UNIVERSITY OF TARTU
Institute of Computer Science
MS(SE) Curriculum

Zohaib Ahmed Butt

Programmatic Perturbation of Business Process Models

Master's Thesis (30 ECTS)

Supervisors:
Marlon Dumas
Orlenys Lopez Pintado

Tartu 2021

Programmatic Perturbation of Business Process Models

Abstract:
As the business goals evolve, the actions, decisions, and events that define the related business processes are also modified. In order to capture those changes at the level of process models, previous work has proposed generic standardizations in the form of change patterns, which will allow the process-aware information systems to become flexible for changing business needs. However, the issue is that the suggested formalizations are still at a theoretical stage, and there is no practical solution available for the implementation of change patterns. The outcome of this Master's thesis is a tool called "Process Model Perturbator," which allows the application of change patterns programmatically. As input, the software tool accepts one or multiple BPMN models and a specification of changes to be done in the Process, and it outputs a perturbed process model resulting from applying the specified changes on the input model. The tool is intended to act as a base for the research towards programmatic perturbation, which will provide a framework for further development of other change patterns and possible usage as an engine in optimizers and process-aware information systems.

Keywords:
Business Process Modelling, BPMN, Change Patterns, Change Primitives

CERCS: P170

Äriprotsesside Mudelite Programmiline Häirimine

Lühikokkuvõte:
Äriliste eesmärkide muutumisel muutuvad ka äriprotsessi defineerivad tegevused, otsused ja sündmused. Selliste muudatuste sisseviimise kohta protsessi mudelisse on varasem kirjandus pakkunud üldiseid standardiseeringuid muudatus mustrite näol, mis võimaldab protsessi tundvatel infosüsteemidel olla paindlikud ärivajaduste muutumise korral. Pakutud formaliseeringute puuduseks on aga asjaolu, et nende arendus on teoreetilisel tasemel ja ei eksisteeri praktilist lahendust, mis rakendaks neid muudatus mustreid äriprotsessi mudelile. Käesoleva magistritöö tulem on tööriist nimega "Process Model Peturbator", mis võimaldab muudatus mustrite rakendamist programmiliselt. Nimetatud tarkvaraline tööriist võtab sisendiks ühe või mitu BPMN mudelit ning protsessi juurutatavate muudatuste kirjelduse, ja tagastab muudetud protsessi mudeli, mis on saadud nimetatud kohanduste rakendamise järel. Käesolev tööriist on mõeldud tulevaste teadustööde aluseks, mis tagab raamistiku muudatus mustrite edasiseks arendamiseks. Samuti võib tulemit kasutada ka mootorina protsesside optimeerijates ning protsessi tundvates informatsioonisüsteemides.

Võtmesõnad: Äriprotsesside modelleerimine, BPMN, mustrite muutmine, primitiivide muutmine

CERCS: P170

Table of Contents

Terms and Notations

PIX[1]: Process Improvement Explorer is an initiative led by Professor Marlon Dumas to introduce a new generation of business model optimization tools that optimize based on the insights derived from data instead of heuristics.

DSL: Domain Specific Language: A language understandable for a computer defined for a specific domain.

BPMN: Business Process Model and Notation: Standard for graphical notation of Business Process Diagrams

GUI: Graphical User Interface: A user interface that allows the users to interact with the computer using graphical entities such as icons, point-and-touch mechanisms.

CLI: Command Line Interface: A user interface that allows the users to interact with the computer using a text-based mechanism in the form of commands.

PAIS: Process-Awareware Information Systems: Information system that manages and executes operational processes involving people, applications, and information sources based on process models **[5].**

[1] https://sep.cs.ut.ee/Main/PIX

1 Introduction

A business process is a sequence of tasks, events, or decisions that must be performed in a specific order based on conditions to achieve particular business goals **[4]**.

Based on different business processes in different domains, Process-aware information systems (PAIS) provide technical support for the execution of processes. PAIS can either be dependent or independent of single or multiple business domains. One of the most recent Estonian startup unicorns, Pipedrive, is an excellent example of a PAIS.

When an organization wants to improve a business process in the long term, it is not feasible to define a business process in vague formats, such as a textual sequence of steps or pseudocode. It is because unclear and loosely defined interpretations of business process specification will have multiple meanings and will only be understandable by a specific group of people who have been working in that specific process domain. Therefore, to represent the processes uniformly, a standard **[4]** known as Business Process Model and Notation (BPMN) is used.

As far as computational aspects are concerned, Business Process Models are usually defined as BPMN diagrams stored in BPMN files, which is essentially an extension of the XML format with its namespace that holds all the valid tags and attributes. It is easily possible to model and modify the BPMN diagrams via GUI with software like **Camunda Modeler**, **BPMN.io,** and **Apromore**.

Changes in existing business processes occur with varying requirements of the businesses due to modifications in the business objectives. For example, the changes in business processes might involve adding new tasks, removing existing tasks, ensuring that a specific subset of tasks is executed conditionally, or performing a subset of tasks in a parallel manner. Incorporating such changes might be possible to allow the existing business process to achieve new goals. It is also possible to apply such changes in order to improve the efficiency of business processes. Similarly, as the business process changes, then the process-aware information systems dependent on the modified business processes also require to be updated.

Currently, the issue is that even though the tools are available to design and define business processes in a digital format, current software allows modifying the business process manually. Moreover, these GUI-based tools only offer the basic operations to add or remove sequence flows, nodes, or subprocesses. These operations are termed as **change primitives [1]**. Due to this reason, the manual effort is duplicated when repetitively applying a standardized set of change primitives to the business processes, known as **change patterns [1].**

The lack of a tool that can implement change primitives and change patterns without the need for manual effort **[1]** acts as a roadblock to developing a completely flexible PAIS that allows the modifications in the underlying Process with the help of an API call or a CLI command. Apart from that, if it were possible to modify the business process programmatically, it would also help solve many optimization problems requiring the structure of the business process to be updated. Furthermore, packaging the tool as a CLI instead of an API allows avoiding the cost of scaling and maintainability.

Moreover, even if there is a command-line solution available to modify existing BPMN diagrams, usability would be decreased in the case of multiple changes, as far as user experience is concerned. Eventually, the CLI will not be an optimized solution if there is a need to make multiple changes in many processes quickly. Therefore, along with the CLI to be used by humans and client software alike, a Domain Specific Language (DSL) can be defined to support a significantly large number of changes in a quicker fashion. Furthermore, by using DSL, the user can easily define all the changes in a language that is understandable by humans.

Therefore, to provide the solution to the gaps mentioned above, the goal of this research topic was to provide a software solution that allows implementation of a subset of change patterns in business processes, which are a standardized way of making modifications in business processes for PAIS. Thus, the tool will allow the domain-specific PAIS to be highly flexible on an implementational level as the software will update its underlying processes according to the business needs. Moreover, the software will also act as a base from where research to provide solutions for the complete subset of change patterns can begin.

The software intended to fill this gap will be referred to as the "**Process Model Perturbator**" (Perturbator for short). As input, it will take either one or two BPMN files and a set of sequence flow IDs or element IDs based on the type of operation. As an output, it will provide a perturbated version of the Process in a BPMN file, based on the input and the commands that the Perturbator is requested to perform via command line or via domain-specific language.

The thesis has been organized as follows: In Section 1, the problem domain is introduced. Section 2 deals with the summary of literature and software that has been reviewed for change patterns. In Section 3, goals are defined, and functional, non-functional requirements are extracted from the goals. Moreover, the motives behind the goals are also discussed in detail. Section 4 discusses the architecture of the tool created as a resultant of this thesis, requirements, limitations of inputs and outputs, functionalities provided, and the decisions that have been taken while choosing specific tools when developing the Perturbator. Section 5 explains the justification for achieving the defined goals and satisfaction of requirements, both functional and non-functional. Section 6 explains the limitations concerning the change patterns discussed in **[1]** and how those inadequacies can be overcome. Finally, Section 7 concludes the work that has been done and highlights the current inadequacies and opportunities for future direction this research may proceed.

2 Background and Related Literature

2.1 BPMN

As discussed in Section 1, BPMN is the widely-used standard for process modeling. It is quite a complex notation with more than 100 symbols **[4]**. As per the scope of the research, it is not feasible to discuss all symbols. However, for the sake of simplicity, the symbols frequently discussed in the thesis are as follows:

Start Event: Represents the beginning of the process instance.

Fig 2.1 Graphical notation of start event in BPMN Diagram

End Event: Represents the termination of the process instance.

Fig 2.2 Graphical notation of end event in BPMN Diagram

Task: Represents the activity being done. Usually, it also has a label that defines the action.

Fig 2.3 Graphical notation of task in BPMN Diagram

Sequence Flow: Gateways, events, and tasks are connected using sequence flow which defines the sequence in which execution is done.

Fig 2.4 Graphical notation of sequence flow in BPMN Diagram

Parallel Gateway: It is used to represent concurrent execution of multiple branches of a process that are not dependent upon each other.

Fig 2.5 Graphical notation of parallel gateway in BPMN Diagram

Exclusive Gateway: It is used to model the relation between alternate branches, whose execution depends upon the satisfaction/rejection of a condition.

Fig 2.6 Graphical notation of exclusive gateway in BPMN Diagram

As far as the implementation of Perturbator is concerned, support is not provided for pools, subprocesses, inclusive gateways, and other BPMN elements at the moment. Therefore, those elements have not been discussed in detail in this thesis. However, some of the change patterns (Section 2.2) discuss **subprocesses**, a compound activity composed of a collection of tasks and other elements connected by sequence flows.

The usage of BPMN Elemens is explained in the following diagram with the example of a process being implemented by a dairy product manufacturer. After completing the activities of acquiring raw material and producing different items in parallel, a decision to dispatch order is taken if a premium order has been placed; otherwise, the products are stored.

Translating the scenario to the BPMN notation introduced above, sequence flows *f1-f18* connect start and end events (represented by the circles with thin and thick border respectively), tasks (represented by rectangles with round borders), exclusive gateways (diamond with a cross), and parallel gateways (diamond with a plus sign). From *f2,* a parallel **split gateway** is introduced, which has a singular incoming sequence flow and multiple outgoing sequence flows. It ensures that the branches *f3-f4-f5, f11-f12-f13,* and *f14-f15-16* are executed in parallel. Next, the branches converge at a parallel **join gateway**, which has multiple incoming sequence flows, but singular outgoing flow, which is *f6* in our case. Similarly, after *f7,* an exclusive split gateway introduced might either execute the branch *f8-f9* or *f17-f18* based on a condition. Finally, the branches converge at an exclusive join gateway, where the branches merge, and a single outgoing flow is directed towards the task to notify delivery.

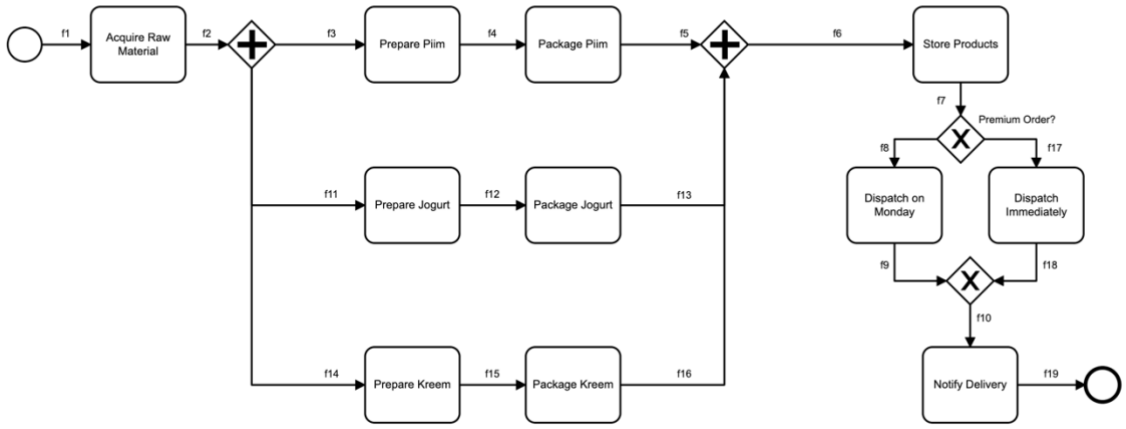

Fig. 2.7: Business Process of Dairy Product Manufacturer

2.2 Change Patterns

A set of fundamental change patterns allow making modifications in a business process. These change patterns are essential in providing a basic framework to enable flexibility in a process after it has been modified **[1]**. A set of change patterns with a much simpler explanation is also given in **[7].**

These defined patterns can be identified either as essential functions such as adding/removing a node or a sequence flow, but the disadvantage is that even for standard implementation of patterns, a combination of these steps of building blocks can get very complex even for a simple mutation.

The change patterns discussed in **[1]** discuss the changes needed to be done at an elemental level in the Process that uses the **process fragments**. Process fragments are a set of BPMN elements with only a single entry and exit point connected via sequence flows. However, the fragments themselves are meaningful and coherent to the extent that if a start event is added before the first element of the process fragment, and an end event is added after the last element of the fragment, it would constitute itself an independent process.

For example, to add a new node *N* at a sequence flow *E* of the business process and translate it in the form of a sequence of fundamental add/remove of node/sequence flow steps, the simplest form will look like something like this:

1. Generate new node *N*
2. Get starting node N_s of the sequence flow *E*
3. Get ending node N_e of the sequence flow *E*
4. Generate new sequence flow E_e starting from the new node *N* and pointing to node N_e
5. Update existing sequence flow *E* to start from node N_s and point towards node *N*

From the above steps, it is evident that even the basic steps can become complex to implement repetitively, even for fundamental steps in case of intricate adaptation patterns, such as adding a process fragment or removing a process fragment. Therefore, it is necessary to standardize complex patterns by combining these basic steps called **change primitives** by the paper being discussed. Following change primitives have been mentioned in **[1]**:

- **CP1:** Add Element
- **CP2:** Remove Element
- **CP3:** Add Sequence Flow
- **CP4:** Remove Sequence Flow

In the Perturbator, the implementation for **CP1** and **CP2** will be delivered as part of this thesis.

