transformed into event objects. If the single event is not already in the set, it is added to the class field *transitions*. Its purpose is to correlate to the first stage of the algorithm.
Or it is built with no specified input and declares an empty list object. This is done if events are added iteratively. There are various more methods in the class, such as getter and setter methods, that are not covered in detail. If you wish to learn more about these methods, please visit the relevant class on GitHub.

The 'Log' class is required for the creation of log objects that embody the concept of an event log. A log is defined by a set of traces. A log object can be created by passing it numerous trace objects as input (see Listing 4.1, line 8) or by simply initializing an empty object. Iteratively filling an empty log with traces is possible.
This class implements the computation methods used to solve the ordering relations (OR). The reason for this is that those methods operate on a log object and its underlying traces. For this reason, four methods are implemented, each of which corresponds to a relation. Before presenting the OR computation methods, a class called 'Pairs' is added to describe pairs of events in this context.

The objective behind the class 'Pair' is to make it as general as possible so that it can be used in as many different applications as possible. An object of this class, for example, can be a pair of two single events, a pair of an event and a set of events or a pair containing two sets of events. Therefore, the principle of generic types is applied to this class. Any two types can be supplied to the constructor to build a pair of objects. This preserves the ability to use the proper type for the pair's first and second positions. The class also has getter and setter methods, as well as a method that flips the first with the second position, which is used in the context of ordering relation computations. After discussing pairs, the methods for determining the ordering relations is now examined.

As previously stated, there are four methods for computing the OR. The method *directSuccession()*

computes the set of pairings that satisfy the first relation stated in Definition 1. Based on a given event log, a set of direct successors is returned. Each event in a trace is linked to its following event to form a pair in order to determine the result set. This is done for each trace in the log.
The second relation, i.e., causality, is addressed by the method *causality()*. This method builds on the preceding one. It examines the set of direct successors for a pair that exists in reverse order. If this is not the case, this pair is added to the set of causal relations.
The direct successors are also used in the third computation for the parallel relation. It is quite similar to the prior approach, except that if a pair in the set of direct successors has a pair in reverse order, it is added to the result set.
The last method *independence()* returns pairs of events that never occur together, which "means that there are no direct causal relations and parallelism is unlikely" (van der Aalst et al., 2004, p.1132). To achieve the result for this step, all possible combinations of the individual events must first be calculated. It must be true for a pair of events that never follow each other that neither the pair is in the set of direct successors nor that this pair with switched places happens in the result set of direct successors.
The 'Log' class additionally includes a method for generating a footprint matrix based on the ordering relations' result sets. Because this is not used in the learning module, the reader is referred to the relevant classes (FootprintMatrix, FootprintMatrixCell, OrderingRelation) on GitHub. This, however, provides an opportunity for further work on this learning module.

Two more classes are required and implemented, particularly to represent the Petri net concept. The class 'Place' relates to Petri net places. A place is defined in this work by a pair of input and output transitions. As a result, in order to build a new place object, a pair containing a list of events for both the first and second positions of this pair must be provided. Further methods, particularly getter methods, can be found in the respective class on GitHub.

The 'Arc' class relates to Petri net arcs. Since arcs connect transitions to places and places to transitions, this class is a general class that allows to define an arc object as needed in a specific scenario. An arc is also defined as a pair with a first and second position. These position can take different types. To generate a new arc object, the constructor must be provided with the elements for the first and second positions. Further methods, particularly getter and setter methods, can be found in the respective class on GitHub.

Finally, all of these data types are combined and used in the class 'AlphaAlgorithm'. This class includes all methods to calculate the appropriate steps of the ordering relations and the algorithm. An event log must be given to the constructor in order for the steps to be performed.
The *run()* method is the core method of this class. It returns a result map with the id of the respective solution step as the key and the solution of this step as the value. The ordering relations are identified by the id 'orl*x*' where x is the relation number ($x \epsilon \{1, 2, 3, 4\}$). The alpha algorithm steps are identified by the identifier 'aal*y*' where y is the relation number ($y \epsilon \{1, 2, 3, 4, 5, 6, 7\}$).

The solution for the ordering relations is computed using the given event log and its corresponding methods, as described in the previous segment. The first step of the algorithm (T_W) can be solved by obtaining the 'Transition' class attribute, which contains all distinct transitions. Step two (T_I) and three (T_O) are identified by examining the log with its associated getter methods.
Additional methods are necessary to solve steps four through seven. These methods and their underlying concepts are described briefly below.

In order to derive step four (X_W) the *deriveXlSet()* function and specifically implemented methods from the class 'Combination' are used. This method's strategy is as follows: First, compute all possible transition combinations for all possible lengths. Following that, all combinations that are not in the result set of the *independence()* method, which is based on the ordering relations, are eliminated. The third step is to compute the product (in the context of pairs) of all remaining combinations and determine if this product of all remaining possible combinations meets the causality condition. The pair of events is added to the result list if the causality criterion is met. By selecting just the largest members in terms of set inclusion, the relationship Y_W is generated from X_W (van der Aalst et al., 2004). The function's (*deriveYlSet()*) strategy for computing this step of the algorithm is as follows: it examines for each pair (A,B) in X_W whether there is a pair (A',B') in X_W for which A is a subset of A' and B is a subset of B', with $(A, B) \neq (A', B')$. If A is not a subset of A' and B is not a subset of B', the pair is included in the result list.
The next step is to use the *getPlSet()* method to identify the underlying process's places (P_W). This approach is based on the Y_W result list and determines the places that connect the transitions. Since a place is defined as a pair of events, an object is created by providing the pairs to the place constructor. It is worth mentioning, that this method includes source and sink places.
Finally, the *deriveFlSet()* method computes the arc relation F_W, which is the final step of the AA. This method receives the places that were previously calculated. The crucial component in the implementation is determining whether the passed places are a source or sink place. If either of the two scenarios is true, the arc is given an 'i' as the first position or a 'o' as the second position, respectively.

4.2.2 Process Log Generator

As explained in Section 2.3, the AA requires an event log as input. The PLG2 tool is used to generate this log. On the one hand, this section describes the customization, on the other, the tool's implementation.
Listing 4.2 shows the main method *finalLogGeneration()* for constructing a process and simulating it, with the result being an event log. Furthermore, it can be seen that the simulation continues until various conditions are met. Based on this method, the implementation is explained.

It is evident from Listing 4.2 that method *finalLogGeneration(...)* must be executed with five parameters. These parameters are set by the tutor when the task is created. Section 5.2.1 explains this procedure.
The parameters are the minimum and maximum log size (min/maxLogSize), the feasible number of activities within a trace (min/maxActivity) and the configuration number (configNr). The log size variables determine the range within which the number of traces in an event log can vary. The maximum and minimum feasible lengths of each trace inside a log are represented by the number of activity parameters. The pre-set values for the probability of the production rules (see Section 2.2) are determined by the value entered for the parameter 'configNr'. This parameter serves as the input for method *simulate(configNr)*.

The procedure of method *simulate(configNr)* (see Listing 4.2, line 5) combines numerous operations. First, a random process is generated. This occurs in accordance with the specified settings, which are identified by the parameter 'configNr'. The parameter's input should be one of the three values 'config1', 'config2', or 'config3'. A default value is automatically chosen if no value is given or a different value is given. A probability configuration for process production rules is chosen based on these variables. This configuration specifies maximum number of OR and AND branches as well as the weights for loops, single activities, sequences, empty sub-graphs and XOR/AND branches. Furthermore, the probability of data objects is set, as is the maximum depth. The depth of the derivation tree is defined by the maximum depth parameter. The exact values for these parameters are listed in Appendix A. After the randomize configuration is initialized the process is generated.

Listing 4.2: PLG2 (from SimulationApplication.java)

```
1      public static List<String[]> finalLogGeneration(int minLogSize, int
           maxLogSize, int minActivity, int maxActivity, String configNr)
           throws Exception{
2           int simulationCounter = 0;
3
4           do{
5               simulate(configNr);
6               simulationCounter++;
7           } while (!(logSize >= minLogSize && logSize <= maxLogSize &&
                maxActivityTrace <= maxActivity && minActivityTrace >=
                minActivity));
8
9           System.out.println("Simulation complete! \nResult Log as input
                for AlphaAlgo (" + simulationCounter + " simulation(s)):");
10          for (String[] s: resultLog){
11              System.out.println(Arrays.toString(s));
12          }
13
14          return resultLog;
15     }
```

The process is then simulated. This is accomplished using the appropriate algorithms, which are not addressed in depth because they are beyond the scope of this work. The technical report of Burratin, 2015 is referred to for more extensive information. However, it should be noted that the simulation algorithms require the generated process model and the number of traces as input. The number of traces is chosen at 1000 for this work in order to achieve a sufficiently large subset of probable behaviours. Furthermore, the option for 'noise' is deactivated in the simulation configurations. This procedure produces an event log in a data format with additional attributes (e.g., time stamp of the activity) that are not applicable in this form for the purposes of this work. As a result, this event log must be processed and adjusted.
For this purpose, each event in a trace is processed to match the requirements and saved in an appropriate data type that can be utilized in the same way later. When the changes are made, the modified log is analysed and further processed. Because the identity of cases and the frequency of traces are ignored following the theory of van der Aalst et al., 2004, the log must be adjusted further. This occurs in the 'LogAnalysis' class. As a result, there are no duplicate mentions of the same trace in the event log, and each trace consists just of a string array containing the various activities. The log size (number of traces) and the highest and lowest number of activities in a trace are also recorded. These parameters are then utilized (see Listing 4.2, line 7) to simulate a log based on these parameters.

When a process and its associated event log match the specified parameters, the result log is returned.

4.2.3 Evaluation Logic

The eTutor++ dispatcher back end not only includes the alpha algorithm and the PLG2 tool but also the evaluation logic of a student submission. This means that the evaluator module analyses and grades each student's submission. In addition, a report is generated for each submission depending on the analysis. The report becomes more thorough when the diagnosis level goes up. The evaluation logic is described in this section. This includes the analysis, grading, and report creation phases. These phases are represented by the corresponding methods which are part of the 'PmEvaluator' class.
The structure of these three phases are already defined, since this procedure should be the same for each module embedded in the dispatcher. This means that the three methods' inputs are specified, as is the data type of the return value. The structure already includes default classes for this purpose (DefaultAnalysis, DefaultGrading, DefaultReport). This is because exactly these return values are expected in the front end. Since the structural framework for the evaluation logic was already provided, the solution for this module had to be implemented to fit within this framework. This solution logic is then applied consistently across all three methods.
The purpose of this section is to present the overall goal and concept behind the methods' implementation. A full description of the entire code would go beyond the scope of this document and can be seen at GitHub. All new classes created for the purposes of this project either begin with 'Pm' or contain these characters within the class name.

Analysis

The first method used in a submission evaluation is *analyze(. . .)*. Its main purpose is to analyse a student's submission based on various evaluation criteria and to provide information if the submission fits the respective solution. This is the default structure. Before going into the procedure in detail, the evaluation criteria used for this work/module are stated first.
Three criteria are used to evaluate the student's submission, which includes the answers to the ordering relations and the respective AA steps. 'CORRECT PAIRS', 'CORRECT EVENT', and 'CORRECT PAIRS ALPHA' are the criteria used. The first criterion is used to evaluate the submitted ordering relations' answers. Criterion two is applied to the first three AA steps, and criterion three is applied to the remaining algorithm steps.

Various parameters are passed to the method. One of them is *passedAttributes*, which contains the student's task submission answers. These answers correspond to each step of the OR and the AA and are accessed using a key for the submitted answer. This key is defined by 'orl*x*' for the OR steps and 'aal*y*' for the AA steps. Another passed parameter is the unique task identifier, which is used to query the stored correct solution for this specific task. The analysis can begin with this information, the submitted answers, and the corresponding solution.
The analysis is based on the following concept: the default class 'DefaultAnalysis' is extended by a new class 'PmAnalysis'. This class stores a map in addition to the default attributes, which stores the submission id (e.g., 'orl1') of the corresponding response as a key and an object of the class 'PmPartialSubmissionAnalysis' as the associated value. As a consequence, an object

of class 'PmAnalysis' can be thought of as a container for the partial analysis of each answer of a corresponding step of the ordering relations and the alpha algorithm (see Listing 4.3, line 4). The actual analysis of each provided answer is then performed using an object of the 'PmPartialSubmissionAnalysis' class (see Listing 4.3). This class is detailed in depth below since it is the heart of the analysis.