The standardized combinations of change primitives are known as the **adaptation patterns,** which will be strictly relevant to making structural changes in a business process. Following adaptation patterns have been discussed in **[1]:**

- **AP1: Insert Process Fragment**
  - **a) Serial Insertion:** Embed the Process Fragment within an existing sequence flow of the Process.
  - **b) Parallel Insertion:** Insert the Process Fragment in parallel with the existing Process Flow.
  - **c) Conditional Insertion:** Insert the Process Fragment to be executed as an alternate to the existing Process Flow.
- **AP2: Skip Process Fragment**:
  - Skip the execution of the Process Fragment.
- **AP3: Move Process Fragment:**
  - Move a process fragment from a current position (Section 5.3.1) to a target position (Section 5.3.1) in the same Process.
- **AP4: Replace Process Fragment:**
  - Replace a process fragment in a process with another process fragment

- **AP5: Swap Process Fragment:**
  - Exchange positions of two different process fragments in a process
- **AP6: Extract Process Fragment:**
  - Remove a process fragment from an existing process and replace it with a subprocess containing the process fragment.
- **AP7: Inline Process Fragment:**
  - Replace a subprocess with the process components inside the Subprocess.
- **AP8: Embed Process Fragment in Loop:**
  - Add a loop construct over a process fragment in a process.
- **AP9: Parallelize Process Fragment:**
  - Process fragments in a sequence are replaced by branches splitting and merging on parallel gateways
- **AP10: Embed Process Fragment in Conditional Branch:**
  - A process fragment is encapsulated with split and join exclusive gateways to be only executed if certain conditions are met.
- **AP11: Add Control Dependency:**
  - CP3, but ensures that the correctness properties (such as avoiding deadlocks and maintaining the correct order of dependencies) are preserved.
- **AP12: Remove Control Dependency:**
  - CP4, but ensures that the correctness properties (such as avoiding deadlocks and maintaining the correct order of dependencies) are preserved.
- **AP13: Update Condition:**
  - Update condition for an alternate flow for a branch generated due to splitting after an exclusive gateway
- **AP14: Copy Process Fragment:**
  - AP3, but the process fragment is not skipped from its original position.

The Perturbator will be providing the implementation for **AP1** and **AP2** only.

As discussed above, implementing these fundamental adaptation patterns will allow developing flexible PAIS. However, it will also solve many technical blockages for optimization problems and fill the gaps present in current implementations in various software as identified in the paper.

2.3 IBM WebSphere Business Modeler Extension

As discussed above, no tool can programmatically implement the change patterns. However, **[6]** discusses the implementation of an extension to IBM WebSphere Business Modeler. The IBM WebSphere Business Modeler is a GUI-based tool developed by IBM which allows the user to develop, simulate, and optimize business models.

The extension cannot implement the sophisticated version of change patterns in a programmatic way that has been discussed in **[1]**. It is only able to apply the changes manually via Graphical User Interface. Nevertheless, it can apply certain combinations of the change primitives termed **[6]** as **pattern compounds.** For example, the extension can apply the following pattern compounds:

- **Sequence:** Sequence can be considered equivalent to the serial insertion pattern (Section 2.2), where the process fragment is inserted in an existing flow.

- **Parallel Compound:** It is equivalent to the parallel insertion pattern (Section 2.2), where the fragment is inserted into the new parallel flow that starts from the parallel split gateway and merges back to the actual Process at the parallel join gateway.

- **Alternative Compound:** It is equivalent to the conditional insertion pattern (Section 2.2), where the fragment is inserted into the new alternate flow that starts from the exclusive split gateway and merges back to the Process at the exclusive join gateway.

- **Cyclic Compound:** This compound is used to implement a loop or a cycle around a specified process fragment. The split occurs from where the fragment is ending, and it merges back to the flow from where the process fragment begins.

These pattern compounds can be applied only in the following scenarios:

- **The pattern is applied only to a single sequence flow:** In this case, all four pattern compounds are applicable, and the soundness **[6]** of the Process is preserved.

- **The pattern applied to a pair of sequence flows:** In this case, only parallel and alternate compounds can be applied, and the soundness cannot be ensured in all combinations.

- **The pattern applied to a set of sequence flows:** As far as compounds are concerned, only parallel and alternate compounds are allowed. However, if the conditions [6] are met, the corresponding basic operations such as split, merge, synchronize are also applicable. Thus, for example, if all sequence flows are not connected, merging can occur.

The operations that are allowed are pretty restricted and fundamental. However, the features of validation, simulation, and restriction allow the user to create error-free process models and increase the tool's utilization.

The main drawback of the tool is that it does not provide support for automation as there is no programmatic interface or library available. Secondly, it has a limited set of patterns applicable. Therefore, the tool is helpful at a small scale where a single user is manually making changes and testing the correctness of the processes.

2.4 PIX

PIX (Process Improvement Explorer) **[8]** is an initiative started at the University of Tartu, which will build the foundations of a new generation of process improvement methods that are not directed or bound by the limits of the models manually designed by the analysis, but instead based on optimization heuristics operating over process execution data.

PIX will be dealing with the development of frameworks and algorithms to identify the possible improvements to business processes that can be obtained by modifying the control

flow of the Process, adjusting resource allocation to different steps of the Process, partial automation, or the changes in rules.

The PIX project will focus on the following improvement opportunities:

- **Task automation**: Discovering elements that can be partially or fully automated.
- **Control-flow optimization**: Optimizing the control flow of a process that can lead to lower cycle times, lower costs, or more minor SLA violations.
- **Decision optimization**: Optimizing the decision logic to ensure that the process can be completed by executing lesser tasks.
- **Resource optimization**: Fine-tuning the resource allocation logic in a process or fine-tuning the prioritization of tasks to reduce costs or decrease cycle times.
- **Risk-based optimization:** Refining the processes by adding/removing elements and updating the structure of the Process based on the risk value of removing a particular task or a process fragment.

As a starting point for this Master's thesis, the idea is to implement fundamental change patterns **[1],** discussed in detail in the previous sections.

As a part of the PIX initiative, the tool to be developed due to this Master's thesis is envisioned to become the fundamental ingredient for the research to be done in risk-based optimization. The Perturbator tool will provide the optimizer with an engine that generates the different variations of a process, which will be the input for the risk-based optimizer.

3 Requirements

3.1 Motivation

3.1.1 Lack of solutions for the programmatic implementation of Change Patterns

There is a lack of software that can incorporate change patterns in a process model, especially in the commercial domain **[1]**. For example, commercial software like Flower and Staffware might provide the features to add/remove nodes and sequence flows from a business process model, but it cannot offer advanced features of inserting/removing business process fragments and other change patterns. Moreover, there is no tool available that has defined a Domain Specific Language for these change patterns. Perturbator's idea is to fill these gaps of implementation present in current software solutions and be extensible enough to cover all possible change patterns in the future eventually.

3.1.2 Automating the perturbation of a modified business process

In order to keep the business process updated with changing business goals, modification is an essential step to manage business processes in an organization to make the processes financially profitable and viable in the long term. For this purpose, the example of a business process of a dairy manufacturer is taken into consideration (Fig. 2.7).

To understand the evolution of a business process to meet the needs, let us assume that there is a scenario that demand for Kreem has decreased during summers compared to the need in winters. Therefore, as the business goals have changed, the effects will also be evident in the business process, as the production of Kreem will be halted for some time unless the current stock has been depleted.

The Perturbator can provide us with a solution like the one given in Fig 3.1 can be more optimized as it will conditionally skip the process fragment that comprises the steps of preparation and packaging of Kreem. To achieve the scenario mentioned above, we can generate the condition for the exclusive split gateway at *f14* to execute the alternate flow f_c consistently.

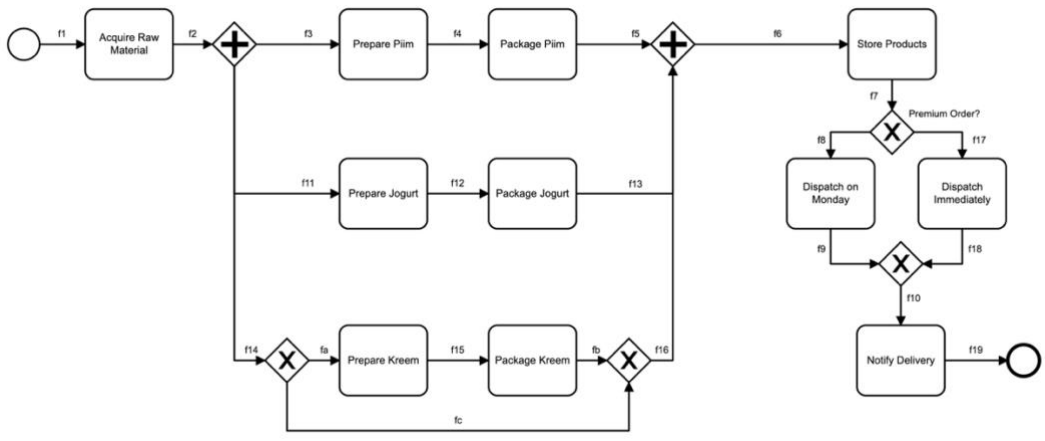

Fig. 3.1: Example of an optimization done in a business process for new business needs

In order to tackle this purpose, the Perturbator is designed to be a solution that can generate an alternate version of a business process in an automated manner to eliminate the manual work for achieving the final product.

3.2 Goals

Following are the high-level goals that are intended to be achieved while researching the thesis.

- Development of change patterns in Perturbator
- Defining a Domain Specific Language for Perturbator change patterns
- Packaging of the Perturbator for reuse and distribution
- Testing of the Perturbator to ensure correctness and performance

The goals can be better explained in much more detail as Functional and Non-Functional Requirements as follows:

3.2.1 Functional Requirements

From a perspective of functionality that the tool should provide, it should implement multiple changes to a single process. Change can either add or remove an element or process fragment from the Process provided to the Perturbator as an input. The elements that might be inserted or removed can be either tasks or gateways. The process fragment to be inserted can be provided in a separate BPMN file along with the primary input BPMN file.

Specifications of changes to be made (e.g., IDs of the elements to be skipped, or the flows where the fragment has to be inserted or removed) can be explicitly mentioned as arguments either in the Command Line Interface or the Domain Specific Language developed especially for the Perturbator.

Following is a formal specification of the functional requirements the Perturbator is expected to satisfy:

No.	**Requirement**
FR1	As a user, I shall be able to add a task to a process at a specified sequence flow using the command line.
FR2	As a user, I shall be able to skip a task from a process using the command line.
FR3	As a user, I shall be able to add a gateway to a process at a specified sequence flow using the command line.
FR4	As a user, I shall be able to skip a gateway from a process using the command line.
FR5	As a user, given two different processes in different BPMN files, I shall be able to serial insert a process fragment in another process at a specified flow.
FR6	As a user, given two different processes in different BPMN files, I shall be able to parallel insert a process fragment in another process at a specified flow.
FR7	As a user, given two different processes in different BPMN files, I shall be able to conditionally insert a process fragment in another process at a specified flow.
FR8	As a user, I shall be able to skip a process fragment in a process.

FR9	As a user, I shall be able to implement requirements FR1-FR8 using CLI commands.
FR10	As a user, I shall be able to implement a combination of requirements FR1-FR8 by explaining the steps in a domain-specific language (DSL)

Table. 3.1: Functional Requirements

3.2.2 Non-Functional Requirements

The Non-Functional Requirement specification tries to cater to the basic needs of the end-users that will be using this tool to perturb their processes. It also attempts to address the common issues that the developers usually face.

From the end-user perspective, a desire has been expressed to provide interfaces to generic patterns and the fundamental building blocks, so the end-user can implement even those patterns themselves that are currently not implemented. Moreover, to promote usage, multiple interfaces are provided according to user preference.

The Perturbator is intended to be written so that the developers working to extend the Perturbator itself, or the engineers building their systems based on the implementation of the Perturbator, find it extremely easy to use and easy to extend.

Following is a formal specification of the non-functional requirements:

No.	**Requirement**	**Description**
NFR1	Extensibility	Code should be written so that the implementation can be extended in the future for remaining change patterns (Section 2.2).
NFR2	Efficiency	The solution should be efficient and must be able to deal with multiple perturbations at one time.
NFR3	Generality	The features implemented should apply combinations of fundamental perturbations to support features that are not built in the tool yet.
NFR4	Reusability	The packaging should be done to be used either as a standalone tool or within a web application.
NFR5	Usability	The solution should be well-documented and easy to use via CLI and DSL.

Table. 3.2: Non-Functional Requirements

4 Design

The source code being discussed during this section is available on the following URL: https://github.com/Zohaib94/perturbator

The research has been entirely focused on the development of new software. Therefore, this section is divided into the following sections:

- Architectural overview
- Limitations and requirements
- Implementation logic behind the features implemented in the Perturbator
- Usage of features, tools used, and development decisions taken for the Perturbator

4.1 Architectural Overview

As far as the architecture is concerned, the flow of execution and code organization is discussed since the software does not include any databases or server deployments, which is left on the users and developers who will be using the package, allowing to mitigate the cost and risk of maintenance and scaling.

Figure 4.1 explains the flow of executing when the user is using CLI to implement singular operations. When a command is run from the terminal/command prompt, which is acting at the CLI, initially, the request goes on to the commands module, which parses the command and extracts the attributes required. Once the parsing of commands and validation of attributes is done, the execution is then forwarded to the perturbator core in the support modules, where the actual implementation resides. Eventually, the user receives the file with the output, according to the output path mentioned in the initial command.

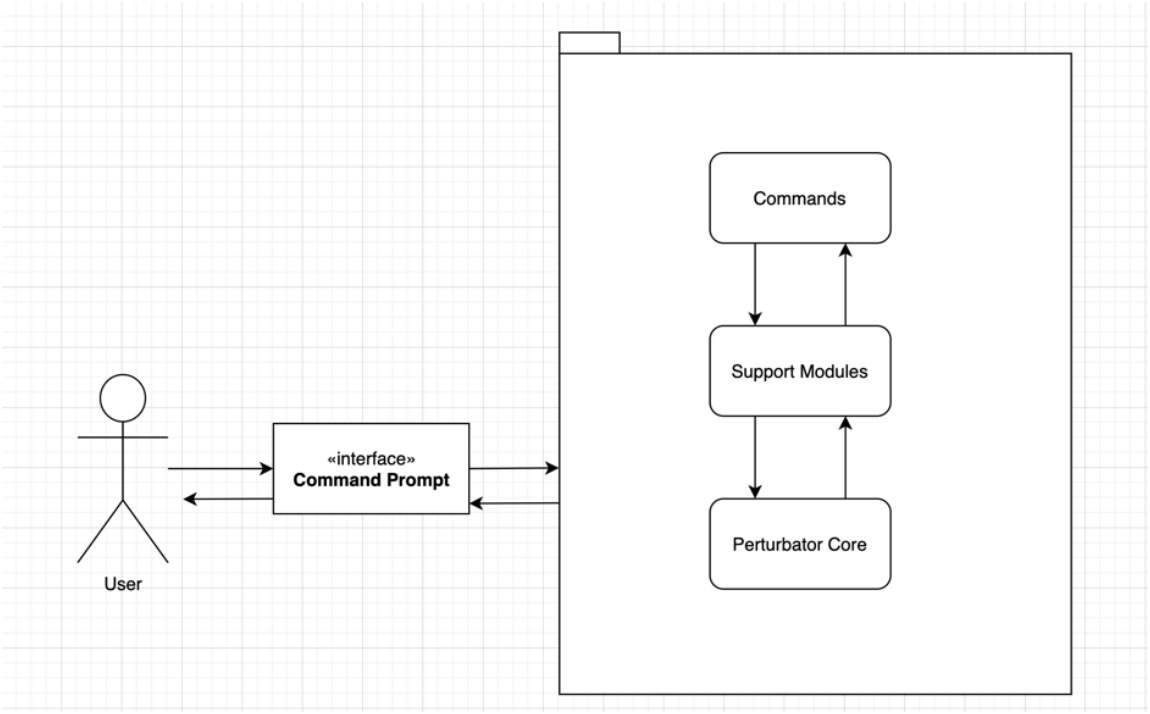

Figure. 4.1: Execution flow of CLI Commands without DSL

Figure 4.2 explains the execution in the case when commands are run in batches using the defined DSL. In the command, the text file with the commands is also passed. After the command has been parsed and validated, the text file is then passed on to the language parser. According to the defined language models, the parser detects the commands and then executes the corresponding method from the perturbator core in the support modules.

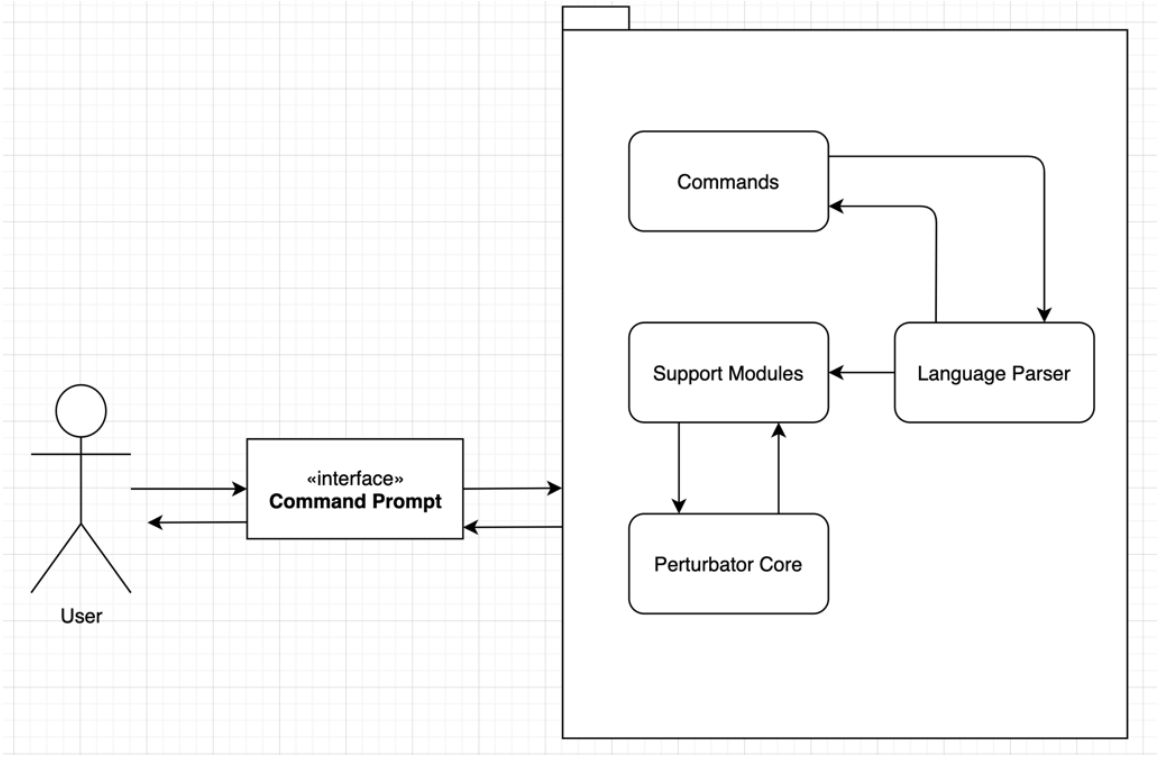

Figure. 4.2: Execution flow of CLI Commands with DSL

As displayed in Figure 4.1 and Figure 4.2, the interface directly communicates with the package only. The reason behind is this is to avoid coupling the interface with the underlying modules. Initially, the command passed as input via interface will communicate with the commands module, but the output does not need to be permanently returned through a specific module inside the package because an error/crash can occur in any module. Error handling would become very difficult to maintain in case if more modules are added to the package. The current architecture allows the flexibility to add new modules and new commands and integrate more functionality without writing spaghetti code containing an implementation that is not relevant to the module.

Following is the code organization of the tool as follows:

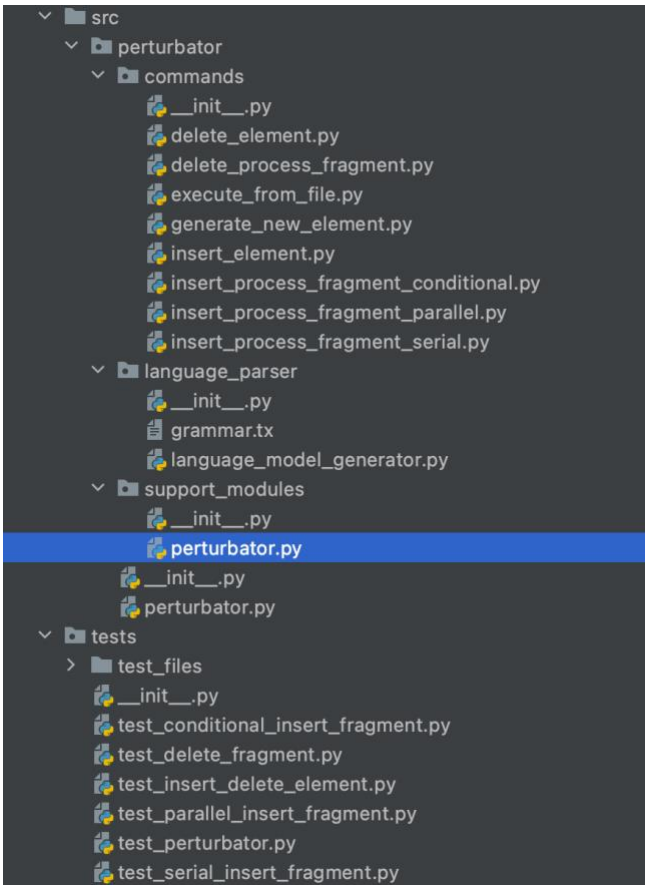

Figure. 4.3: Code Structure

The source code has been divided into three modules:

1. **Commands:**
   All the command wrappers are mentioned in these files, and all features regarding the command-line interface are provided in these files for the methods given in the perturbator support module.

2. **Language Parser:**
   The language parser contains all the logic regarding defining the domain-specific language and reading input files for batch commands and parsing the file according to the language model being generated according to the language definition given in **grammar.tx.**

3. **Support Modules:**
   The support module contains all the core implementation regarding parsing the BPMN files and modifying the existing BPMN structures, such as adding/removing flows and implementing change patterns based on the core functionalities known as the change primitives.

Apart from that, the unit and integration tests are provided in the tests folder.

4.2 Input/Output Formats and System Requirements

As far as the requirements are concerned, the Python version that needs to be used is below 3.9 because the native library **XMLElementTree** has a method known as *getChildren*, used to iterate over the children of the process tree has been deprecated in Python version greater than 3.9.

Currently, the portion of the BPMN diagram that deals with the implementation of a graphical view over the BPMN diagram is not under concern. Therefore, the input BPMN files should contain only the process elements.

Following is an example of the BPMN file that has to be passed to the Perturbator.