Listing 4.3: Process Mining Evaluator - Analyse (from PmEvaluator.java)

```
1         for(var entry: completeSubmission_Param.entrySet()){
2             PmPartialSubmissionAnalysis partialSubmissionAnalyzer = new
                 PmPartialSubmissionAnalysis(entry.getKey());
3             partialSubmissionAnalyzer.analyze(entry.getValue(),
                 pmAnalyzerConfig);
4             analysis.addPartialSubmissionAnalysis(entry.getKey(),
                 partialSubmissionAnalyzer);
5         }
```

For each object of 'PmPartialSubmissionAnalysis' (PSA), information on whether the partial submission is correct, whether an exception occurred, and which criteria were used for the analysis can be queried. As previously stated, each partial submission is evaluated using a single criterion. This means that each partial submission can have an assessment criterion associated to it. However, in order to make the analysis module as general as possible as well as to allow for future changes or additions, the criterion for each partial submission is stored in the following way.
For each object of PSA the evaluation criterion is stored in a map with its respective criterion analysis. This enables the evaluation of a partial submission based on several criteria, which is not currently used in this version of the learning module but could be in future versions.
The analysis' logic for each partial submission is contained in the method *analyze(...)* (see Listing 4.3, line 3). This method calls the relevant criterion analysis method depending on the partial submission and the criterion to be evaluated, which then thoroughly examines the partial submission and returns an object of the respective criterion analysis class. Four classes of criterion analysis are implemented, the concept of which will be explained in more detail below.

All criterion analysis classes have one thing in common: a criterion is either satisfied or not, and it has an exception if it is not satisfied. This is accomplished through the use of an abstract class called 'AbstractPmCriterionAnalysis,' which all criteria classes extend. Furthermore, more specific information for each criterion examined should be saved in order to be used later for report creation. At this point, the classes diverge due to the different data types in the specific information (such as pairs that are missing or surplus).

One concrete example will be used to clarify the concept/solution for analysing submitted solutions. The concept demonstrated here can then be applied to the remaining partial submissions in the same manner.
The evaluated partial submission is an answer to ordering relation one, hence it has the id 'orl1' and the appropriate evaluation criterion is 'CORRECT PAIRS'. With two input variables, the method *analyzePairs(analyzerConfig, submittedAnswer)* is used. The first input is the analysis configuration, which includes the correct solution to this submission. This is an object from the 'PmAnalyzerConfig' class. The second input is the student's actual submitted answer. This approach checks the validity of the student's answer. It is important to mention, that the answer

must have a certain syntax. This syntax is shown at each input field in the front end and is based on the work of van der Aalst et al., 2004. The student's answer, which is in the form of a data type string, is then converted step by step into an object of the same data type as the correct solution. It is of the type *Set<Pair<Event, Event> >* in this instance.
The transformation procedure is as follows: first, the answer string is converted into a StringBuilder object (e.g., called 'workSB'). The transformation algorithm is executed as long as this object contains an open parenthesis. The first opening and closing parenthesis indexes are saved. A new string object is generated based on these indices (e.g., called 'tempSubstring'). This is a string of the form '(x,y)'. It is then iterated over this string object, and the entered characters are converted to an object of type 'Event'. If the first position of a new generated object with the type 'Pair' is not yet occupied, the first letter/event is assigned to it; otherwise, it is assigned to position two. If the element on the index being iterated is not a letter, the index is skipped and the algorithm moves on to the next index. Furthermore, the case is evaluated to see if the direct neighbour of a letter is likewise a letter. This is checked because it is feasible that the created process and event log will contain more than 26 separate activities, resulting in the 27^{th} activity consisting of two letters (e.g. AA). When a new object of type 'Pair' is created, it is added to the result set. The corresponding string ('tempSubstring') is removed from the StringBuilder object's ('workSB') initial state. This procedure is repeated as many times as the student response has an open parenthesis.
If the transformation of the student's answer is complete, the submission is examined to see if it matches the corresponding solution. This process is divided into two steps.
The first step is to identify the surplus pairs. These are pairs too much in the submitted answer. If a submitted pair does not appear in the list of correct pairs, it is treated as a surplus pair. The second step is to determine the missing pairs. To do so, it is determined whether each accurate pair from the solution is present in the list of submitted pairings. The criterion is flagged as not satisfied and a matching exception is created as soon as either too many or too few pairings are present.
The evaluated criterion and the accompanying result are stored in the PSA object as soon as the analysis is complete. The properties of the PSA are also changed in accordance with the analysis. If the criterion is met, the PSA is marked as correct; otherwise, the exception from the criterion analysis class is taken accordingly.

This procedure is generally adopted for the student's solutions to the different input fields. However, it differs in one essential point. The algorithms for transforming the student's solution differ in that the syntax of the input differs. This is considered accordingly with the transformation, whereby the principle stays the same as in the preceding example.

The final step in the submission evaluation is to determine whether all of the solutions entered by the student correspond to the correct solution. To accomplish this, all PSA objects are iterated and examined to see if the partial submission is accurate. If a PSA is incorrect, the entire submission is tagged as incorrect. As a final result, an analysis object is returned by the *analyze(. . .)* method. This includes information on whether or not the student's submission was correct, as well as detailed information on all answers in specific.

Grading

Once the analysis is done, the submission can be graded. The previously constructed analysis object is used as an input by the *grade(. . .)* method. Additionally, the tutor defines the maximum points for this method in the context of a configuration generation (see Section 5.2.1). This method's goal is to compute the points scored based on the submitted answers and the related analysis. Since the platform back end is where the diagnosis level discount is determined, the obtained points are then sent to the front end and used as the foundation for the calculation of the final result.

The grading is based on a similar concept as the analysis. The present dispatcher structure includes a 'DefaultGrading' class, which is defined by the two properties 'points' and 'maxPoints'. The 'PmGrader' class extends the default class by providing an attribute that stores a separate 'PmPartialSubmissionGrading' for each step of the ordering relation and the alpha algorithm, comparable to the analysis. This class serves as a container for the grading of each submitted solution step.
The idea here is to make the method as generic as feasible. As a result, two configuration classes are created: 'PmGraderConfig' on the one hand, and 'PmCriterionGradingConfig' on the other. The first of these two classes stores the appropriate 'PmCriterionGradingConfig' for each used evaluation criterion. This setup allows for the storage of both positive and negative points. It is worth mentioning that in current version of the project, no negative points are awarded for incorrectly submitted answers, although this option is available if desired. Furthermore, the current implementation assigns one point to the evaluation criteria 'CORRECT PAIRS' and 'CORRECT EVENT' and two points to the evaluation criterion 'CORRECT ALPHA PAIRS', since the input for this criterion is more difficult than the input for the other two criteria. This gives a total of 14 points. This number of points should also be specified in the user interface configuration creation.

Listing 4.4: Process Mining Evaluator - Grading (from PmEvaluator.java)

```
1           while(pmPartialSubmissionAnalysisIterator.hasNext()){
2               partialSubmissionAnalysis =
                    pmPartialSubmissionAnalysisIterator.next();
3               partialSubmissionGrading = new
                    PmPartialSubmissionGrading(partialSubmissionAnalysis,
                    pmGraderConfig);
4               partialSubmissionGrading.grade(true);
5
6               pmGrader.addPmPartialSubmissionGrading(
7                       partialSubmissionAnalysis.getSubmissionID(),
8                       partialSubmissionGrading);
9           }
```

Listing 4.4 demonstrates the grading procedure. The provided analysis object includes an iterator for looping through the saved partial submission analyses. As a result, each partial submission is then graded independently (see Listing 4.4, line 3 & 4). Furthermore, in this step, one can choose whether or not to grade partial points. The boolean flag is set to true (line 4), indicating that partial points are graded. The 'PmPartialSubmissionGrading' class (PSG) is explained in the following.

A PSG object contains the total achieved points for this given answer, as well as the points achieved per criterion applied to an input, for each partial submission. This means that in this phase, the potential of applying multiple assessment criteria to a single input is also introduced. Method *grade(...)* (see Listing 4.4, line 4) is the main method for grading a partial submission. It determines for each criterion used for this submitted answer if the criterion is satisfied. If so, positive points are given. Negative points could potentially be assigned if the criterion is not met. The points earned for each criterion reviewed are saved. As previously stated, it is possible to choose whether or not partial points are graded. This means that points for a correct criterion can be graded even if the partial submission as a whole is incorrect due to an unsatisfied second criterion.
Finally, each partial grading is saved in the 'PmGrader' object with its unique id (see Listing 4.4, line 6-8). Since the front end expects the overall sum of points from the default grading object, the points are totalled over each PSG object and returned by the *grade(...)* method.

Report

The report is the final stage of the student's submission evaluation. It serves as the foundation for the proper response based on the level of feedback. The dispatcher also provides the structure for the report as well as for the other two stages of the evaluation logic. The 'DefaultReport' class defines a report's structure, which includes a hint, an error message, and a description of the error, all of which are implemented as single strings. Method *report(...)* generates a report based on the analysis object and the student's feedback level. A variable of the type locale is also required as input. This variable indicates which language the feedback is delivered in and is also selected by the student in the user interface. If the student's answers are 'submitted', the resulting report is based on feedback level two. This is the default value. If the student's activity is to 'diagnose' the submission, the report is generated based on the feedback level selected. The report is constructed using the *createReport(...)* method implemented in the 'PmReporter' class with the analysis object, the grading object, the local variable, and a 'PmReporterConfig' object as input. An object generated from the 'PmReporterConfig' class contains information about the student's diagnosis level and the chosen action (diagnose or submit).

The concept of creating reports is comparable to that of grading, and is as follows:
each PSA from the corresponding analysis object is iterated over. If this partial submission is correct, no feedback for this respective step is needed and the next element is taken. If it is not accurate, an object of the 'PmErrorReport' class is generated. Once this object is generated, an iteration over each assessment criterion used within each partial submission is performed. Again, this adheres to the generality notion, which is applied throughout the review process. If one of the used criteria is not met, an error and description message is generated if the diagnose level is greater than zero (a value of zero indicates that no feedback is selected). The associated criterion analysis class and the value of the locale variable determine the actual phrasing of the error and its description. The error message always follows the same idea, stating that the student's input is incorrect.
The extent of the error description is determined by the diagnostic level. For the first level ('diagnose level little'), the description of the error only says that something in the answer is missing and/or surplus. Level two ('diagnose level some') offers information on the amount of

missing and/or surplus elements. The third diagnosis level ('diagnose level much') specifies how many elements are missing and/or are surplus, as well as the detailed description of the elements. This is the same as a sample solution. These details are available since they are kept in the relevant analysis objects.
A new object of the 'PmErrorReport' class is created for each PSA. By completing the feedback message for a certain PSA, this error report is saved in a list attribute implemented in a 'PmReport' class object. This variable contains a collection of all error report objects generated for the appropriate PSAs. The most significant fact to retain the dispatcher's structure is to append the error and description message created for a PSA to the initially empty string of the 'DefaultReport' attributes. With each PSA feedback, this string is extended with the relevant error message and descriptive message.

4.2.4 Database Structure

In order to store the event log generation result, the answer to the respective steps of the ordering relations and the alpha algorithm as well as the configuration parameter, a relational database structure had to be thought of and implemented. The database is named 'process_mining' and is stored on the dispatcher server. On the public schema, three tables are added to the database.

The first table is named 'exerciseconfiguration'. It stores the values for the process log generation which are defined in the user interface. As one can see in Table 4.1, it consists of six columns.

Table 4.1: Relational database - exerciseconfiguration (own illustration)

	config_id [PK]integer	max_activity integer	min_activity integer	max_logsize integer	min_logsize integer	configuration_number character varying (200)
1	1	8	3	8	3	config2

The first column ('config_id') contains the unique identification for each configuration setting and serves as the table's primary key. Section 4.2.5 explains how this id is assigned. Columns two and three hold the maximum and minimum number of activities that a trace can have. The greatest and smallest feasible logsizes are shown in the fourth and fifth columns. The configuration number is stored in the table's last column. As seen in Table 4.1, an example row (row 1) is entered, displaying the values for an example configuration.

The second table is named 'logs'. The randomly generated log is stored in Table 4.2. Because the log contains an unknown amount of traces, the table scheme described below is developed.