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn:definitions xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
xmlns:di="http://www.omg.org/spec/DD/20100524/DI" id="Definitions_05j5hnm"
targetNamespace="http://bpmn.io/schema/bpmn" exporter="Camunda Modeler"
exporterVersion="4.5.0">
  <bpmn:process id="Process_1elnczg" isExecutable="true">
    <bpmn:startEvent id="StartEvent_1">
      <bpmn:outgoing>flow11</bpmn:outgoing>
    </bpmn:startEvent>
    <bpmn:task id="task11">
      <bpmn:incoming>flow11</bpmn:incoming>
      <bpmn:outgoing>flow12</bpmn:outgoing>
    </bpmn:task>
    <bpmn:task id="task12">
      <bpmn:incoming>flow12</bpmn:incoming>
      <bpmn:outgoing>flow13</bpmn:outgoing>
    </bpmn:task>
    <bpmn:endEvent id="Event_0gckvrh">
      <bpmn:incoming>flow13</bpmn:incoming>
    </bpmn:endEvent>
    <bpmn:sequenceFlow id="flow11" sourceRef="StartEvent_1" targetRef="task11" />
    <bpmn:sequenceFlow id="flow12" sourceRef="task11" targetRef="task12" />
    <bpmn:sequenceFlow id="flow13" sourceRef="task12" targetRef="Event_0gckvrh" />
  </bpmn:process>
</bpmn:definitions>
```

As far as the other inputs are concerned, they are primarily arguments of the commands, so no particular input format is required. However, the arguments needed for the file paths should be absolute and not relevant paths.

A single BPMN file should only consist of a single Business Process Model. That is why two separate arguments for two different files are required in case of inserting process fragments.

The current implementation expects that the insertion and skipping do not include boundary events as their support is yet to be reviewed. However, there is no issue in the working of the Perturbator if there are multiple incoming and outgoing flows to an element, as the user has to provide the sequence flow IDs themselves in case of branched or serial insertion.

As far as insertion of elements is concerned, even though the implementation is generic to insert any kind of BPMN element, but at the moment, there is an internal validation to allow

the user to insert only tasks and exclusive gateways. This issue will be fixed to enable all possible elements soon.

Even though an internal implementation is available to add new flows from one element to another is written, the interface via CLI or DSL to that functionality is not provided. Therefore, currently, it is not possible to add alternate flows to existing gateways in the Process.

The output will also be in the form of a BPMN file, which will have a format same as the above sample BPMN file. However, it will not have any details regarding the graphical view of the BPMN diagram.

4.3 Implementation Details

The high-level logic and the details of implementation for the following features are discussed in this section:

- **CP1:** Insert Element
- **CP2:** Skip Element
- **AP 1a**: Insert Process Fragment (Serial Insertion)
- **AP 1b**: Insert Process Fragment (Parallel Insertion)
- **AP 1c**: Insert Process Fragment (Conditional Insertion)
- **AP2**: Skip Process Fragment

4.3.1 Insert Element

Following are the high-level logical steps (example in Fig 4.4) for implementing the insertion of an element *E* at a specific flow. Assuming it to be *f* in our case.

1. Generate a new element *E.* (Fig 4.4(b))
2. Store the source node of flow *f* as N_s
3. Store the target node of flow *f* as N_t
4. Generate a new sequence flow f_n that starts from *E* and ends at N_t (Fig 4.4(c))
5. Update the target of sequence flow *f* to the new element *E* (Fig 4.4(d))

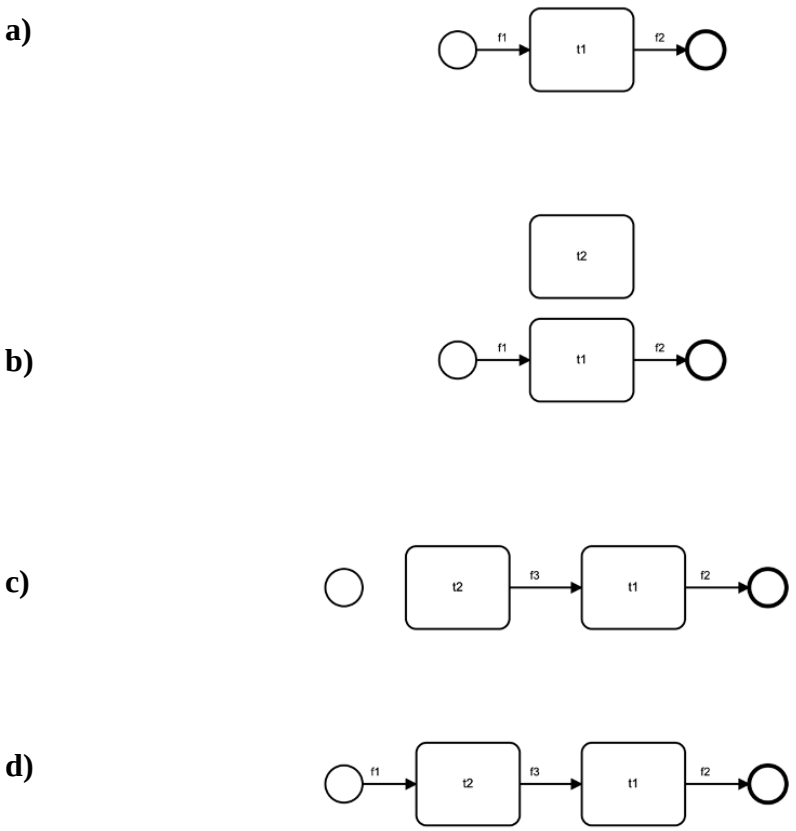

Figure 4.4: Steps for insertion of an element

Referring to Fig. 2.7, to update the Process to deliver condensed Piim, a new element of type task is inserted in the flow directed towards Processing Piim and Packaging Piim. Assuming that the insertion's flow has a unique ID *f4*, then the insertion will modify the business process so that a new task is added at *f4*. In order to connect the task **Condense Piim** to **Package Piim,** another new flow f_a is created. Furthermore, the existing *f4* has an updated target for the newly added task. A pictorial representation of the result obtained after the changes is given in Fig. 4.5.

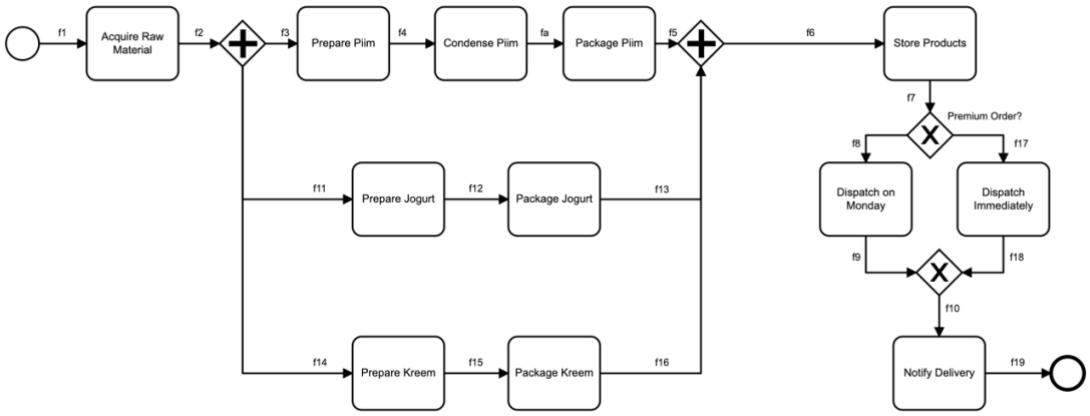

Figure. 4.5: Dairy Production Process (Figure 1.1) after a task is inserted to condense Piim

4.3.2 Skip Element

In order to implement this feature, split and join gateways are utilized. Split gateways are exclusive gateways that decide whether to continue with the execution or whether the alternate flow can skip the element. From the split gateway, the process flow diverges **[4].** Both the alternate flow and the actual flow outgoing from element E are then connected to the join gateway, from which process execution is resumed as before, and here the flow of Process converges **[4].**

The feature to remove a task entirely from a process is also implemented, but the functionality of that change primitive is limited to tasks with one incoming and one outgoing flow. It is not possible at the moment to implement the change primitive to entirely remove gateways from the Process because gateways can have multiple incoming or outgoing sequence flows. Complete removal of an element, in that case, will result in a non-deterministic process graph. Removal of an element connected to multiple sequence flows of the Process will cause an *N*x*M* combination of possible flows, where *N* is the number of incoming flows and *M* is the number of outgoing flows. Due to this reason, the Process itself loses context knowledge and correctness.

Since the use of the change primitive to remove a single task is limited in nature, a decision has been taken to only provide the interface for the primitive to skip an element, which is generic and can be used for any kind of element.

Following are the high-level logical steps (example in Fig 4.6) to implement skipping an element *E*.

1. Generate a new split gateway G_s
2. Generate a new join gateway G_j
3. Store all incoming sequence flows to element *E* in list F_i
4. Store all outgoing sequence flows from element *E* in list F_o
5. Add a new sequence flow F_s, from G_s to *E* (Fig 4.6(b))
6. Add a new sequence flow F_j, from *E* to G_s (Fig 4.6(c))
7. Add a new alternate flow F_a, from G_s to G_j (Fig 4.6(d))
8. For each flow f_i in list F_i, update the target of flow f_i to G_s
9. For each flow f_o in list F_o, update source of flow f_o to G_j

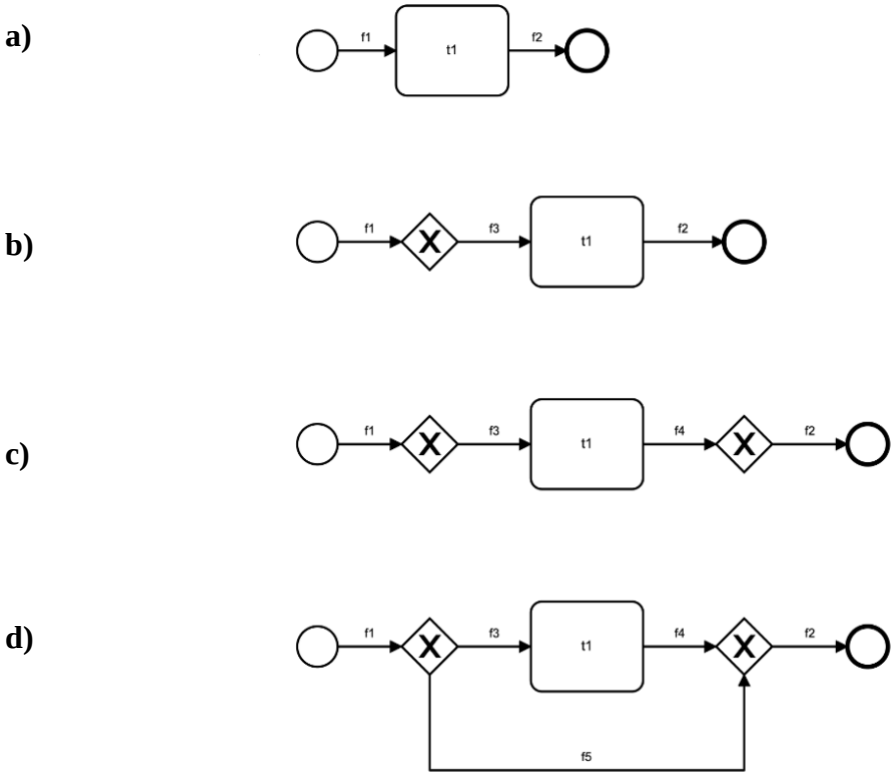

Figure. 4.6: Steps for skipping an element

Considering the running example of the dairy process given in Fig. 2.7, assuming that the Process instance has completed the operation of preparing Jogurt and there are no more raw materials available to prepare more Jogurt, then the Process can just skip the task of "Prepare Jogurt." After skipping this task, the Process will look as follows:

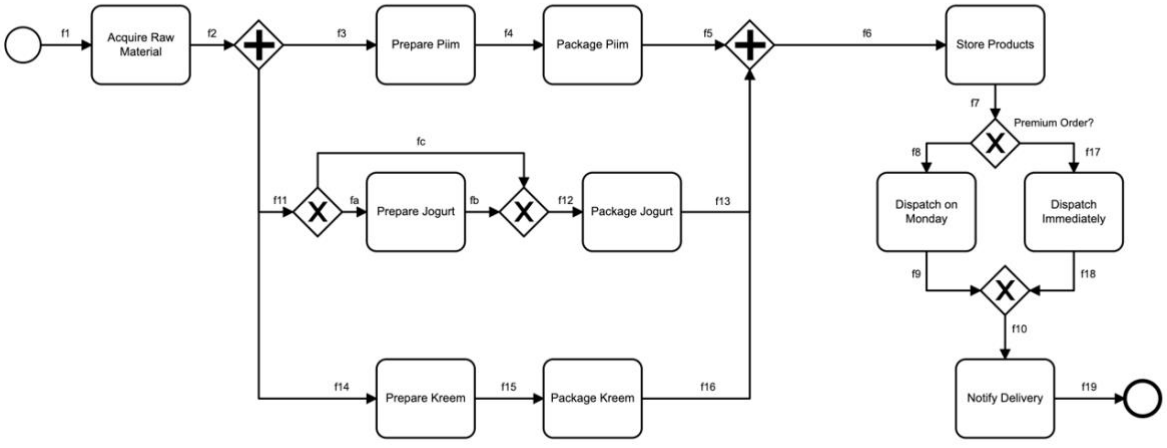

Figure. 4.7: Dairy Production Process (Figure 2.7) after the task to prepare Jogurt is skipped

4.3.3 Adaptation Pattern 1a: Insert Process Fragment (Serial):

Now that the required change primitives are available, the development of Adaptation Patterns can be carried out.

The process fragments are already discussed in Section 2.2. However, from an implementational point of view, it was decided to represent a process fragment as a separate BPMN process, with both start and end events. The implementation will remove these events automatically so that the user does not have to make manual changes in BPMN files.

Following are the high-level logical steps (example in Fig. 4.8) for the implementation of serial insertion of a process fragment *PF* in a process *P* at a specific flow *f*.

1. Copy process fragment *PF* from the source file to the target file
2. Save target element *E1* of outgoing sequence flow of start event in *PF*
3. Save source element E_n of incoming sequence flow of end event in *PF*
4. Remove start event from *PF* (Fig 4.8(b))
5. Remove end event from *PF* (Fig 4.8(b))
6. Remove outgoing sequence flow of start event in *PF* (Fig 4.8(b))
7. Remove incoming sequence flow of end event in *PF* (Fig 4.8(b))
8. Save source element E_s of flow *f* in *P*
9. Save target element E_t of flow *f* in *P*
10. Remove flow *f* from *P* (Fig 4.8(c))
11. Generate new sequence flow f_a starting from element E_s of *P* to target element E_1 of *PF* (Fig 4.8(d))
12. Generate new sequence flow f_b starting from source element E_n of *PF* to target element E_t of flow *f* in *P* (Fig 4.8(e))

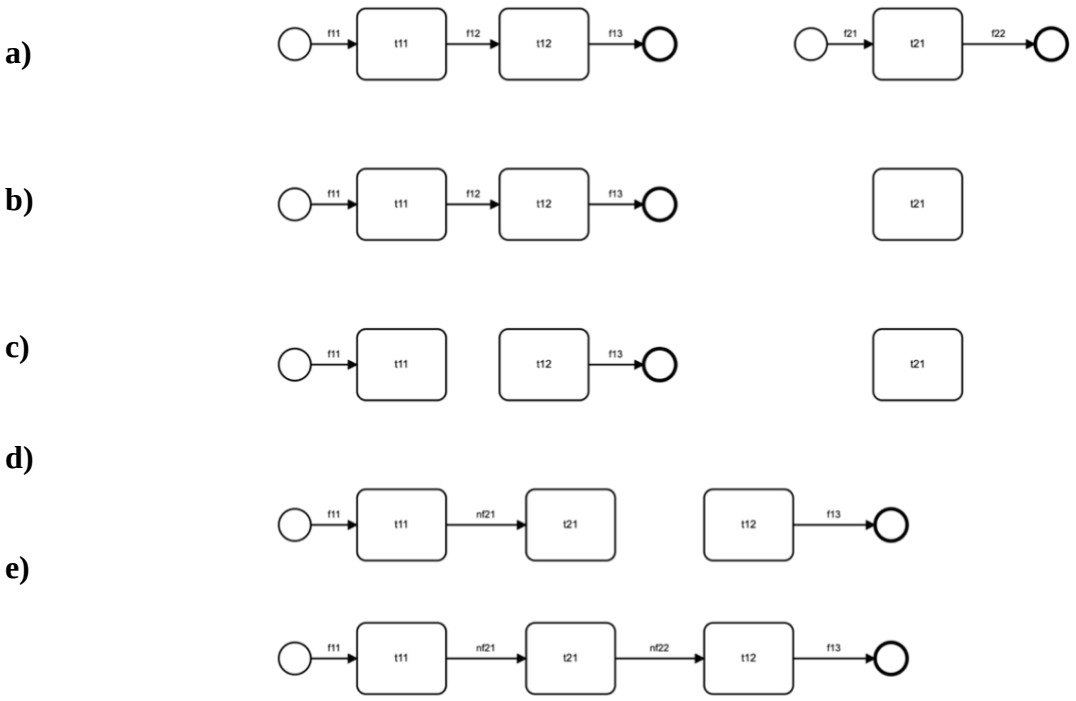

Figure. 4.8: Steps for serial insertion of a process fragment

Considering the running example of the dairy process given in Fig. 2.7, assuming another process for separation and dispatching of Raw Materials as follows:

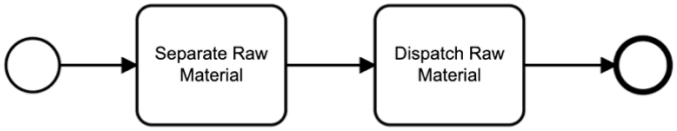

Figure. 4.9: Process Fragment for separation and dispatching raw materials for production

According to the business requirements, to merge this Process in the primary Process in Fig. 2.7, the operations can be performed after raw materials have been acquired. Supposing that, serial insertion occurs at flow *f2,* which is directed from the task "Acquire Raw Materials" and directed towards the parallel split gateway, from where the preparation of products begins. Thus, the updated Process will look as follows:

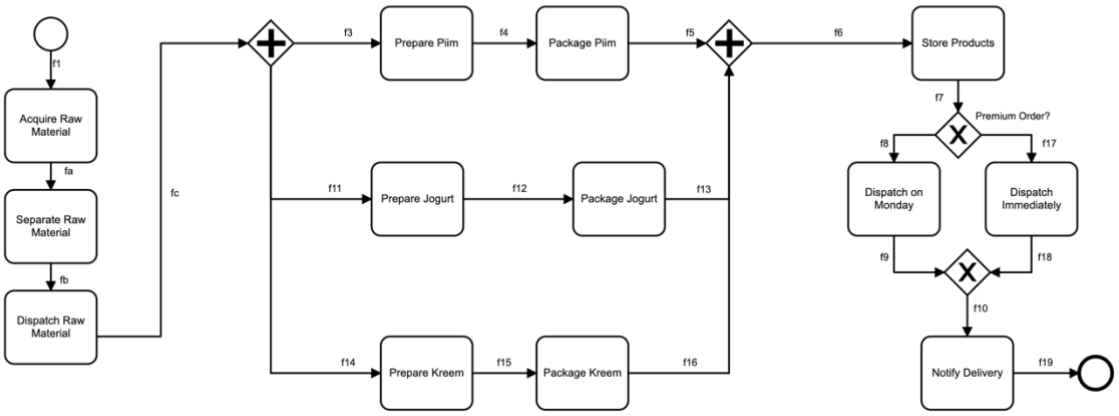

Figure. 4.10: Dairy Process (Figure 2.7) after serial insertion of process fragment (Figure 4.9) after acquiring raw material

4.3.4 Adaptation Pattern 1b: Insert Process Fragment (Parallel):

Basically, from a conceptual perspective, the insertion of a process fragment in a branch (parallel/conditional) is a combination of skip and serial insert process fragment operations. Furthermore, it uses the implementation of insert serial process fragment as discussed in detail before.

Following are the high-level implementation steps (example in Fig. 4.11) for parallel insertion of a process fragment when the starting flow f_s and ending flow f_e of the branch in process P is provided:

1. Insert element (Section 4.3.1) parallel gateway P_s in process P at flow f_s (Fig 4.11(b))
2. Insert element (Section 4.3.1) parallel gateway P_e in process P at flow f_e (Fig 4.11(c))
3. Create alternate sequence flow f_a in P, starting from P_s and ending at P_e (Fig 4.11(d))
4. Insert process fragment serially (Section 4.3.3) in the alternate flow f_a in process P (Fig 4.11(e))

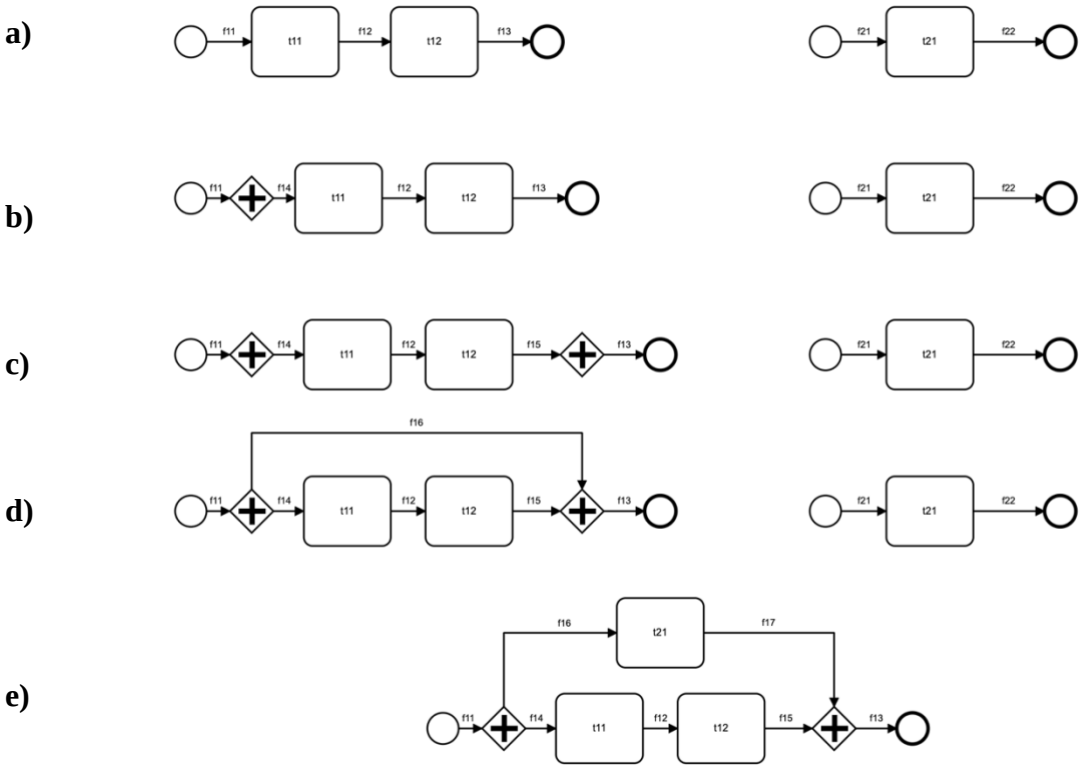

Figure. 4.11: Steps for parallel insertion of a process fragment

Considering our running example of Dairy Process (Fig. 2.7) and process fragment to arrange raw materials (Fig. 4.9), there is a need to acquire, separate, and arrange raw materials in parallel to optimize our Process. To satisfy this requirement, insert the process fragment in a parallel way on the incoming and outgoing flow from the task "Acquire Raw Materials." that have IDs *f1* and *f2*, respectively. Consequently, the Process will look as follows:

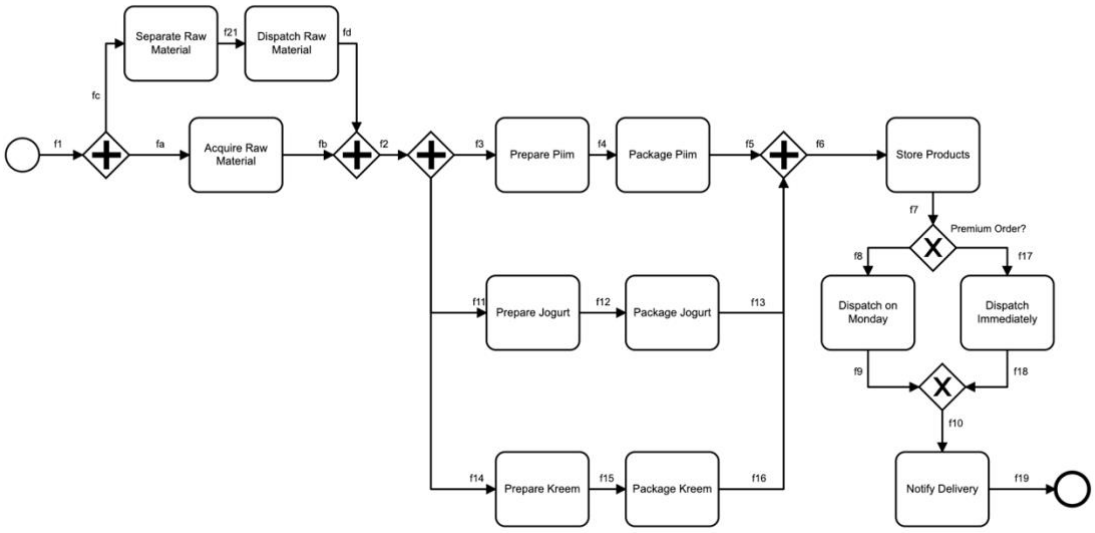

Figure. 4.12: Dairy Process (Figure 2.7) after parallel insertion of process fragment (Figure 4.9) for raw materials

4.3.5 Adaptation Pattern 1c: Insert Process Fragment (Conditional):

From a conceptual perspective, this pattern is identical to pattern 1b. The only difference is the gateways being used. In this pattern, exclusive gateways decide which path to continue based on conditions or attributes.

Following are the high-level implementation steps (example in Fig. 4.13) for conditional insertion of a process fragment when the starting flow f_s and ending flow f_e of the branch in process P is provided:

1. Insert element (Section 4.3.1) exclusive gateway C_s in process P at flow f_s (Fig 4.13(b))
2. Insert element (Section 4.3.1) exclusive gateway C_e in process P at flow f_e (Fig 4.13(c))
3. Create alternate sequence flow f_a in P, starting from C_s and ending at C_e (Fig 4.13(d))
4. Insert process fragment serially (Section 4.3.3) in the alternate flow f_a in process P (Fig 4.13(e))

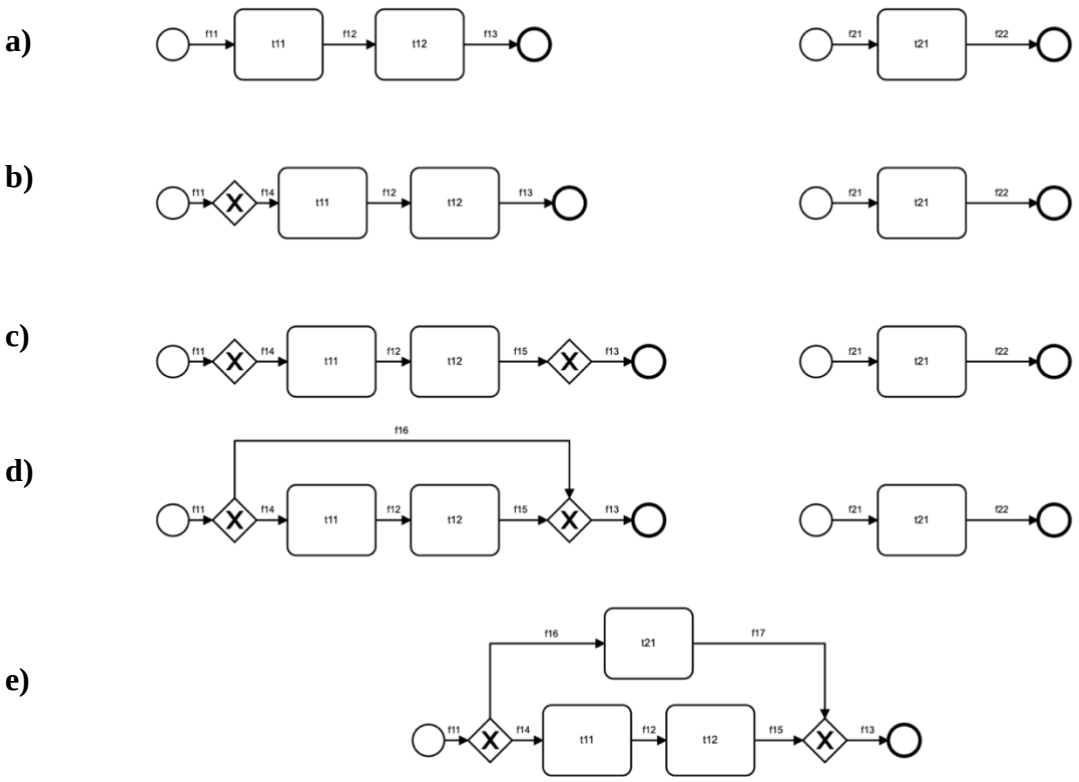

Figure. 4.13: Steps for conditional insertion of a process fragment

Considering our running example of the Dairy Process (Fig. 2.7), assuming another process fragment for preparing Cheese as follows.

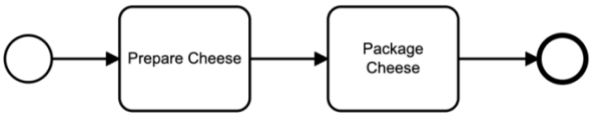

Figure. 4.14: Process Fragment for Preparing Cheese

Supposing that as per business requirements, to ensure that if the demand for Kreem is decreasing and Cheese is increasing, operations for Kreem are halted and resumed for production of Cheese and vice versa. For this purpose, conditional insertion of the process fragment (Fig 4.14) occurs in between the flows from where the task "Prepare Kreem" is starting and the task "Package Kreem" is concluding. The IDs for the flows where the insertion is to be done are *f14* and *f16*, respectively. Thus, the updated Dairy Process (Fig. 2.7) will look as follows:

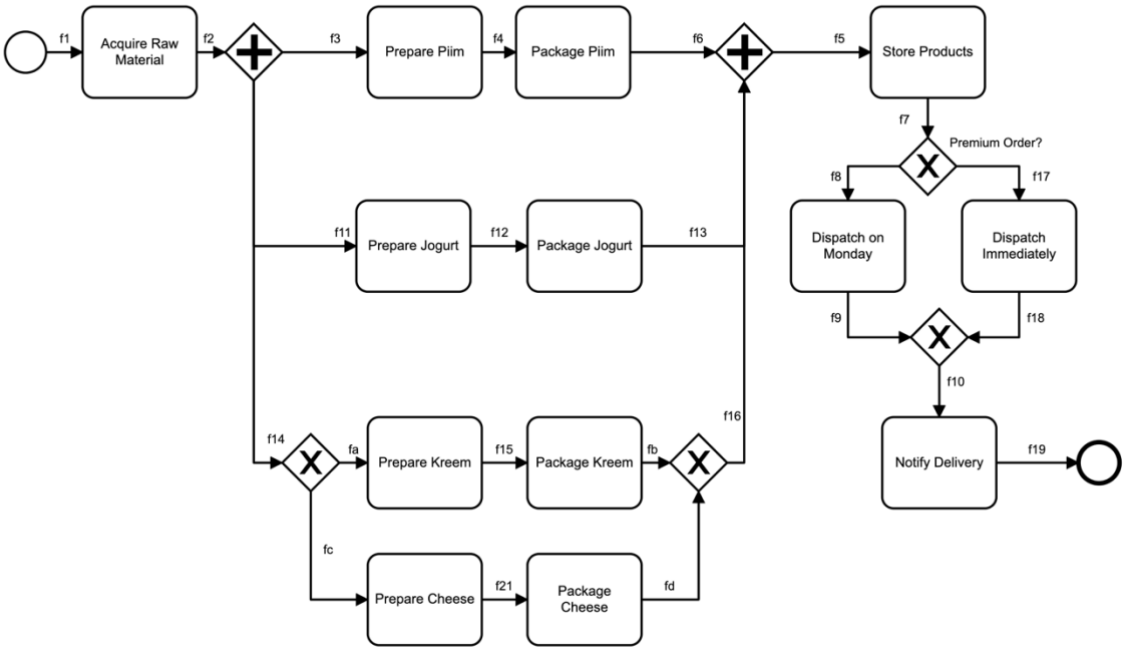

Figure. 4.15: Dairy Process (Figure 2.7) after conditional insertion of process fragment (Figure 4.14) for conditional cheese preparation

4.3.6 Adaptation Pattern 2: Skip Process Fragment:

As discussed in Section 4.3.2, it is not feasible to remove a fragment or an element from the Process altogether. Therefore, from a conceptual perspective, implementing this adaptation pattern becomes a subset of the implementation of Adaptation pattern 1c: Insert Process Fragment (conditional). The only difference is that another fragment in the alternate flow is not to be inserted serially.

Ensuring that the exclusive gateway always moves to the alternate path where no task is being executed, the execution of the fragment is essentially disabled, which is as good as deleting it entirely from the structure of the BPMN diagram.

Following are the high-level implementation steps (example in Fig. 4.16) for skipping a process fragment when the starting flow f_s and ending flow f_e of the branch in process P is provided:

1. Insert element (Section 4.3.1) exclusive gateway C_s in process P at flow f_s (Fig 4.16(b))
2. Insert element (Section 4.3.1) exclusive gateway C_e in process P at flow f_e (Fig 4.16(c))
3. Create alternate sequence flow f_a in P, starting from C_s and ending at C_e (Fig 4.16(d))

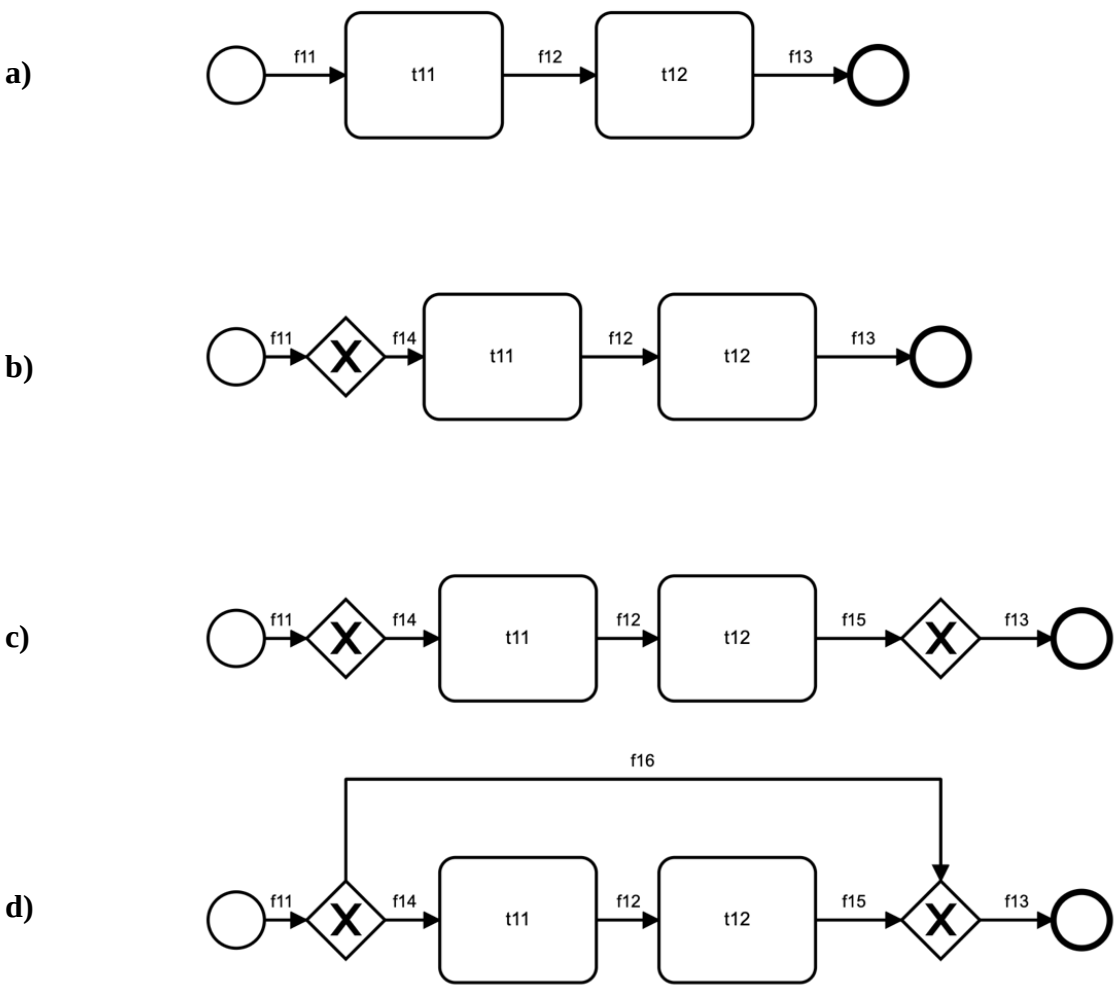

Figure. 4.16: Steps for skipping a process fragment

Considering our running example of the dairy process (Fig 2.7), the example of the skipping of a process fragment has already been discussed as a use case in Fig. 3.1.

4.4 Development Tools and Usage of Features

4.4.1 Tools used for Perturbator Module

The programming language that has been used for the development of the Perturbator is **Python**. The main benefits obtained by choosing to develop in Python are as follows:

- Native File Handling for BPMN files using **os** and **xml.etree.elementTree.**
- Native XML and BPMN parsing using **xml.etree.elementTree.**
- A mature IDE like **PyCharm** is available for debugging purposes.
- A collection of numerous packages for small purposes, such as generation of dynamic IDs using **shortuuid**. The third-party packages saved a lot of time and effort in implementing complex features such as developing a CLI and a DSL.
- Ease of packaging the final product at **PyPI.**
- Ease of testing using **PyTest.**

4.4.2 Tools and Definitions for Domain Specific Language Module

TextX package provided in Python was used to develop domain-specific language that allows providing input to the Perturbator in a text format within a text file, so several change patterns and operations can be implemented in batches without using the command line.

Instead of using an if-else approach, which is not extensible for such operations, a Domain Specific Language is introduced that can be easily extended in the future for the implementation of more change patterns. **TextX** provides the functionality to define a language in a "**.tx**" file and generate a language model itself. Based on the language model generated, both the reader and the parser can easily understand and verify which command has been requested or called from the input. Consequently, there is no need to write a specialized compiler to interpret and correlate the natural language commands to the Perturbator's functionality. Definition of the DSL is provided in Appendix III.

Following are the definitions that have been used for the commands:

Insert Element:

```
insert <elementName> to process in file <sourceFile> at flow <flowId> and output in
<outputPath>
```

Where,
<*elementName*>*:* name of the element to be added
<*sourceFile*>: an absolute path of the BPMN file to which element has to be added
<*flowId*>: ID of the sequence flow where the element has to be added
<*outputPath*>: an absolute path of the BPMN file where the output has to be written

Skip Element:

```
delete element with id <elementId> from process in file <sourceFile> and output in
<outputPath>
```

Where,

<*elementID*>*:* ID of the element to be removed
<*sourceFile*>: an absolute path of the BPMN file from which element has to be skipped
<*outputPath*>: an absolute path of the BPMN file where the output has to be written

Insert Process Fragment (Serial):

```
serial insert process from <sourceFile> to process in file <targetFile> at flow <flowId> and
output in <outputPath>
```

Where,
<*sourceFile*>: an absolute path of the BPMN file which is to be inserted
<*targetFile*>: an absolute path of the BPMN file to which insertion is to be done
<*flowId*>: ID of the sequence flow where process fragment is to be inserted
<*outputPath*>: an absolute path of the BPMN file where the output has to be written

Insert Process Fragment (Conditional):

```
conditionally insert process from <sourceFile> to process in file <targetFile> between
<initialFlowId> and <finalFlowId> and output in <outputPath>
```

Where,
<*sourceFile*>: an absolute path of the BPMN file which is to be inserted
<*targetFile*>: an absolute path of the BPMN file to which insertion is to be done
<*initialFlowId*>: ID of the sequence flow where the split gateway is to be added
<*finalFlowId*>: ID of the sequence flow where join gateway is to be added
<*outputPath*>: an absolute path of the BPMN file where the output has to be written

Insert Process Fragment (Parallel):

```
parallel insert process from <sourceFile> to process in file <targetFile> between
<initialFlowId> and <finalFlowId> and output in <outputPath>
```

Where,
<*sourceFile*>: an absolute path of the BPMN file which is to be inserted
<*targetFile*>: an absolute path of the BPMN file to which insertion is to be done
<*initialFlowId*>: ID of the sequence flow where the split gateway is to be added
<*finalFlowId*>: ID of the sequence flow where join gateway is to be added
<*outputPath*>: an absolute path of the BPMN file where the output has to be written

Skip Process Fragment:

```
delete process fragment from process in file <sourceFile> between <initialFlowId> and
<finalFlowId> and output in <outputPath>
```

Where,
<*sourceFile*>: an absolute path of the BPMN file which is to be inserted
<*initialFlowId*>: ID of the sequence flow where the split gateway is to be added
<*finalFlowId*>: ID of the sequence flow where join gateway is to be added

<*outputPath*>: an absolute path of the BPMN file where the output has to be written

4.4.3 Tools and Definitions for Command Line Interface Module

For the development of the command-line interface, Python has a highly sophisticated package available known as **Click.** It allows writing production-ready command-line tools by defining decorators above the methods to dispatch as a command-line functionality. Moreover, it also provides built-in file path handling across Operating Systems. Thirdly, the UNIX command-line conventions are implemented by default in the package, so the techniques of generating and executing the command are delegated to the package. Finally, the most crucial benefit is that it automatically generates help commands as well, so one can read about the command's documentation by just adding **–help** flag.

Usage of click package allowed to avoid writing a separate parser to read and parse commands from the command line, and it saved much time while allowing to implement a more extensive set of functionalities.

The list of the CLI commands available, and the detail of the arguments is given in detail in Appendix II.

4.4.4 Tools for packaging and release

In order to release the tool for mass usage, it was decided that taking the conventional route of deploying it on a server as an API service would be over-engineering for such implementation because, at the moment, the tool does not require any database resources or significantly enormous memory resources in order to complete executions. Secondly, executing batch commands via Domain-Specific Language via API would take much time, and the cost of scalability would increase a lot as the solution would have to be deployed on multiple containers so that other clients do not have to face downtime or wait for the resources to be released by another request-response cycle.

To avoid these issues mentioned above, launching this software as a python package seemed to be a more viable solution. For this purpose, another sophisticated python package, **Poetry** is used to build and release python packages. Furthermore, it also provides the support to test the package locally in a virtual environment and provide testing functionalities.

Another main issue in packages is maintaining the versions of Python, core dependencies, and development dependencies easily configurable using **Poetry.** The package provides an easy-to-use and easy-to-maintain abstraction over **setuptools** available by default in Python, challenging to configure and maintain for this purpose.

The Perturbator Python package is available at https://pypi.org/project/perturbator

4.5 CLI and DSL Examples

Considering the following BPMN for the example of Dairy Process (Fig. 2.7) is stored in a file path ***~/Desktop/input1.bpmn***,

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bpmn:definitions xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
xmlns:di="http://www.omg.org/spec/DD/20100524/DI" id="Definitions_0mome6e"
targetNamespace="http://bpmn.io/schema/bpmn" exporter="Camunda Modeler"
exporterVersion="4.5.0">
  <bpmn:process id="Process_1h951yo" isExecutable="false">
    <bpmn:startEvent id="StartEvent_0w7daze">
      <bpmn:outgoing>f1</bpmn:outgoing>
    </bpmn:startEvent>
    <bpmn:task id="a1" name="Acquire Raw Material">
      <bpmn:incoming>f1</bpmn:incoming>
      <bpmn:outgoing>f2</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f1" sourceRef="StartEvent_0w7daze" targetRef="a1" />
    <bpmn:sequenceFlow id="f2" sourceRef="a1" targetRef="g1" />
    <bpmn:parallelGateway id="g1">
      <bpmn:incoming>f2</bpmn:incoming>
      <bpmn:outgoing>f3</bpmn:outgoing>
      <bpmn:outgoing>f4</bpmn:outgoing>
      <bpmn:outgoing>f5</bpmn:outgoing>
    </bpmn:parallelGateway>
    <bpmn:task id="a2" name="Prepare Piim">
      <bpmn:incoming>f3</bpmn:incoming>
      <bpmn:outgoing>f6</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f3" sourceRef="g1" targetRef="a2" />
    <bpmn:task id="a3" name="Prepare Jogurt">
      <bpmn:incoming>f4</bpmn:incoming>
      <bpmn:outgoing>f8</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f4" sourceRef="g1" targetRef="a3" />
    <bpmn:task id="a4" name="Prepare Kreem">
      <bpmn:incoming>f5</bpmn:incoming>
      <bpmn:outgoing>f10</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f5" sourceRef="g1" targetRef="a4" />
    <bpmn:task id="a5" name="Package Piim">
      <bpmn:incoming>f6</bpmn:incoming>
      <bpmn:outgoing>f7</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f6" sourceRef="a2" targetRef="a5" />
    <bpmn:task id="a6" name="Package Jogurt">
      <bpmn:incoming>f8</bpmn:incoming>
      <bpmn:outgoing>f9</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f8" sourceRef="a3" targetRef="a6" />
    <bpmn:task id="a7" name="Package Kreem">
      <bpmn:incoming>f10</bpmn:incoming>
      <bpmn:outgoing>f11</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f10" sourceRef="a4" targetRef="a7" />
    <bpmn:sequenceFlow id="f7" sourceRef="a5" targetRef="g2" />
    <bpmn:parallelGateway id="g2">
      <bpmn:incoming>f7</bpmn:incoming>
      <bpmn:incoming>f9</bpmn:incoming>
      <bpmn:incoming>f11</bpmn:incoming>
      <bpmn:outgoing>f12</bpmn:outgoing>
    </bpmn:parallelGateway>
    <bpmn:sequenceFlow id="f9" sourceRef="a6" targetRef="g2" />
    <bpmn:sequenceFlow id="f11" sourceRef="a7" targetRef="g2" />
    <bpmn:task id="a8" name="Store Products">
      <bpmn:incoming>f12</bpmn:incoming>
      <bpmn:outgoing>f13</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f12" sourceRef="g2" targetRef="a8" />
    <bpmn:exclusiveGateway id="g3" name="Premium Order?">
```

```
      <bpmn:incoming>f13</bpmn:incoming>
      <bpmn:outgoing>f14</bpmn:outgoing>
      <bpmn:outgoing>f16</bpmn:outgoing>
    </bpmn:exclusiveGateway>
    <bpmn:sequenceFlow id="f13" sourceRef="a8" targetRef="g3" />
    <bpmn:task id="a9" name="Dispatch Immediately">
      <bpmn:incoming>f14</bpmn:incoming>
      <bpmn:outgoing>f15</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f14" sourceRef="g3" targetRef="a9" />
    <bpmn:task id="a10" name="Dispatch on Monday">
      <bpmn:incoming>f16</bpmn:incoming>
      <bpmn:outgoing>f17</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f16" sourceRef="g3" targetRef="a10" />
    <bpmn:exclusiveGateway id="g4">
      <bpmn:incoming>f17</bpmn:incoming>
      <bpmn:incoming>f15</bpmn:incoming>
      <bpmn:outgoing>f18</bpmn:outgoing>
    </bpmn:exclusiveGateway>
    <bpmn:sequenceFlow id="f17" sourceRef="a10" targetRef="g4" />
    <bpmn:sequenceFlow id="f15" sourceRef="a9" targetRef="g4" />
    <bpmn:intermediateThrowEvent id="Event_1c2iwom">
      <bpmn:incoming>f18</bpmn:incoming>
    </bpmn:intermediateThrowEvent>
    <bpmn:sequenceFlow id="f18" sourceRef="g4" targetRef="Event_1c2iwom" />
  </bpmn:process>
</bpmn:definitions>
```