Table 4.2: Relational database - logs (own illustration)

	log_id [PK]integer	exercise_id integer	trace character varying[] (2000)
1	1	1	{A,B,C,F,D,H}
2	2	1	{A,B,C,E,D,H}
3	3	1	{A,B,C,G,D,H}

Since the number of traces is uncertain, the various traces are stored in one column in Table 4.2. Furthermore, each trace is identified by an unique identifier (column 'log_id'), which serves as the table's primary key. The 'exercise_id' column labels these traces with the associated id of an exercise in order to retrieve the traces relating to an event log and hence to a specific exercise. Traces are saved in the form of a character array. Three example rows are inserted to demonstrate the usage. The event log contains three traces, as can be seen in Table 4.2. Further, the event log serves as the basis for an exercise with the exercise id '1'.

The name of the third table is 'randomexercises'. Table 4.3 has 13 columns: two columns to hold the identifier of the specific task and the identifier of the related configuration parameter, four columns for the solution of the relevant ordering relation and seven columns for the solution of the alpha algorithm.

Table 4.3: Relational database - randomexercises (own illustration)

	exercise_id	or_one	...	or_four	aa_one	...	aa_seven	config_id
	[PK]integer	cv(5500)	...	cv(10000)	cv(2000)	...	cv(10000)	[FK]integer
1	1	...	...	...	...	...	...	1

The answer to the four stages corresponding to the ordering relation are kept in columns 'or_one' through 'or_four' of Table 4.3. The two additional columns for these relations are represented by the dots between these two columns. The solution for the alpha algorithm is kept in columns 'aa_one' through 'aa_seven'. Once more, the remaining columns that correspond to the AA stages are represented by the dots between these two columns. The results of the OR and AA procedures are all saved with the data type 'character varying' (cv) of various widths. In essence, this means that they are kept as JSON strings of varying lengths. Moreover, the exercise identifier is kept in the first column as an integer type, serving as the table's primary key. The last column of Table 4.3 has the matching identifier for the configuration settings that were used. This attribute also serves as the foreign key of this table and references the primary key 'config_id' of Table 4.1. One row has been placed as an example. This would be a solution for an exercise with corresponding id and the equivalent solutions based on the configuration parameters with corresponding id.

4.2.5 REST Controller

When a configuration is created, edited, or deleted in the front end, a new task is put in place, or a student's provided solution is to be evaluated, the information is handled and processed in the dispatcher back end. Processing requires information to be exchanged between the dispatcher back end and the platform back end. This occurs as a result of the REST controller, which is described in this section. This controller handles the HTTP requests to manage a process mining exercise and is implemented with the 'ETutorPMController' class. It is worth mentioning that the '@RestController' annotation is part of the Spring MVC concept. A request handling method "automatically serializes return objects into HttpResponse" (baeldung, 2022) when the RestController annotation is present. In addition, the controller automatically calls the appropriate request handling method.

Six HTTP requests must be handled for the purposes of this work. A request to create and a request to update an exercise configuration, with the respective parameter being stored in the relational database. Further, a request that manages the deletion of specific exercise configuration and a request that returns the corresponding database parameters for an existing configuration.

The fifth request handling method handles the request to generate a random exercise based on a previously created exercise configuration. The sixth request that must be fulfilled is to return an event log belonging to a certain exercise. This is requested since the log is visible to students on the task editing page.

The concept of request handling methods will be demonstrated using an example method. This example method was chosen since it does the majority of the tasks that the other methods do as well. It not only fetches data from the database but it also stores data in the database. Furthermore, the PLG2 tool and the alpha algorithm are used in this method. Please see the related GitHub for more details on the other request handlers.

The example method *createRandomExercise(@PathVariable int configId)*, which handles the request to generate a random exercise, is shown in Listing 4.5.

Listing 4.5: REST Controller - Exercise creation (from ETutorPMController.java)

```
 1      @GetMapping("/exercise/{configId}")
 2      public ResponseEntity<Integer> createRandomExercise(@PathVariable
            int configId) throws ApiException{
 3          logger.info("Enter: createRandomExercise() {}", "for config:
                "+configId);
 4          try{
 5              int id = resourceService.createRandomExercise(configId);
 6              logger.info("Exit: createRandomExercise(){} with Status Code
                    200", id);
 7              return ResponseEntity.ok(id);
 8          }catch (Exception e){
 9              logger.error("Exit: createRandomExercise() with Status Code
                    500", e);
10              logger.info("Deleting exercise");
11              throw new ApiException(500, e.toString(), null);
12          }
13
14      }
```

The method is a '@GetMapping' that can be called via the address 'http://localhost:8081/pm/exercise/{configId}', as shown in Listing 4.5, line 1. This means that the configuration id of a stored configuration is required as a 'PathVariable' of this method. The computations are carried out using the appropriate resource service method *createRandomExercise(configId)* (see Listing 4.5, line 5). This function is implemented in the 'PmResourceService' class, which does all computations necessary by the REST controller's request handling methods. This class and its relevant methods will be explained in the following.

The method *createRandomExercise(configId)* handles the creation of an exercise based on a specified exercise configuration. It initializes the connection to the database and also initializes that a unique exercise id is retrieved and returned, as well as the next available log id. Therefore, two separate functions are called. The first function, *getAvailableExerciseId()*, queries the database to find an available id. It selects the maximum exercise id already present in the table 'randomexercises' and adds a one to it. This ensures that the biggest number is always chosen and that the id is unique and at the end of the table. This value serves as the id for the exercise to generate. The same procedure is employed to obtain an id for the first entry of the newly generated log.

The new exercise is then generated based on the ids assigned to the exercise and the first item of an event log. To accomplish this, method *createRandomExerciseUtil(Connection conn, int exerciseId, int configId, int logId)* is used. All computations and data storage are summarized in this method. This comprises producing a random process depending on the specified configuration parameter, saving the resulting event log, applying the ordering relations and alpha algorithm steps to the log, and storing the answer to these computations.

First, a process is generated randomly with the result of an event log that matches the specified configuration option. The actual configuration parameters are fetched from the database using a SQL query in order to accomplish this. Only those parameters are chosen that have the matching configuration id provided to the method. These settings are then used to create a process and the associated event log using the procedure described in Section 4.2.2. By executing the appropriate SQL query on the database in the next step, the result of this computation is instantly saved in table 'logs' (see Table 4.2). After that, the next step is to run the algorithm steps based on the event log. This is accomplished using the procedure described in Section 4.2.1. When the results are ready, they must be transformed from their original data type to a string before being put in the database. This is done with the help of an 'ObjectMapper' object. This converts the objects to a string, which can then later be restored straight to the original data format using the matching method. However, it should be noted that this applies to all algorithm solutions except the final step, the solution to the arcs of the Petri net (key 'aal7'). Therefore, this algorithm's output is translated to a string in a different way. The disadvantage is that this solution step cannot be simply converted back to its original data format later; instead, the string must be manually converted to an object with the proper data format. This is then handled using an effective method implemented that has no drawbacks and is error-free. When the conversion is complete, the values are placed in the database in the table 'randomexercises' using the appropriate SQL statement. All database operations are enclosed in a try/catch block to guarantee that errors are caught rather than causing the program to crash. Method *handleThrowables(. . .)* should be highlighted for this. This method handles an SQL exception by rolling back the connection, logging a message and throwing a 'DatabaseException'.

If method *createRandomExercise(configId)* was successfully executed without errors, the corresponding exercise id is returned (see Listing 4.5, line 5) and an HTTP response with status code 200 is sent to the platform back end. Furthermore, this HTTP response includes the generated exercise id, which will be utilized to continue working in the platform. If an error occurs, a status code 500 is returned, and the relevant information is logged.

The same logic applies to the other request handling methods. The 'PmResourceService' class always has a corresponding method that calls the necessary service methods. These then perform a database query or insert new values into the database, as well as the necessary computations.

5

Platform

This chapter describes the platform's architecture in detail, including what needs to be implemented, how it was done, and how the front and back end interfaces work. The code can be followed up on GitHub using the link provided **here** (Görner, 2022b).

5.1 Platform Architecture

Before going into the actual implementation in the following sections, Figure 5.1 provides an overview of the eTutor++ platform's architecture. The platform architecture is composed of a

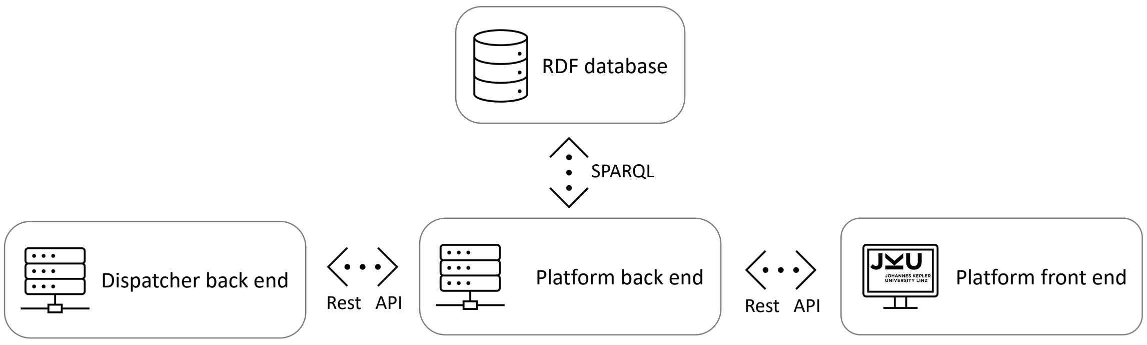


Figure 5.1: eTutor++ platform architecture (own illustration)

front end and a back end, which are depicted as rectangles in Figure 5.1. It is run by a Fuseki server. RESTful APIs are used to exchange data between the front and back end of the platform, as well as between the platform back end and the dispatcher back end (see Section 4.2.5). These connections are represented in Figure 5.1 by the dotted arrows. The structure of the front end web app is based on the Angular framework, which provides a structure with components that make up an application. These components include three files: a TypeScript class, a HTML template and a style sheet.
The platform's back end manages the information exchange between the dispatcher and the platform, which occurs through the use of HTTP requests. Furthermore, all relevant data is stored in a graph database that employs the Resource Description Framework concept (RDF). The query language SPARQL is used to store or query information, such as student, task, and system state information. The methods for querying the database and storing data are implemented in the platform's back end. The platform's back end and the dispatcher's back end are Spring-based applications. The front end is based on the Angular framework.

Since these frameworks are already part of the eTutor++ and the module presented here is embedded in the existing structure, and an explanation of these frameworks would exceed the scope of this work, it is referred to the corresponding documentation for detailed information.

5.2 Platform Implementation

There are numerous requirements for integrating the learning module into the platform architecture. Two user interfaces in the format of web forms are required to be created. One for configuring the event log simulation's control parameters. Only the tutor has access to this data. The students can work on the assignment and either submit it or diagnose it using the second user interface that is accessible. This requirement refers to the front end of the platform. This provides the web application and is embedded in an Angular framework. The concept of the implementation, the changes made, and the new components added are described in Section 5.2.1.
Section 5.2.2 describes all changes and additions that affect the platform's back end. This includes, among other things, the addition of new interfaces, new processes required for smooth operation, and the addition of new 'Data Transfer Objects' (DTO).

5.2.1 Front End Implementation

Since working on an exercise begins with specifying configuration parameters to control an event log simulation, the component for defining those parameters is explained first. The implementation approach for the task editing page is described next. It should be noted, however, that this chapter only describes the components of the front end. Section 5.2.2 describes the associated interfaces handled in the back end.

Configuration Parameter Component

In order to make use of the platform front end's existing structure, the entire implementation of the user interface for entering configuration parameters is embedded in the 'task-update.component'. The component includes the appropriate html, typescript and stylesheet (scss) files. However, this component is primarily used to create new exercises on the basis of the task types that are supported. In the framework of this work, it is now also utilized to create a task configuration.
As a consequence, the existing html file that is implemented as an update form is supplemented on the one hand by the input fields required for the configuration, while on the other hand some already existing input fields are used. Each input field is enclosed by a '<div>' tag, which defines a page segment and/or is used to group elements together. In order to use an HTML form for multiple purposes, the Angular framework's 'ngIf' function is used to display input fields only if they match a specific task type. The input fields that are the same for all task types are the task header, information to the creator, the organization unit, and the task difficulty. After entering the general information, the user must select the appropriate task type. The 'ProcessMining task' must be selected for this learning module. Once this task type is selected, only the input fields assigned to this task type in the corresponding html file appear. The already existing input fields used for a process mining task are the maximum points assigned to the exercise and the diagnose level weighting. The weighting of the diagnose level is a factor that reduces the achieved points as the diagnose level increases.

However, new input fields had to be defined as well. This was accomplished by creating a '<div>' tag for each attribute that contained the definition for the corresponding attribute. The attributes are maximum and minimum logsize, maximum and minimum number of activities, and configuration number. Furthermore, the corresponding text phrase is defined for each attribute. The information about how the attribute is displayed in English and German is stored in the two json files 'task-management', which are located in the corresponding German or English folder. In addition, the task instruction is created in this form, which is displayed on the task editing page.

The configuration information provided by the tutor in the user interface is handled by the typescript component 'task-update.component'. However, in order to work with the variables that define an exercise configuration, they must be added in another typescript file called 'task.model'. The variables are added to the interface that defines a new task (INewTaskModel) in this step. The new task assignment type must also be defined in this script, along with its corresponding value. The stored value for a process mining task is 'http://www.dke.uni-linz.ac.at/etutorpp/TaskAssignmentType#PmTask' . Three code sections of the previously mentioned typescript component 'task-update.component' deserve special mention. This is the variable *updateForm* on the one hand, and the two methods *save()* and *taskModel(value: ITaskModel | undefined)* on the other hand.
The variable *updateForm* is a new Angular 'FormGroup' object that contains a group of variables that match the names of the variables used as input fields in the html file. On the one hand, this form design allows the entered values to be saved and accessed later. On the other hand, when the configuration user interface is invoked again, the previously entered values can be inserted into the input fields.
Method *save()* is the second code section to be highlighted. If the tutor wants to save the entered configuration values in the currently open configuration parameter form, this function handles the activity. This method creates a new task object (*INewTaskModel*) and assigns the relevant values stored in the *updateForm* variable to the new task object's related attributes. Furthermore, the method addresses two potential scenarios. The first probable case is that no data have ever been entered and stored in the currently opened input form. This means it is a new input form that will be saved using the appropriate method. The other case is that it is an input form containing previously saved values, meaning that it is no longer a new form and is thus saved in an *ITaskModel* object rather than in a new *INewTaskModel* object. It should be noted at this point that the *ITaskModel* interface is an extension of the *ITaskNewModel* interface, with the addition of a task id and a creation date. At the end of this section, the process for saving a new task is explained.
Method *taskModel(value: ITaskModel | undefined)* is the third code section that requires special attention. When opening an already stored configuration parameter input form, this method is responsible for showing the entered values in the respective input fields. The saved attributes from the passed variable *ITaskModel* are assigned to the matching attributes in the *updateForm* for this purpose and can thus be mirrored in the user interface.

As previously stated, if the tutor saves the exercise configuration parameters, this information is saved in the dispatcher back end. However, various interfaces are required in order to communicate these data to the dispatcher back end and ultimately store them in the relational database. The first interface is explained in depth below since it is the interface between the platform front end and platform back end, as seen by the dotted arrow in Figure 5.1 between the two respective rectangles.

If the configuration parameter form is a new form, the object containing the information is saved using the method *saveNewTask(task: INewTaskModel)*. It is implemented in the typescript file 'task.service' and sends a post request to the platform's back end, where the data is subsequently processed further. The same scheme is used to get data into the dispatcher back end for an existing task configuration that is being edited. This is covered in greater detail in Section 5.2.2.

Task Editing Component

This section describes the solution concept of the user interface through which the student enters the answers to the ordering relations and alpha algorithm steps. There is a brief overview of the task editing page before explaining the concept in detail.
The input page is composed of two blocks. The first section contains general information, such as the task description. The second component serves as the task processing framework. The student can select the feedback level, the points earned are presented, the relevant event log is displayed, and the input fields are arranged one after the other (see Appendix B, Figure 9). First, there are three fields for entering the solution to each stage of the ordering relations. At this point it is important to mention, that the creation of an input field for relation $\#_W$ was purposefully omitted for two reasons. Firstly, this relationship is implied by the proper solution of the other three relations. The second reason is that the input is prone to errors. Since this connection yields the most combinations and the error susceptibility with input containing 20 pairs or more is very high, it was decided to skip this step deliberately. This fact, however, should be mentioned in the task description.
Following that, there are seven input fields where the student can enter the solution to the respective step of the alpha algorithm. This section concludes with two buttons, one for submission and the other for diagnosing the solution entered. Furthermore, each input field has a label with the relevant step description and an information circle indicating the expected syntax of the response to be entered. This information is available in both English and German. Once the task is either submitted or diagnosed, the feedback is displayed according to the feedback level at the end of the page.
Two angular components are required to display a process mining exercise to students. One component (parent), 'student-task.component', already exists and is implemented in the structure; however, it must be adapted to accommodate the process mining task type. The second component (child), 'pm.assignment.component', must be implemented in the context of this work. This component is embedded as a child component with a separate '<div>' element in the parent's 'student-task.component' HTML file. The parent component provides certain attributes. Angular's lifecycle method *ngOnInit()* is used to update these attributes every time the component is initialized. These updates are based on the HTTP requests that have already been defined in the 'student-service.ts' file.
The maximum points defined in the configuration parameter component or, if available, the attribute corresponding to an already submitted solution are among the variables that the 'PmAssignmentComponent' receives as input from the 'StudentTaskComponent'. These and the other variables that are taken as input from the 'StudentTaskComponent' are declared in the corresponding HTML file's '<div>' element.

Any attributes utilized in the appropriate HTML file, as well as all attributes taken over as input from the 'StudentTaskComponent' are declared in the 'PmAssignmentComponent' typescript template. This component's key concept will be explained below.
Two Angular lifecycle methods are used: on the one hand *ngOnInit()* and on the other hand *ngAfterContentChecked()*. The first one is used to display the corresponding event log for the exercise as soon as the student opens it. In order to obtain the log information, a new method *getPmLogForIndividualTask(. . .)* is implemented in the 'assignment.service.ts' file that requests the corresponding log. As a consequence, this method sends an HTTP get request to the platform's back end. The dispatcher returns the required log information in the form of a two-dimensional string array wrapped in a log DTO, which can be processed further.
The second method is mainly used to display a previously entered submission in the appropriate input fields. This applies only if a submission is available and does not apply if the task has never been submitted. As previously stated, the variable *submission* is an '@Input' variable and was thus already queried in the parent component. A submission is stored in the RDF database as a single JSON string (see Section 5.2.2), which must be converted first. As a result, a method for converting the JSON string back to a map has been implemented. The values that result can then be assigned to the appropriate input field variables.
Another important method in this component is *processSubmission()* (see Listing 5.1). It handles the student's submitted solution. This comprises creating a submission DTO, sending it to the dispatcher back end for analysis, and processing the results of the analysis. Only the modifications will be presented here since this method is already known from another component (assignment.component.ts).

Listing 5.1: Process Mining Component (from pm.assignment.component.ts)

```
1   private processSubmission(): void {
2     this.diagnoseLevel = this.mapDiagnoseText(this.diagnoseLevelText);
3     const submissionDTO = this.initializeSubmissionDTO();
4
5     this.assignmentService.postSubmission(submissionDTO).subscribe(
6         submissionId => {
7       this.submissionIdDTO = submissionId;
8       this.submissionDTO.submissionId = submissionId.submissionId;
9       setTimeout(() => {
10        this.getGrading();
11      }, 2000);
12    });
13  }
```

In line 3 of Listing 5.1, the submission DTO is declared and initialized. This DTO object, which serves as a container for data transfer between processes, contains the variables that the process mining evaluator described in Section 4.2.3 requires. The method *initializeSubmissionDTO()* sets all essential attributes, including the variable *passedAttributes*, which holds the students' submitted responses for each stage.
Furthermore, once the submission DTO is initialized, it is posted to the dispatcher back end. Therefore, an HTTP post request is sent to the platform back end, which is implemented in the method *postSubmission(. . .)* (see Listing 5.1, line 5). Section 5.2.2 describes how this request is

further processed.
If a grading is received, it is processed further (see Listing 5.1, line 10). Two front end related elements are examined in this procedure. On the one hand, the received grading is used to determine whether or not the submitted submission had errors, through checking if the maximum dispatcher points (up to 14 points) were obtained. If this is not the case, the response was incorrect and the submission is incorrect. This boolean attribute determines whether or not feedback is displayed in the user interface.
If, on the other hand, the task has been 'submitted', the points earned must be calculated again in order to be displayed directly. This appears to be redundant at first glance, as it is done on the platform back end (see Section 5.2.2), but it has a purpose in the eTutor's present framework. When an assignment is submitted, the points earned are immediately calculated and saved on the platform's back end. However, these would not be visible in the user interface until the task would be called up again. As a result, in order to provide an immediate display, one must make a workaround and calculate the points redundantly in the front end. The actual points earned are then calculated using the highest feedback level ever set, the diagnosis level weighting, and the dispatcher's points. Section 5.2.2 covers this in full.

The component's HTML template is composed of a number of '<div>' elements, each of which defines a different element on this page. The event log that is being shown is represented as a table whose size changes depending on how many traces are present. A trace from the event log is represented by one row of the table (see Appendix B, Figure 9). Each input field is specified by the HTML '<input>' element. Furthermore, each input field must allow for both a new input and the indication of a previously entered solution for the field, similar to the configuration parameter input form. This is accomplished in this case using the two-way binding with Angular's 'ngModel'. This makes it possible to import values that have already been typed into the input field, but it also updates the values whenever a new input is made. In addition, once a student requests a grading on an exercise and the required grading arrives in the platform front end, a notice of success or failure is delivered, along with the required feedback. The Angular component 'ngIf' ensures that these fields are not visible if no grading is requested.

5.2.2 Back End Implementation

The back end of the platform is described and explained in this section. It serves primarily to handle the interfaces between the platform's front end and back end as well as between the platform and the dispatcher's back end in the context of this work. Additionally, in order to save the information in the RDF graph database, all relevant information (such as student, submission, and course-related information) are handled appropriately. The structure of this section is as follows: in order to obtain an understanding of the data flow, the implemented solution strategy for handling configuration parameters is provided in detail first. The following describes a concept for managing incoming HTTP requests based on the input form for exercises.
Furthermore, the flow of the descriptions of the implemented methods corresponds to the data flow as it occurs in practice.

Configuration Parameter

Several procedures must be completed in order to store the configuration parameters defined by the tutor in both the relational and graph databases. The method *saveNewTask(task: INewTaskModel)* implemented in 'task.service.ts' concludes Section 5.2.1. This sends a request from the front end of the platform to the back end of the platform. The request is handled by the already defined REST endpoint *createNewTaskAssignment(. . .)* in the platform's back end class 'TaskAssignmentResource'.
Here, two paths are taken. For the first path, a method *createTask(. . .)* in class 'DispatcherProxyService' is called, given the information wrapped in a DTO. This method ensures that the data is delivered to the dispatcher's back end. For the second path, the passed parameters are saved in the RDF database. For this a method is called, which is implemented in class 'AssignmentSPARQLEndpointService'. Both paths will be briefly described below. At this point, it is important to note that the already defined methods that will be mentioned here have the word 'Task' in their name. This means that the information to be processed will contribute to the storage of an actual task. These methods will be used in the context of this work, but not to save a task, but to save the configuration parameters for a task generation.

Method *createTask(. . .)*, which is implemented in class 'DispatcherProxyService', is already part of the eTutor's structure, but it was extended during this work by the method *handlePmTaskConfigCreation(newTaskAssignmentDTO)*, which specifically handles the storage of a process mining task configuration. The aim of this function is to provide the configuration parameters to the next method (*createPmTaskConfiguration(newTaskAssignmentDTO)*), and to assign the dispatcher id received, to the object *newTaskAssignmentDTO*. The dispatcher id obtained corresponds to the relational database's unique id corresponding to the stored configuration parameter. Method *createPmTaskConfiguration(newTaskAssignmentDTO)* (see Listing 5.2) is essentially a transitional step that handles the HTTP response of the interface between the platform back end and the dispatcher back end while also calling this interface (line 5-6).

Listing 5.2: Service Method (from DispatcherProxyService.java)

```
1     private int createPmTaskConfiguration(NewTaskAssignmentDTO
         newTaskAssignmentDTO) throws DispatcherRequestFailedException{
2         // get PmExerciseConfigDTO required by the dispatcher to create
             the configuration
3         var pmExerciseConfigDTO =
             getPmExerciseConfigDTOFromTaskAssignment(newTaskAssignmentDTO);
4         // Proxy request to dispatcher
5         var response = proxyResource.createPmExerciseConfiguration(
6                 pmExerciseConfigDTO);
7         // return dispatcher -id of the exercise configuration
8         return response.getBody() != null? response.getBody() : -1;
9     }
```

The *NewTaskAssignmentDTO* object must first be converted to an appropriate DTO with the relevant configuration parameters, as given in Listing 5.2, line 3. This DTO is then supplied to a function in class 'DispatcherProxyResource' (line 5) that sends an HTTP put request to the dispatcher back end. This method serves as the interface between the two back ends. The methods' request is processed in the dispatcher back end with an appropriate method in the REST controller, and the configuration parameters are stored in the relational database. The HTTP response is an integer that corresponds to the task configuration's id in the relational database.