Example CLI and DSL specification for all provided change primitives and change patterns in order to implement the example of changes done to Dairy Process in Section 4.3 are as follows:

Insert Element

Considering the BPMN provided above, in order to achieve the Process given in Fig. 4.5, the options are as follows:

CLI

```
perturbator insert-element --insert-to ~/Desktop/input1.bpmn --element-type task --insert-at
f7 --output-file ~/Desktop/output.bpmn
```

To insert other elements, please follow the specification used for the XML notation of BPMN Diagrams, e.g., *exclusiveGateway*, *parallelGateway* for the **–element-type** argument.

DSL

```
insert' task' to process in file '~/Desktop/input1.bpmn' at flow' f7' and output in'
~/Desktop/output.bpmn'
```

Skip Element

Considering the BPMN provided above, in order to achieve the Process given in Fig. 4.7, the options are as follows:

CLI

```
perturbator delete-element --delete-from ~/Desktop/input1.bpmn --element-id a3 --output-file
~/Desktop/output.bpmn
```

DSL

```
delete element with id' a3' from process in file '~/Desktop/input1.bpmn' and output in
'~/Desktop/output.bpmn'
```

AP 1a: Insert Process Fragment (Serial)

Consider the following BPMN stored in a path ***~/Desktop/input2.bpmn*** for the process fragment shown in Fig. 4.9:

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn:definitions xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
xmlns:di="http://www.omg.org/spec/DD/20100524/DI" id="Definitions_0hjjaxj"
targetNamespace="http://bpmn.io/schema/bpmn" exporter="Camunda Modeler"
exporterVersion="4.5.0">
  <bpmn:process id="Process_0e6ngph" isExecutable="true">
    <bpmn:startEvent id="StartEvent_1">
      <bpmn:outgoing>f21</bpmn:outgoing>
    </bpmn:startEvent>
    <bpmn:task id="a21" name="Separate Raw Material">
      <bpmn:incoming>f21</bpmn:incoming>
      <bpmn:outgoing>f22</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f21" sourceRef="StartEvent_1" targetRef="a21" />
    <bpmn:task id="a22" name="Dispatch Raw Material">
      <bpmn:incoming>f22</bpmn:incoming>
      <bpmn:outgoing>f23</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f22" sourceRef="a21" targetRef="a22" />
    <bpmn:intermediateThrowEvent id="Event_0k2a9m8">
      <bpmn:incoming>f23</bpmn:incoming>
    </bpmn:intermediateThrowEvent>
    <bpmn:sequenceFlow id="f23" sourceRef="a22" targetRef="Event_0k2a9m8" />
  </bpmn:process>
</bpmn:definitions>
```

In order to achieve the process given in Fig. 4.10, the options are as follows::

CLI

```
perturbator insert-process-fragment-serial --insert-to ~/Desktop/input1.bpmn --insert-at f2 -
-insert-from ~/Desktop/input2.bpmn --output-file ~/Desktop/output.bpmn
```

DSL

```
serial insert process from ‘~/Desktop/input2.bpmn’ to process in file ‘~/Desktop/input1.bpmn’
at flow ‘f2’ and output in ‘~/Desktop/output.bpmn’
```

AP 1b: Insert Process Fragment (Parallel)

Considering the above given process fragment stored in path ***~/Desktop/input2.bpmn*** for the process fragment shown in Fig. 4.9, process perturbation given in Fig. 4.12 is done with the following options:

CLI

```
perturbator insert-process-fragment-parallel --insert-to ~/Desktop/input1.bpmn --insert-from
~/Desktop/input2.bpmn --output-file ~/Desktop/output.bpmn --branch-start f1 --branch-end f2
```

DSL

```
parallel insert process from '~/Desktop/input2.bpmn' to process in file
'~/Desktop/input1.bpmn' between 'f5' and 'f11' and output in '~/Desktop/output.bpmn'
```

AP 1c: Insert Process Fragment (Conditional)

Consider the following BPMN stored in a path ***~/Desktop/input3.bpmn*** for the process fragment shown in Fig. 4.14:

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn:definitions xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
xmlns:di="http://www.omg.org/spec/DD/20100524/DI" id="Definitions_0hjjaxj"
targetNamespace="http://bpmn.io/schema/bpmn" exporter="Camunda Modeler"
exporterVersion="4.5.0">
  <bpmn:process id="Process_0e6ngph" isExecutable="true">
    <bpmn:startEvent id="StartEvent_1">
      <bpmn:outgoing>f31</bpmn:outgoing>
    </bpmn:startEvent>
    <bpmn:task id="a31" name="Prepare Cheese">
      <bpmn:incoming>f31</bpmn:incoming>
      <bpmn:outgoing>f32</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f31" sourceRef="StartEvent_1" targetRef="a31" />
    <bpmn:task id="a32" name="Package Cheese">
      <bpmn:incoming>f32</bpmn:incoming>
      <bpmn:outgoing>f33</bpmn:outgoing>
    </bpmn:task>
    <bpmn:sequenceFlow id="f32" sourceRef="a31" targetRef="a32" />
    <bpmn:intermediateThrowEvent id="Event_0k2a9m8">
      <bpmn:incoming>f33</bpmn:incoming>
    </bpmn:intermediateThrowEvent>
    <bpmn:sequenceFlow id="f33" sourceRef="a32" targetRef="Event_0k2a9m8" />
  </bpmn:process>
</bpmn:definitions>
```

In order to achieve the process given in Fig. 4.15, the options are as follows:

CLI

```
perturbator insert-process-fragment-conditional --insert-to ~/Desktop/input1.bpmn --insert-
from ~/Desktop/input3.bpmn --output-file ~/Desktop/output.bpmn --branch-start f5 --branch-end
f11
```

DSL

```
conditionally insert process from '~/Desktop/input3.bpmn' to process in file
'~/Desktop/input1.bpmn' between 'f5' and 'f11' and output in '~/Desktop/output.bpmn'
```

Skip Process Fragment

Considering the BPMN provided above, in order to achieve the Process given in Fig 1.2, the options are as follows:

CLI

```
perturbator delete-process-fragment --delete-from ~/Desktop/input1.bpmn --output-file
~/Desktop/output.bpmn --branch-start f5 --branch-end f11
```

DSL

```
delete process fragment from process in file '~/Desktop/input1.bpmn' between 'f5' and 'f11'
and output in '~/Desktop/output.bpmn'
```

Execute DSL Commands

Assuming that all the DSL specifications are stored in a file ***~/Desktop/commands.txt***, execution via CLI is done as follows:

CLI

```
perturbator execute-from-file --command-file ~/Desktop/commands.txt
```

5 Results

5.1 Evaluation Criteria

Since the proposed solution is a software solution, the criteria being selected to determine whether the venture has been successful is also defined by the commonly used metrics in the software industry. Therefore, the criteria for the evaluation of Perturbator is as follows:

- Compliance of solution with functional requirements
- Compliance of solution with non-functional requirements
- Unit Testing
- Integration Testing

5.2 Compliance of solution with functional requirements

Referring to the requirements given in Table 3.1 of Section 3.2.1, correlating the requirements specified with the sections of this document and the adaptation patterns mentioned in Section 4 (Design) is as follows:

Func. Req. (Table 3.1)	**Satisfied by**
FR1	Implementation of Change Primitive of Insert Element (Section 4.3.1)
FR2	Implementation of Change Primitive of Skip Element (Section 4.3.2)
FR3	Implementation of Change Primitive of Insert Element (Section 4.3.1)
FR4	Implementation of Change Primitive of Skip Element (Section 4.3.2)
FR5	Adaptation Pattern 1a: Insert Process Fragment (Serial) (Section 4.3.3)
FR6	Adaptation Pattern 1b: Insert Process Fragment (Parallel) (Section 4.3.4)
FR7	Adaptation Pattern 1c: Insert Process Fragment (Conditional) (Section 4.3.5)
FR8	Adaptation Pattern 2: Skip Process Fragment (Sec. 4.3.6)
FR9	Implementation of CLI (Section 4.4.3)
FR10	Implementation of DSL (Section 4.4.2)

Table. 5.1: Functional Requirements

5.3 Compliance of solution with non-functional requirements

5.3.1 Non Functional Requirement 1: Extensibility

In table 3.2, a desire has been mentioned to write the code to extend it easily for implementing other change patterns.

In order to prove that the current implementation of the code allows implementing other change patterns quickly, some of the following patterns mentioned in **[1]** can be implemented using the features provided in Section 4. The logic according to the Perturbator that will be used to implement the below-mentioned patterns using the Perturbator solutions is as follows:

- **AP3**: Move Process Fragment
- **AP4**: Replace Process Fragment
- **AP5**: Swap Process Fragment

AP3: Move Process Fragment
This pattern aims to move one process fragment from its current position in the process model to another **[1]**. However, since the process fragment cannot be removed entirely, as discussed above in Section 4.3.2 and 4.3.6, the implementation is a variant of the pattern discussed in **[1]**. In the scope of this thesis, move means skipping the execution for the existing fragment and serial insertion of the same fragment in a sequence flow specified by the user in arguments.

The current position is the pair of ids for the sequence flows from which the process fragment to be moved begins and ends.

The target position will be taken as an input from the user, which will be the id of the sequence flow, to where the process fragment has to be inserted. Finally, from a development perspective, the building blocks for this pattern are implemented.

For now, as there is no interface to copy process fragments from one file to another, the user will have to manually copy the process fragment to be skipped in another BPMN file.

Therefore, provided that there is a starting sequence flow f_s, an ending sequence flow f_e of the process fragment PF in process P, and the target sequence flow f_t where insertion has to be done, the following steps accomplish the changing pattern for Move Process Fragment:

1. Copy process fragment to be skipped in another BPMN file.
2. Skip process fragment PF in process P starting from f_s, ending at f_e (Adaptation Pattern 2)
3. Insert process fragment PF serially from the new BPMN file at target flow f_t. (Adaptation Pattern 1a)

AP4: Replace Process Fragment
This pattern aims to replace one process fragment with another at the same position **[1].** However, since the process fragment cannot be removed entirely, as discussed above in Section 4.3.2 and 4.3.6, the implementation is a variant of the pattern discussed in **[1]**. In the scope of this thesis, replacement means the skipping execution for an existing fragment and insertion of the new fragment in the alternate path which is to be executed.

From an implementation perspective, assuming that there is a starting sequence flow f_s and the ending sequence flow f_e of the old process fragment PF_o to be replaced, and the new process fragment PF_n to replace within a separate BPMN file, the following steps are needed to implement this pattern:

1. Skip process fragment PF_o (Adaptation Pattern 2) from process P, starting from f_s and ending at f_e.
2. Insert process fragment PF_n serially (Adaptation Pattern 1a) in process P at the alternate flow f' of PF_o.

AP5: Swap Process Fragment
Since the process fragment cannot be removed entirely, as discussed above in Section 4.3.2 and 4.3.6, the implementation is a variant of the pattern discussed in **[1]**. In the scope of this thesis, a swap means skipping the execution for the existing fragments and conditional insertion of the fragment in an alternate path which should have a condition (which should force the execution towards an alternate path) to execute. Effectively, it is a combination of skipping and replacing process fragments.