The second path, as mentioned earlier, is to save the required configuration parameter in the RDF graph database. This is accomplished by executing the method *insertNewTaskAssignment(. . .)* in the class 'AssignmentSPARQLEndpointService'. Because this is already part of the structure and implemented, the only thing left to do was to add the configuration parameters to the function *constructTaskAssignmentFromDTO(. . .)*, which stores the data.
Finally, the approach for updating and deleting an exercise configuration is the same as for adding one, and the methods are implemented in the same classes.

Task Editing

Since the back end of this user interface is fairly large and is made up of numerous already implemented methods, this chapter should only focus on the components that have been updated or newly implemented.
The process of establishing a new exercise for the student to complete will be described first. For this, method *insertNewAssignedTask(. . .)* implemented in class 'StudentService' is highlighted. The moment a student opens an exercise sheet, this method is invoked. A new task is produced for the student the moment this method is called. The method had to be adapted accordingly for a process mining task.
First, the RDF graph must be queried for the id of the configuration assigned to this exercise. This is accomplished using a method implemented in the class 'AssignmentSPARQLEndpointService' for this purpose. The id is then supplied to the service method in the 'DispatcherProxyService' class that corresponds to it. This, in turn, invokes the appropriate method in the 'DispatcherProxyResource' class, which sends an HTTP get request to the dispatcher back end. In the dispatcher back end the REST controller handles this request. In Section 4.2.5 the appropriate request handling method has been already described. This procedure generates the matching id, which can be used to identify the task in the relational database. In the RDF graph, this id is stored alongside the associated variables. For this purpose a method implemented in the class 'AssignmentSPARQLEndpointService' is used. The result of this process is a task that is displayed to the student.

The next process concerns querying the event log that is displayed as part of the task processing (see Section 5.2.1). The platform front end request is handled by the newly built method *getLogToCorrespondingExerciseId(. . .)* in the platform back end class 'StudentResource'. First, the dispatcher-assigned id of the relevant exercise must be queried. A method in the 'StudentService' class accesses the RDF graph database and returns the corresponding id. This id is passed to a method in the 'DispatcherProxyService' class, which calls the appropriate method in the 'DispatcherProxyResource' class. Based on the specified exercise id, this method initiates an HTTP get request to the dispatcher back end and obtains the appropriate event log data. Again, this request is handled by one of the defined request handling methods in the dispatcher back end's REST controller. The HTTP response in the form of a JSON string is then mapped to the appropriate DTO. The result of this process is a DTO containing the log information which can be further processed in the front end.

Method *handleDispatcherUUID(. . .)*, which is implemented in the 'StudentResource' class, is the last essential request handling method implemented in the platform back end that is highlighted in this section. This method processes a dispatcher-provided submission and grading as a result of a student's submission of an individual exercise assignment. This method was modified in the context of a process mining exercise. These modifications primarily concern exercise grading and are discussed in the following.

Listing 5.3: Points Calculation (from StudentResource.java)

```java
        }else{  // isPmTask
            if(points == 0
                && (!studentService.isTaskSubmitted(courseInstanceUUID,
                    exerciseSheetUUID, matriculationNo, taskNo))
                && submission.getPassedAttributes().get(
                        "action").equals("submit")
                && grading.getMaxPoints() != 0
            ){
                var diagnoseLevelWeighting =
                    weightingAndMaxPointsIdArr[0];

                if(highestDiagnoseLevel == 3){
                    points = grading.getPoints() - (3 *
                        highestDiagnoseLevel * diagnoseLevelWeighting);
                    if(points < 0){
                        points = 0;
                    }
                }else if(highestDiagnoseLevel == 2){
                    points = grading.getPoints() - (highestDiagnoseLevel
                        * diagnoseLevelWeighting);
                    if(points < 0){
                        points = 0;
                    }
                }else if(highestDiagnoseLevel == 1){
                    points = grading.getPoints() - (highestDiagnoseLevel
                        * diagnoseLevelWeighting);
                    if(points < 0){
                        points = 0;
                    }
                }else{
                    points = grading.getPoints();   // full points
                }


                studentService.setDispatcherPointsForAssignment(
                        courseInstanceUUID, exerciseSheetUUID,
                            matriculationNo, taskNo, points);
                studentService.markTaskAssignmentAsSubmitted(
                        courseInstanceUUID, exerciseSheetUUID,
                            matriculationNo, taskNo);
            }
        }
```

Listing 5.3 illustrates the calculation of points for an exercise.
Points are only calculated if the exercise is 'submitted' and the student has not yet submitted the exercise. The computation is based on the highest selected feedback level and the tutor-defined diagnosis level weighting. As shown in Listing 5.3, if the student selects the highest feedback level, the point decrease should be maximum (line 10), since this level displays the correct solution. A diagnosis level weighting of 1.5, for example, subtracts 13.5 (3x3x1.5) points from the maximum points at diagnosis level three (assumed that the maximum achievable points are 14, as it is also assumed in the dispatcher grading module). If 14 points were earned but the highest diagnosis level was three, only 0.5 points were kept. There will be no points deducted if the student does not select a feedback level. The computed points are then saved in the RDF graph (see Listing 5.3, line 30). Furthermore, the task is marked as 'submitted' in the RDF graph (see Listing 5.3, line 32) and presented with a check mark in the eTutor++ user interface.

However, it should be noted that the grading can be changed in the code if necessary.

6

Conclusion

The design and implementation of a learning module for the alpha algorithm in the eTutor++ are described in this master's thesis. It allows students to apply the alpha algorithm to a variety of randomly generated event logs. Additionally, based on the student's submission, detailed feedback can be generated and presented to the student.

The algorithm's learning module is implemented in the eTutor++'s current framework, including its dispatcher and platform. This learning module supplements the existing structure, which already provides learning modules for specific query languages. The numerous components of this work are defined and characterized in terms of how they are planned, implemented, and interconnected. This covers the alpha algorithm component, the process log generating tool component, the eTutor++'s two existent back ends, and the front end. The main newly implemented classes and methods, as well as the primary changes to the old structure, are presented in this context.

This learning module comes with some benefits. First, the theoretical foundations of process mining and process discovery taught in the university course can be deepened and applied using this learning module. Second, the learning module gives students instant feedback on their performance. This improves learning success and course enjoyment. For the third, the existing eTutor++ is extended and hence thematically expanded. This increases the opportunities to actively use it for additional course assessments. Finally, the studied literature indicates that this learning and active application of an algorithm in this form is unprecedented in the academic field of business informatics.

Future work can extend this learning module. It could be included as part of the Exam Setting to further promote digital education. This would require making the necessary adjustments. Furthermore, a graphical representation of the found process based on the Alpha Algorithm solution might be implemented in the form of a Petri Net. Additionally, the ability to generate event logs could be used for future process mining algorithms.

Bibliography

Ailenei, I., Rozinat, A., Eckert, A., & van der Aalst, W. M. P. (2012). Definition and validation of process mining use cases. In *Business process management workshops* (pp. 75–86). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-28108-2_7

Antonucci, P., Estler, C., Nikolić, D., Piccioni, M., & Meyer, B. (2015). An incremental hint system for automated programming assignments. *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. https://doi.org/http://dx.doi.org/10.1145/2729094.2742607

Augusto, A., Conforti, R., Dumas, M., Rosa, M. L., Maggi, F. M., Marrella, A., Mecella, M., & Soo, A. (2019). Automated discovery of process models from event logs: Review and benchmark. *IEEE Transactions on Knowledge and Data Engineering*, *31*(4), 686–705. https://doi.org/10.1109/TKDE.2018.2841877

Avancena, A. T., Kondo, C., & Nishihara, A. (2013). Design and assessment of an algorithm learning tool for high school computer science. *Proceedings of 2013 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE)*. https://doi.org/10.1109/TALE.2013.6654521

baeldung. (2022). The spring @controller and @restcontroller annotations. https://www.baeldung.com/spring-controller-vs-restcontroller

Burattin, A. (2016). Plg2: Multiperspective process randomization with online and offline simulations. *BPM (Demos)*, 1–6.

Burratin, A. (2015). *Plg2: Multiperspective processes randimization and simulation for online and offline settings* (tech. rep.). CoRR. abs/1506.08415

Chinosi, M., & Trombetta, A. (2012). BPMN: An introduction to the standard. *Computer Standards and Interfaces*, *34*(1), 124–134. https://doi.org/10.1016/j.csi.2011.06.002

Garcia, R., Falkner, K., & Vivian, R. (2018). Systematic literature review: Self-regulated learning strategies using e-learning tools for computer science. *Computers and Education*, *123*, 150–163. https://doi.org/https://doi.org/10.1016/j.compedu.2018.05.006

Görner, F. (2022a). Dispatcher, pm modul. *GitHub Repository*. https://github.com/eTutor-plus-plus/dke-dispatcher/tree/feature/process-mining

Görner, F. (2022b). Platform, pm modul. *GitHub Repository*. https://github.com/eTutor-plus-plus/platform/tree/feature/processmining

John E. Hopcroft, J. D. U., Rajeev Motwani. (2011). *Einführung in automatentheorie, formale sprachen und berechenbarkeit* (3rd ed.). Pearson Studium.

Keuning, H., Jeuring, J., & Heeren, B. (2016). Towards a systematic review of automated feedback generation for programming exercises. *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. https://doi.org/http://dx.doi.org/10.1145/2899415.2899422

Odekirk-Hash, E., & Zachary, J. L. (2001). Automated feedback on programs means students need less help from teachers. *ACM SIGCSE Bulletin*, *33*(1), 55–59. https://doi.org/https://doi.org/10.1145/366413.364537

Reijers, H. A. (2021). Business process management: The evolution of a discipline. *Computers in Industry*, *126*, 103404. https://doi.org/https://doi.org/10.1016/j.compind.2021.103404

Robinson, P. E., & Carroll, J. (2017). An online learning platform for teaching, learning, and assessment of programming. *2017 IEEE Global Engineering Education Conference (EDUCON)*. https://doi.org/10.1109/EDUCON.2017.7942900

Tiwari, A., Turner, C., & Majeed, B. (2008). A review of business process mining: State-of-the-art and future trends. *Business Process Management Journal*, *14*(1), 5–22. https://doi.org/10.1108/14637150810849373

van der Aalst, W., Weijters, T., & Maruster, L. (2004). Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, *16*(9), 1128–1142. https://doi.org/10.1109/tkde.2004.47

van der Aalst, W. (2012). Process mining: Overview and opportunities. *ACM Transactions on Management Information Systems*, *3*(2), 1–17. https://doi.org/10.1145/2229156.2229157

van der Aalst, W. (1997). Verification of workflow nets. In *Application and theory of petri nets 1997* (pp. 407–426). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-63139-9_48

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., & Barros, A. (2003). Workflow patterns. *Distributed and Parallel Databases*, *14*(1), 5–51. https://doi.org/10.1023/a:1022883727209

van der Aalst, W., & Weijters, A. (2004). Process mining: A research agenda. *Computers in Industry*, *53*(3), 231–244. https://doi.org/10.1016/j.compind.2003.10.001

vom Brocke, J., Jans, M., Mendling, J., & Reijers, H. A. (2021). A five-level framework for research on process mining. *Business and Information Systems Engineering*, *63*(5), 483–490. https://doi.org/10.1007/s12599-021-00718-8

Wang, L., & Wei, H. (2021). A website design to support teaching of algorithms. *2021 IEEE International Conference on Consumer Electronics and Computer Engineering (ICCECE)*. https://doi.org/10.1109/ICCECE51280.2021.9342224

Xu, S., & Chee, Y. S. (2003). Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, *29*(4), 360–384. https://doi.org/10.1109/TSE.2003.1191799

Zerbino, P., Stefanini, A., & Aloini, D. (2021). Process science in action: A literature review on process mining in business management. *Technological Forecasting and Social Change*, *172*, 121021. https://doi.org/10.1016/j.techfore.2021.121021

Appendix A

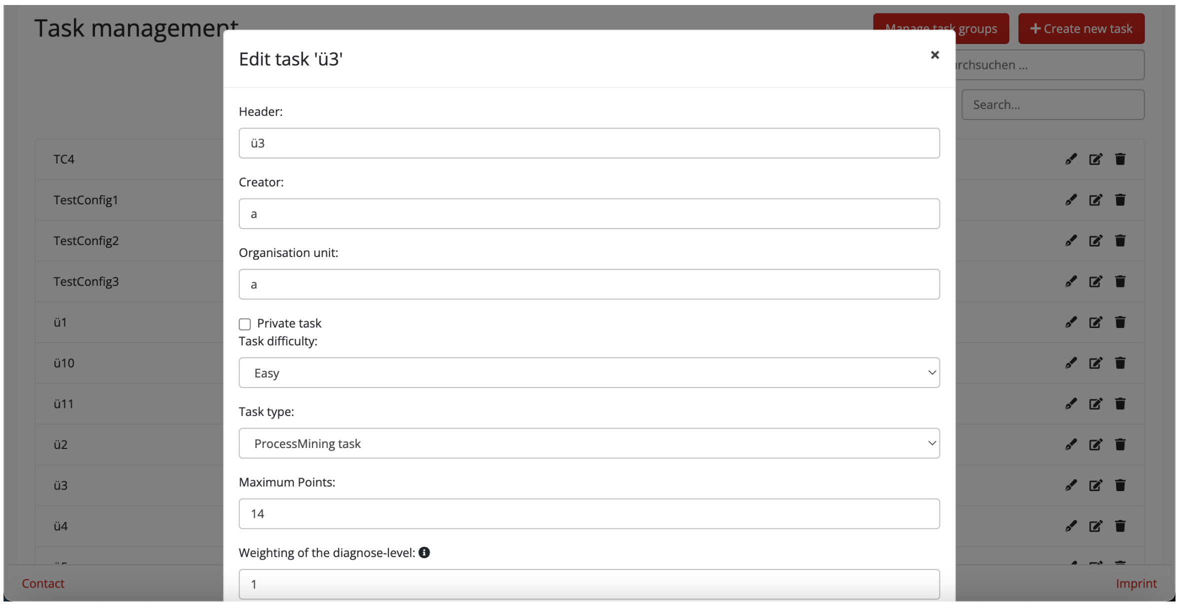
Table 1: Example Configuration parameter PLG2

Parameter	config1	config2	config3	default
ANDBranches	3	2	3	5
XORBranches	2	3	3	5
loopWeight	0.1	0.2	0.2	0.1
singleActivityWeight	0.2	0.1	0.3	0.2
skipWeight	0.0	0.0	0.0	0.1
sequenceWeight	0.3	0.4	0.2	0.7
ANDWeight	0.3	0.2	0.2	0.3
XORWeight	0.2	0.3	0.3	0.3
maxDepth	3	3	3	3
dataObjectProbability	0.0	0.0	0.0	0.0

Appendix B

The steps in the eTutor++ that are necessary to show the students a concrete task and give them the ability to perform it appropriately are presented in the following. The aim is to clarify and illustrate the previously explained steps of the implementation using a concrete example. In order to explain the required activities, the teacher's perspective is used on the one hand and the student's perspective on the other.

The teacher must complete the following steps first. The procedure starts with the creation of a task configuration. The teacher must establish a new 'task' in the eTutor++ task management area. The term appears to be confusing, however this is due to the current structure of the eTutor++. In the resulting dialog box, general information is first requested (see Figure 1), followed by the task type selection. If 'ProcessMining task' is selected in this field, the dialog box



TC4
TestConfig1
TestConfig2
TestConfig3
ü1
ü10
ü11
ü2
ü3
ü4

Edit task 'ü3'

Header: ü3

Creator: a

Organisation unit: a

☐ Private task

Task difficulty: Easy

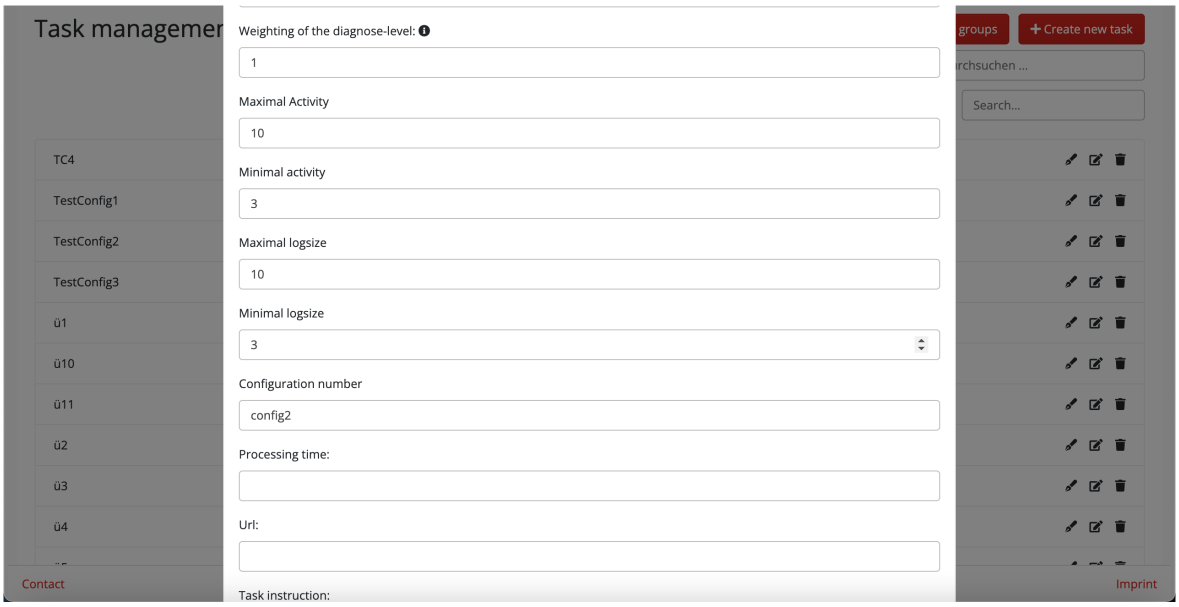
Task type: ProcessMining task

Maximum Points: 14

Weighting of the diagnose-level: 1

Figure 1: Example Task configuration parameter page (1) - eTutor++

expands and fields specific to this task type appear. Specific and general parameters are both queried. 'Maximum Points' and 'Weighting of the diagnose-level' are general parameters. As previously explained in Section 4.2.3, it is recommended to pass the value '14' as the maximum number of points. The input fields specific for this task type are 'Maximal activity', 'Minimal activity', 'Maximal logsize', 'Minimal logsize' and 'Configuration number' (see Figure 2). These input fields must be filled out in accordance with the teacher's expectations regarding the event log's complexity. When it comes to the input field 'Configuration number', there are three options: 'config1', 'config2', or 'config3' (see Section 4.2.2). The description of the task is entered in the


Task managemer

Weighting of the diagnose-level:
1

Maximal Activity
10

Minimal activity
3

Maximal logsize
10

Minimal logsize
3

Configuration number
config2

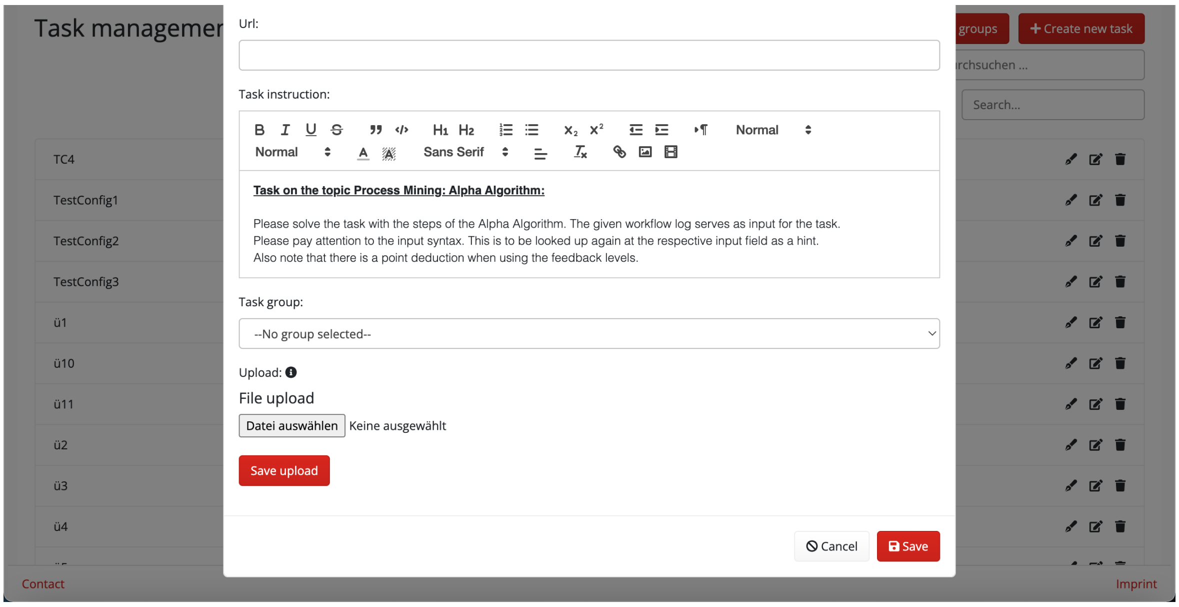
Processing time:

Url:

Task instruction:

Figure 2: Example Task configuration parameter page (2) - eTutor++

'Task instruction' input field, and it is displayed accordingly during task processing (see Figure 3).


Task managemer

Url:

Task instruction:

Task on the topic Process Mining: Alpha Algorithm:

Please solve the task with the steps of the Alpha Algorithm. The given workflow log serves as input for the task.
Please pay attention to the input syntax. This is to be looked up again at the respective input field as a hint.
Also note that there is a point deduction when using the feedback levels.

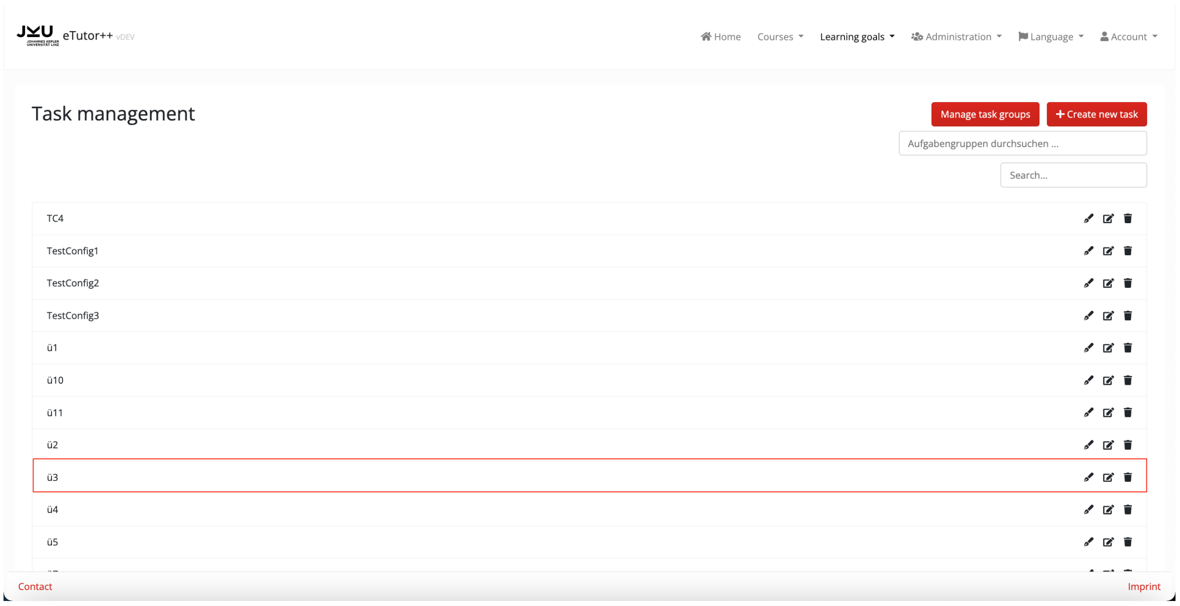
Task group:
--No group selected--

Upload:
File upload
Datei auswählen Keine ausgewählt

Figure 3: Example Task configuration parameter page (3) - eTutor++

This process can be repeated several times to generate a set of task configurations that will be used later in the task creation process. Figure 4 depicts an overview of a large number of task configurations. The task configuration 'ü3' will be utilized in this example, which is why it is framed in red in Figure 4.

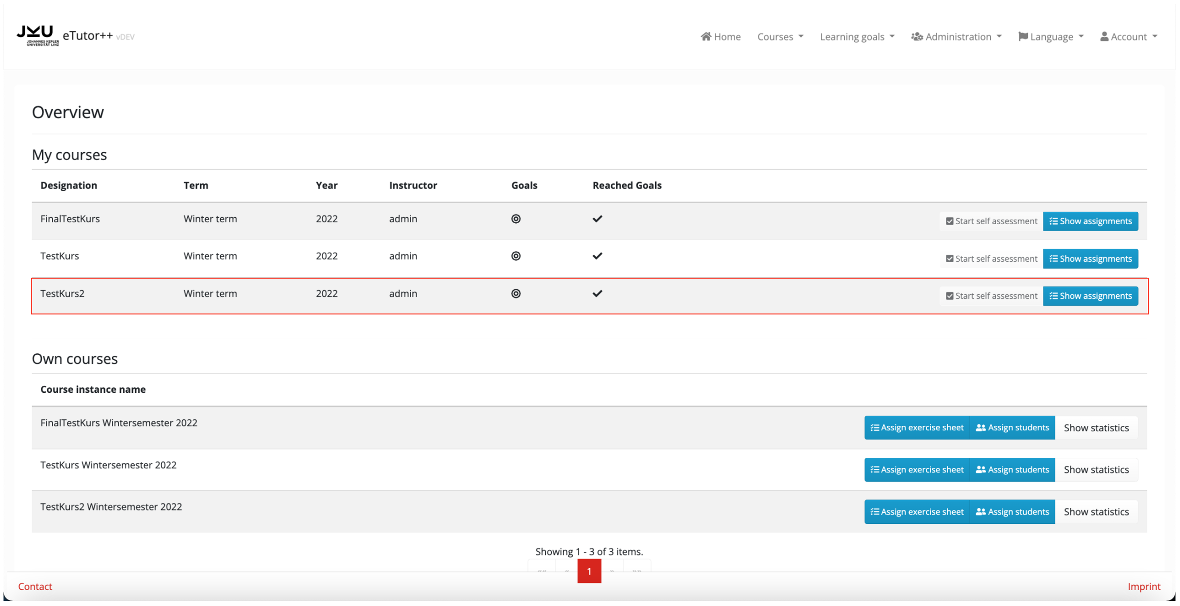
The next step in the process for the teacher is to create a 'course'. A course contains the various practice sheets, and students are added to this course in order to work on tasks. Figure 5 illustrates a teacher's (and also the student's) perspective on the courses to which the student has been assigned. The course 'TestKurs2' is utilized in this example, hence it is framed in red. After that,



Task management

TC4
TestConfig1
TestConfig2
TestConfig3
ü1
ü10
ü11
ü2
ü3
ü4
ü5

Figure 4: Example Task configuration parameter page overview - eTutor++



Overview

My courses

Designation	Term	Year	Instructor	Goals	Reached Goals
FinalTestKurs	Winter term	2022	admin	◎	✓
TestKurs	Winter term	2022	admin	◎	✓
TestKurs2	Winter term	2022	admin	◎	✓

Own courses

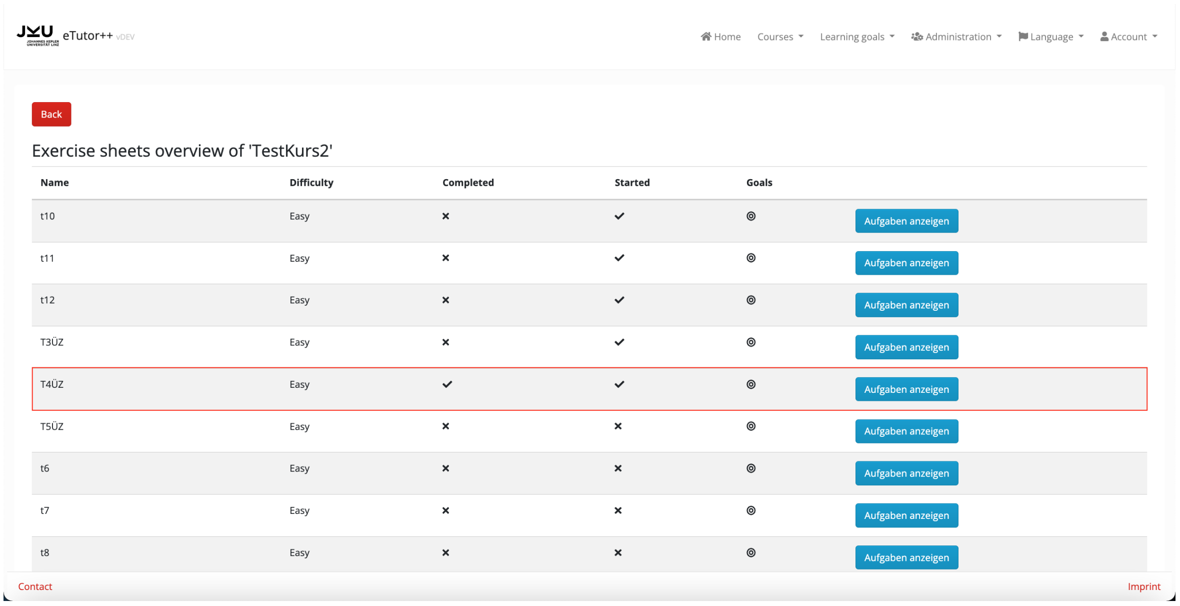
Course instance name
FinalTestKurs Wintersemester 2022
TestKurs Wintersemester 2022
TestKurs2 Wintersemester 2022

Showing 1 - 3 of 3 items.

Figure 5: Example Course overview page - eTutor++

the teacher creates an exercise sheet. The exact amount of tasks on an exercise sheet is chosen by the teacher when the exercise sheet is created. During the procedure, multiple exercise sheets can be created. The relevant exercise sheets are allocated to the desired course as soon as this process is done. Figure 6 depicts an example of this. It shows that the course 'TestKurs2' has received a large number of exercise sheets for testing purposes. This is also a representation of the student's point of view. With this final step, the teacher has completed all of the essential processes so that a student can access an exercise sheet and, as a result, work on tasks. The exercise sheet 'T4ÜZ' is considered further in this example, which is why it is framed in red in Figure 6.

Once a student is enrolled to a course with a process mining learning objective, they can access the relevant exercise sheets, as shown in Figure 6. In this scenario, a student has been assigned to course 'TestKurs2' and is expected to complete exercise sheet 'T4ÜZ'. When the student selects 'Aufgaben anzeigen' (show tasks) on the appropriate exercise sheet, a new task is generated based

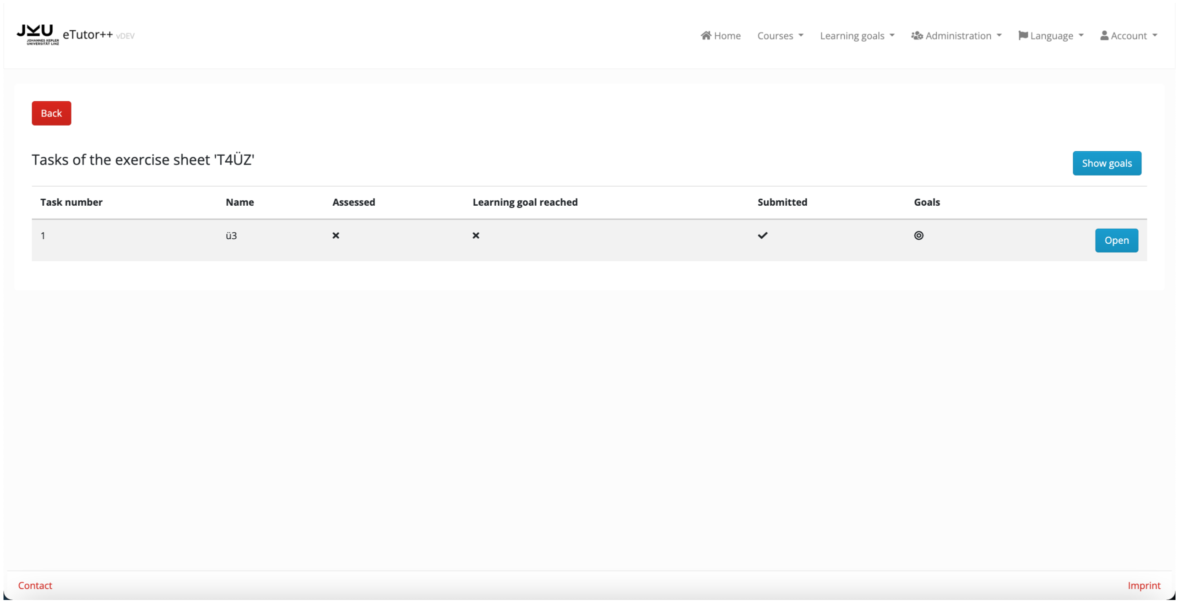


Exercise sheets overview of 'TestKurs2'

Name	Difficulty	Completed	Started	Goals	
t10	Easy	×	✓	◎	Aufgaben anzeigen
t11	Easy	×	✓	◎	Aufgaben anzeigen
t12	Easy	×	✓	◎	Aufgaben anzeigen
T3ÜZ	Easy	×	✓	◎	Aufgaben anzeigen
T4ÜZ	Easy	✓	✓	◎	Aufgaben anzeigen
T5ÜZ	Easy	×	×	◎	Aufgaben anzeigen
t6	Easy	×	×	◎	Aufgaben anzeigen
t7	Easy	×	×	◎	Aufgaben anzeigen
t8	Easy	×	×	◎	Aufgaben anzeigen

Figure 6: Example Exercise sheet page overview - eTutor++

on one of the previously defined task configuration parameter. Section 5.2.2 explains how this is implemented. As a result of this activity, a task for this exercise sheet is displayed. As shown in the column 'Name' in Figure 7, a task was generated based on the task configuration 'ü3'. Only one task is displayed at first; once the student has completed it, and the task sheet contains several tasks, the next, newly generated task is displayed. As mentioned in Section 4.2.5, the



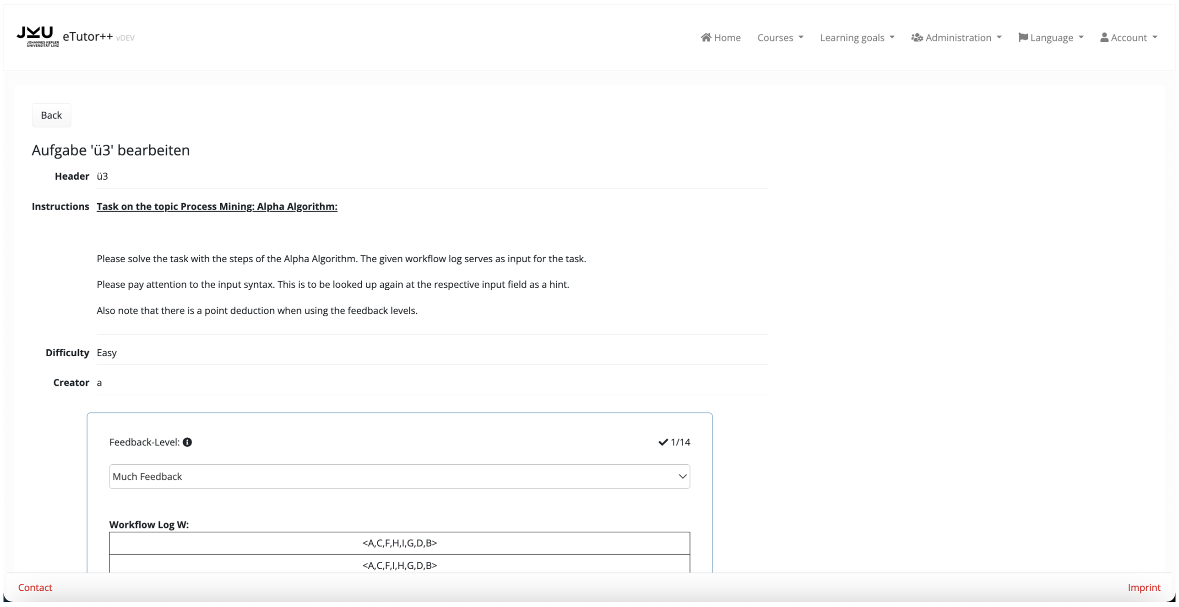
Tasks of the exercise sheet 'T4ÜZ'

Task number	Name	Assessed	Learning goal reached	Submitted	Goals	
1	ü3	×	×	✓	◎	Open

Figure 7: Example Exercise sheet page - eTutor++

newly generated task is stored in the relational database and can be edited by the student. In order to do so, the student must click on the associated task's 'Open' button, as indicated in Figure 7. This action will open the task editing page (see Figure 8, Figure 9 and Figure 10).

The task description, which was defined by the teacher at the start of the process, can be found on the task processing page (see Figure 8). The students can select the level of feedback that they want to receive. There are four feedback levels to select from: no feedback, little feedback, more feedback, and much feedback. The event log is then shown (see Figure 9), with each line



Aufgabe 'ü3' bearbeiten

Header ü3

Instructions **Task on the topic Process Mining: Alpha Algorithm:**

Please solve the task with the steps of the Alpha Algorithm. The given workflow log serves as input for the task.

Please pay attention to the input syntax. This is to be looked up again at the respective input field as a hint.

Also note that there is a point deduction when using the feedback levels.

Difficulty Easy

Creator a

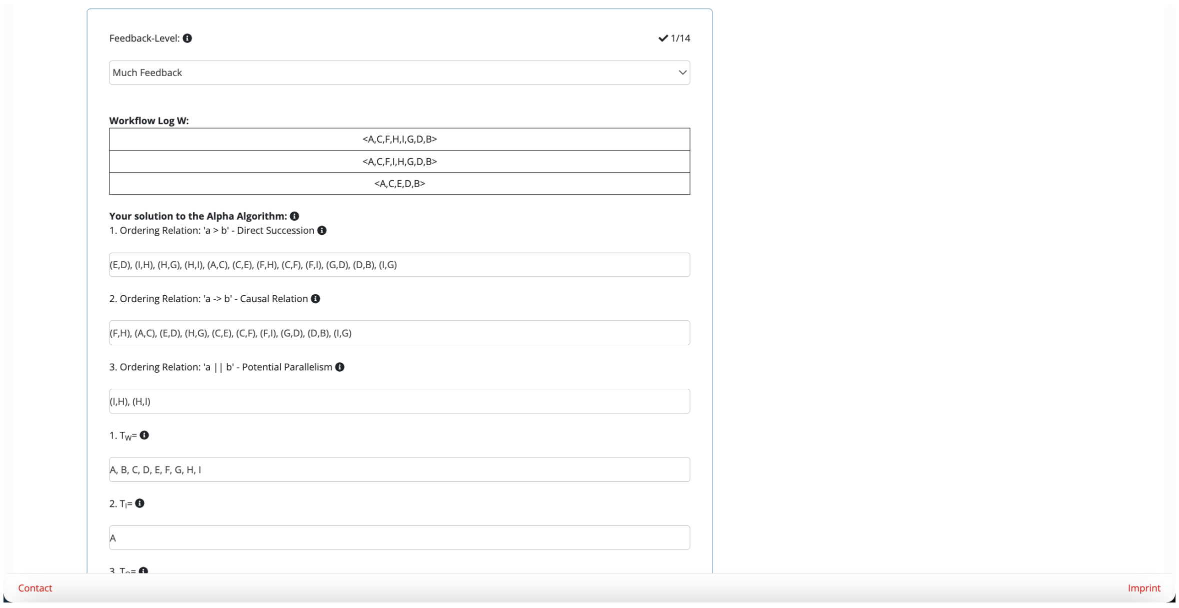
Feedback-Level: ✓ 1/14

Much Feedback

Workflow Log W:

<A,C,F,H,I,G,D,B>
<A,C,F,I,H,G,D,B>

Figure 8: Example Task editing page (1) - eTutor++


Feedback-Level: ✓ 1/14

Much Feedback

Workflow Log W:

<A,C,F,H,I,G,D,B>
<A,C,F,I,H,G,D,B>
<A,C,E,D,B>

Your solution to the Alpha Algorithm:
1. Ordering Relation: 'a > b' - Direct Succession

(E,D), (I,H), (H,G), (H,I), (A,C), (C,E), (F,H), (C,F), (F,I), (G,D), (D,B), (I,G)

2. Ordering Relation: 'a -> b' - Causal Relation

(F,H), (A,C), (E,D), (H,G), (C,E), (C,F), (F,I), (G,D), (D,B), (I,G)

3. Ordering Relation: 'a || b' - Potential Parallelism

(I,H), (H,I)

1. T_W=

A, B, C, D, E, F, G, H, I

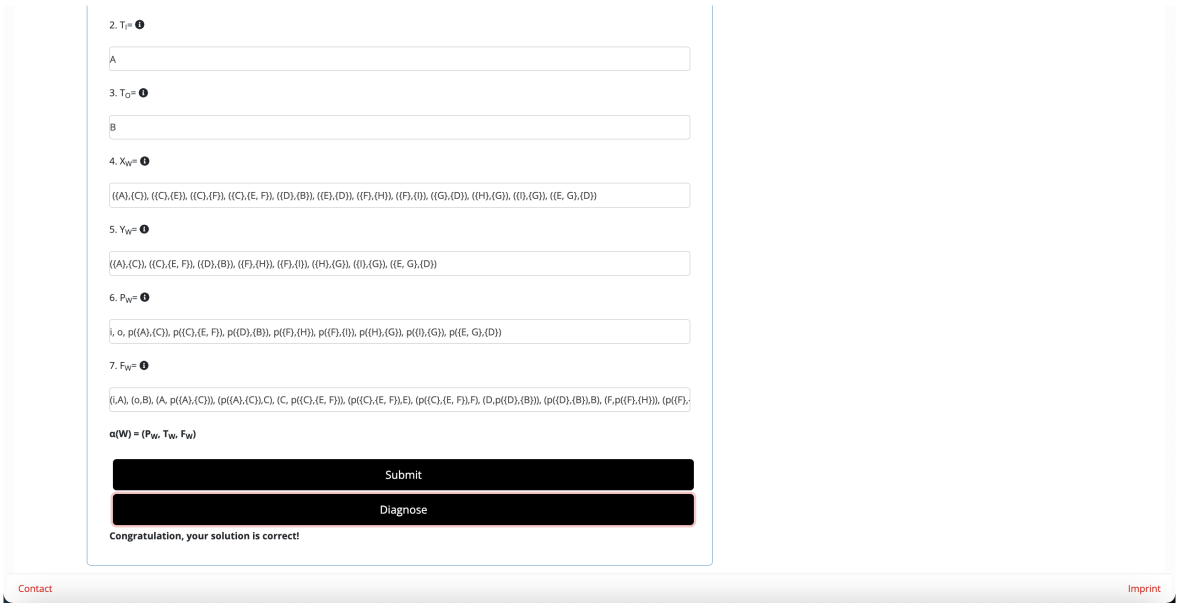
2. T_I=

A

3. T_O=

Figure 9: Example Task editing page (2) - eTutor++

representing a trace of the event log. This log serves as the basis for processing the task. Students enter their solutions to the respective alpha algorithm solution steps in the following input areas. The student can review the anticipated syntax of the input for each input field by pressing the information icon, which is presented as a black circle with an 'i' in the center adjacent to each input field. After entering the solutions, the student must decide whether to 'Submit' or 'Diagnose' them. Points are awarded if the solution is submitted; if it is diagnosed, no points are awarded. Depending on the correctness of the entered answers, either the solution is marked as correct (see Figure 10) or it is indicated as incorrect, and feedback is offered appropriately (see Figure 11). In this example, the student has set the highest feedback level, so that even though the solution was correct, the majority of the 14 possible points were deducted, as seen in Figure 9 and explained in Section 5.2.2.



2. T_I=

A

3. T_O=

B

4. X_W=

({A},{C}), ({C},{E}), ({C},{F}), ({C},{E, F}), ({D},{B}), ({E},{D}), ({F},{H}), ({F},{I}), ({G},{D}), ({H},{G}), ({I},{G}), ({E, G},{D})

5. Y_W=

({A},{C}), ({C},{E, F}), ({D},{B}), ({F},{H}), ({F},{I}), ({H},{G}), ({I},{G}), ({E, G},{D})

6. P_W=

i, o, p({A},{C}), p({C},{E, F}), p({D},{B}), p({F},{H}), p({F},{I}), p({H},{G}), p({I},{G}), p({E, G},{D})

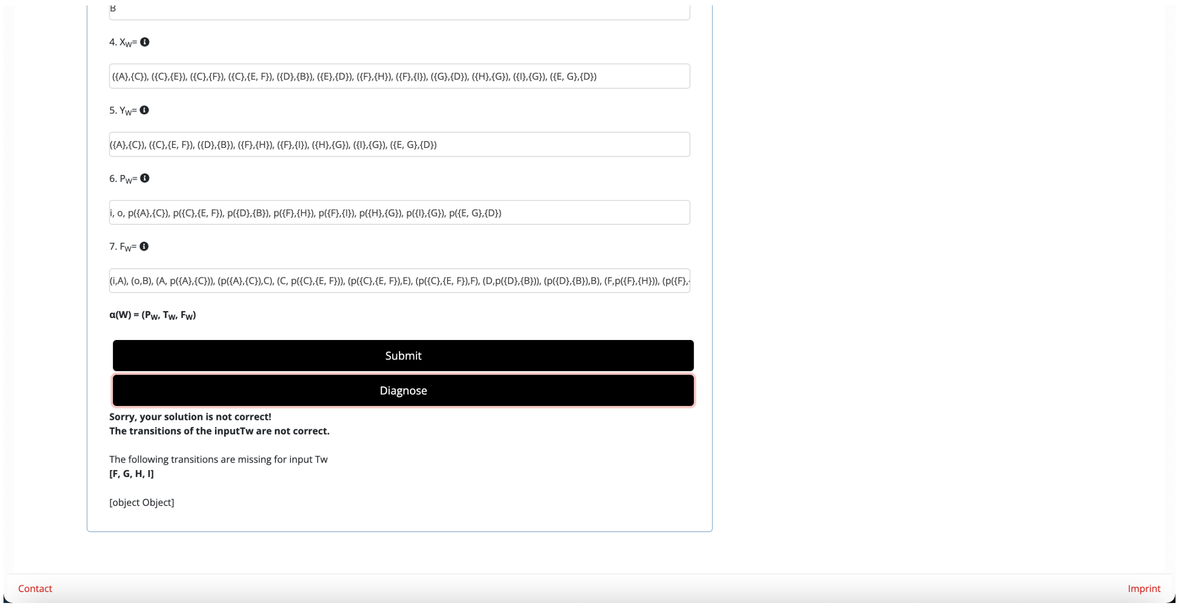
7. F_W=

(i,A), (o,B), (A, p({A},{C})), (p({A},{C}),C), (C, p({C},{E, F})), (p({C},{E, F}),E), (p({C},{E, F}),F), (D,p({D},{B})), (p({D},{B}),B), (F,p({F},{H})), (p({F},

$\alpha(W) = (P_W, T_W, F_W)$

Congratulation, your solution is correct!

Figure 10: Example Task editing page (3) - eTutor++



B

4. X_W=

({A},{C}), ({C},{E}), ({C},{F}), ({C},{E, F}), ({D},{B}), ({E},{D}), ({F},{H}), ({F},{I}), ({G},{D}), ({H},{G}), ({I},{G}), ({E, G},{D})

5. Y_W=

({A},{C}), ({C},{E, F}), ({D},{B}), ({F},{H}), ({F},{I}), ({H},{G}), ({I},{G}), ({E, G},{D})

6. P_W=

i, o, p({A},{C}), p({C},{E, F}), p({D},{B}), p({F},{H}), p({F},{I}), p({H},{G}), p({I},{G}), p({E, G},{D})

7. F_W=

(i,A), (o,B), (A, p({A},{C})), (p({A},{C}),C), (C, p({C},{E, F})), (p({C},{E, F}),E), (p({C},{E, F}),F), (D,p({D},{B})), (p({D},{B}),B), (F,p({F},{H})), (p({F},

$\alpha(W) = (P_W, T_W, F_W)$

Sorry, your solution is not correct!
The transitions of the inputTw are not correct.

The following transitions are missing for input Tw
[F, G, H, I]

[object Object]

Figure 11: Example Task editing page (4) - eTutor++