From an implementation perspective, given that Adaptation Pattern 4 is implemented and there exist source and target sequence flows, f_{s1} and f_{e1} respectively, for the first process fragment PF1 and source and target sequence flows, f_{s2} and $f_{e2,}$ respectively, for the second process fragment PF2, swap is implemented as follows:

1. Copy PF1 from f_{s1} to f_{e1} in a separate BPMN file
2. Copy PF2 from f_{s2} to f_{e2} in a separate BPMN file
3. Replace process fragment (Adaptation Pattern 4) PF1 with the copied version of PF2
4. Replace process fragment (Adaptation Pattern 4) PF2 with the copied version of PF1

Implementation Details:

The only thing missing from the current implementation is the method to copy the process fragment. The logic has been implemented to insert a process fragment serially, but it has not been modularized for reuse. Once that is done, then all a new developer has to do is to define three new methods in the perturbator support module and define three new commands in three separate files with the arguments in the "commands" module, register those commands in the CLI and add new tests in the "tests" directory (Section 4.1, Figure 4.3).

Testing is also straightforward by running the following command

```
poetry run pytest
```

Once testing is complete, the developer has to run the following command to launch the updated perturbator package:

```
poetry build && poetry publish
```

5.3.2 Non Functional Requirement 2: Efficiency

In table 3.2, a desire has been mentioned to implement the tool to handle many perturbations and not consume too many valuable resources such as memory, processing power, and storage.

For this purpose, the **pytest-benchmark** package is incorporated, which describes the minimum, maximum, average time of execution of methods in seconds and the CPU rounds required to complete the execution of a method and the operation per seconds needed to execute one method.

Test Suite	Min. Time	Max. Time	Mean Time	Operations/second	Total Time
Conditional Insert Fragment	0.001935589	0.003078419	0.00225625869	443.2115893	0.539245827
Skip Fragment	0.000550108	0.001541472	0.0006110748338	1636.460781	0.617796657
Insert Element	0.000423686	0.020000764	0.0004716697503	2120.127482	0.855608927
Parallel Insert Fragment	0.001911363	0.002656122	0.002147702171	465.6139075	0.766729675
Serial Insert Fragment	0.000547649	0.001155055	0.0005919923006	1689.211159	0.748278268

Table. 5.2: Benchmarking Results

As evident from Table 5.2, It is evident that the conditional insertion of a process fragment is the most inefficient feature if minimum time is the prime evaluation factor.

Assuming that instead of running conditional insertion for one iteration, if it is run continuously in a sequential manner for **5000 iterations**; in that case, the minimum possible time taken to insert one of the more straightforward process models would be **5000 x 0.001935589** seconds, which is equal to **9.67** seconds.

Please note that the results can differ since the solution resides in the consumer's machine as a library. Therefore, the performance depends on the resources used by the consumer. Current statistics are obtained by performing the benchmarking activities on a MacBook Pro with 8GB RAM; consequently, the results are always very optimistic. With the difference in hardware, results might improve or decline.

5.3.3 Non Functional Requirement 3: Generality

This requirement desires that the system contain base functionality to allow the user to implement an array of new operations or change patterns.

As the feature to execute batch commands on a process model via Domain-Specific Language defined in a text file is available, the users themselves can perform many features continuously on a single process model. Moreover, as already discussed in Section 5.3.1, many other change patterns are the combinations of already implemented change patterns. Therefore, even if the commands are not available now, the users can manually implement those change patterns using either the CLI interface the Domain Specific Language.

5.3.4 Non Functional Requirement 4: Reusability

This requirement states that the user should be able to use the tool in various ways.

As discussed in Section 4.4.3 and 4.4.2, the users can use the tool in various ways they prefer. For example, it can be used by different software as a CLI tool, or even researchers can use it manually via the defined DSL. Although there is a lack of implementation for using a module package in the code, this will also be easily possible.

5.3.5 Non Functional Requirement 5: Usability

This requirement states that the tool must be easy to use and well-documented.

As per our discussion in Section 4.4.4, the tool has already been launched at **https://pypi.org/project/perturbator.** The webpage is provided with detailed documentation regarding the available commands and the syntax for the domain-specific language.

Installation is possible with just a single command on the terminal/command prompt, thanks to the packaging and release at PyPI. Furthermore, the rest of the documentation is also available via **–help** flag for every CLI command, the Unix standard provided by **Click**, the module used to develop the command line interface.

Moreover, to use it, one does not have to install any dependencies separately since everything is built in Python and packaged using **Poetry,** which maintains all the dependency and python version constraints.

5.4 Unit Testing

In order to verify the working of currently implemented functionality, unit tests have been written to ensure that in case of any modification or addition, the behavior of the methods implementing change primitives and change patterns is preserved.

For testing purposes, consider two fundamental process models: process P1 with two tasks and three sequence flows and process P2 with 1 task and two flows. However, the assertions are generic so that the tests will pass even if the processes are changed since the count of sequence flows, elements, and gateways are considered in specs. In order to avoid the complexity and performance of the test suite, ordering is not tested for now.

Following is the legend that is used to define the formulas being used for assertions:

Legend	**Meaning**
T1	Tasks in Process P1
T2	Tasks in Process P2
T	Total Tasks after an operation
G1	Gateways in Process P1
G2	Gateways in Process P2
G	Total Gateways after an operation
F1	Flows in Process P1
F2	Flows in Process P2
F	Total Flows after an operation

Table. 5.3: Legends used for test formulae

Following are the unit tests that have been written until now:

Serial Insert Fragment:

The implementation of serial insertion of a fragment should satisfy the following conditions:

- T1 + T2 = T (No new tasks are generated)
- G1 + G2 = G (No new gateways are generated)
- F1 + (F2 – 2) + 1 = F (Starting and ending flows are removed from P2, and a flow is added in P1 to connect to the first element of P2)

Parallel Insert Fragment:

The implementation of parallel insertion of a fragment should satisfy the following conditions:

- T1 + T2 = T (No new tasks are generated)
- G1 + G2 + 2 = G (Split and join gateways are generated, parallel gateways are counted here)
- F1 + F2 + 2 = F (Two new flows are added, one from split and one towards join gateway to establish complete branching)

Conditional Insert Fragment:

The implementation of conditional insertion of a fragment should satisfy the following conditions:

- T1 + T2 = T (No new tasks are generated)
- G1 + G2 + 2 = G (Split and join gateways are generated, exclusive gateways are counted here)
- F1 + F2 + 2 = F (Two new flows are added, one from split and one towards join gateway to establish complete branching)

Skip Fragment:

The implementation of skipping a fragment should satisfy the following conditions:

- T1 = T (No new tasks are generated)
- G1 + 2 = G (Split and join gateways are generated, exclusive gateways are counted here)
- F1 + 3 = F (Three new flows are added, alternate flow from the split gateway to join gateway, from the split gateway to the first element in the skipped fragment, from the last element in fragment to join gateway)

5.5 Integration Testing

Since most of the implementation is done using the methods for change primitives to add and remove elements; therefore, if the correctness of these two methods is tested in combination with each other accordingly, it is verified that the complete solution is working up to expectation.

Therefore, an integration test has been provided to ensure that these two methods offer the correct solution when implemented together.

Following are the conditions (using legends in Table 5.3) that should be satisfied when inserting a task and then skip an element from a process P:

- T1 + 1 = T (a new task is added during insertion)
- G1 + 2 = G (split and join gateways are added during skipping)
- F1 + 4 = F (one flow is added during insertion, one alternate flow is added from split to join gateway, one flow is added from split gateway to skipped element, and another is added from skipped element to join gateway)

6 Limitations

This section concerns the explanation of Adaptation Patterns that have been described in **[1]** but have not been yet implemented.

As far as the change primitives are concerned, **CP3** and **CP4** are implemented internally, but an interface is not provided for these primitives to add/remove sequence flows yet.

In Section 4.3, the implementation of **AP1** and **AP2** has already been discussed, which is directly supported by the Perturbator. In Section 5.3.1, it is mentioned in detail how **AP3**, **AP4,** and **AP5** could be implemented using the existing functionality and missing components from the codebase to achieve our goal of reusability from the standpoint of development.

In this section, an examination of the adaptation patterns **AP6 – AP14** is done, regarding if the Perturbator supports these patterns directly/indirectly and how it is possible to achieve such functionality using the current release of the Perturbator.

AP6: Extract Process Fragment:
The pattern requires removing a process fragment from an existing process and replace it with a subprocess containing the process fragment.

It is not supported either directly or indirectly by the Perturbator, as the subprocesses are not supported at the moment. Thus, for example, the Perturbator might be able to insert a subprocess element at a flow, but it might not be able to populate the Subprocess with the activities and flows contained inside the Subprocess.

In order to implement this, a new internal method is needed to populate subprocesses with a process fragment. The method will take the starting sequence flow ID and the ending sequence flow ID of the process fragment. If that method is available, then the following can be done to achieve AP6:

1. Populate Subprocess with the Process Fragment
2. Apply **AP2: Skip Process Fragment** to the existing Process Fragment in the Process
3. Apply **CP1: Insert Element** to the alternate flow generated by AP2 to insert the new Subprocess.

AP7: Inline Process Fragment:
Replace a subprocess with the process components inside the Subprocess.

At the moment, this pattern is also not supported directly or indirectly by the Perturbator. However, to implement this pattern, a new internal method is needed to extract the contents of a subprocess in a separate BPMN file. Furthermore, the method will only take the ID of the subprocess element as an argument. Moreover, a modification is needed in the method for **CP2: Skip Element** to return the alternate flow ID. If these pre-requisites are available, then the following logic can be used to provide AP7 as a feature:

1. Extract Subprocess Contents as a separate process in a new BPMN file.
2. Apply **CP2: Skip Element** to the ID of the subprocess element.

3. Apply **AP 1a: Serial Insert Process Fragment** to the alternate flow ID returned by **CP2** and the BPMN file generated by extracting subprocess contents in step 1.

AP8: Embed Process Fragment in Loop:
Add a loop construct over a process fragment in a process.

This pattern is not supported directly/indirectly. Nevertheless, implementing it is very straightforward, as the pre-requisites are already implemented in **AP2: Skip Process Fragment.**

In order to implement this, the only modification needed in **AP2** is to reverse the direction of the alternate flow.

AP9: Parallelize Process Fragment:
Process fragments in a sequence are replaced by branches splitting and merging on parallel gateways

This pattern is not supported directly/indirectly, as the interface to generate sequence flows is not provided yet. In order to implement this pattern, a method is needed, which is a variant of **AP2: Skip Process Fragment** is needed, which creates an alternate flow from parallel split and join gateways instead of exclusive gateways. Moreover, implementation of **AP3,** as discussed in section 5.3.1, is also needed.

Given that the variant method is available, the following logic would be implemented to achieve **AP9:**

1. Apply variant method to create an empty alternate flow from parallel split and join gateways.
2. Apply **AP3: Move Process Fragment** to move the process fragment to the newly created alternate flow.

AP10: Embed Process Fragment in Conditional Branch:
A process fragment is encapsulated with split and join exclusive gateways to be only executed if certain conditions are met.

This pattern is directly supported by the Perturbator, as it is an equivalent of the skip process fragment implementation of AP2 in the tool.

AP11: Add Control Dependency:
It is a variant of **CP3: Add Flow** which includes that the properties of soundness are preserved.

The Perturbator does not directly/indirectly support this pattern. In order to achieve this, a validator has to be written, which ensures that the modified process model follows the BPMN rules, has no deadlocks, does not have irregularities in sequence flows that violate the dependencies, and does not result in an infinite loop. These checks are a small subset of the validations required to ensure that the modified BPMN is sound and error-free.

AP12: Remove Control Dependency:
It is a variant of **CP4: Add Flow** which includes that the properties of soundness are preserved.

The Perturbator does not directly support this pattern as it needs the validator to check if the Process is sound or not, as discussed above in **AP11.**

AP13: Update Condition:
Update condition for an alternate flow for a branch generated due to splitting after an exclusive gateway.

Currently, it is not supported directly/indirectly by the Perturbator. In order to implement this pattern, a generic method is needed that takes element ID, attribute key and attribute value as arguments. The method will parse the BPMN, update the key-value pair for the element with ID passed as an argument, and write the updated BPMN in the output file.

AP14: Copy Process Fragment:
It is a variant of AP3, where the process fragment is not skipped from the original position.

At the moment, it is not supported directly/indirectly by the Perturbator, as it requires the implementation of **AP3** as discussed in Section 5.3.1.

In order to implement this, a boolean argument in **AP3** is needed to check if the fragment is being copied, then it does not apply **AP2: Skip Process Fragment** to the process fragment**.**

7 Conclusion

In this thesis, business process modeling concepts were discussed, and the document addressed the need for a tool that allows making perturbations to existing business process models programmatically. However, the existing tools are limited in their features, and there is a lack of a tool that will enable implementing change patterns in business process models, which are the standardized operations for process models.

The perturbator tool developed and released due to this thesis research can perform basic change patterns and change primitives needed to combine and implement other change patterns. It is also usable in multiple formats, either via a CLI or a domain-specific language, which allows using the tool for several purposes. For example, a significant application of the tool is to generate an extensive collection of different possible perturbations of a process model to identify further process optimization work opportunities. This application would be especially useful in risk-based process optimization, where the structure of the Process has to be modified to calculate the cost or profit gained by removing/adding tasks.

Limitations

The development is at an initial stage where the focus was mainly on implementing the fundamental change patterns and providing an extensible and flexible base framework for further evolution of the tool. Therefore, it is impossible to carry out any analysis to validate that the business process model will preserve its correct properties after the perturbation has been applied to it.

Currently, a ready-made solution is available only for limited change patterns, which can be increased. However, the base functionality offered is enough to implement different change patterns as a combination of the existing operations.

Another limitation is that even though the system can be used via a CLI or apply batch commands using the text file format according to a defined domain-specific language, the Perturbator does not provide support for the tool to be used as a module and be able to provide functionality as function calls from within the code. However, as a workaround, the developers can execute operations by running shell scripts on their servers.

Another thing that is missing from the implementation is the support to update the BPMN diagram as well. Currently, the structure of the BPMN process in the BPMN file is updated, but the BPMN diagram components are ignored for now. As a result, users can observe the correctness by correlating the IDs but cannot view the updated BPMN files in the diagram viewer, such as those implemented in Camunda.

Future Work

The research and development for this tool can take the following directions in the future:

- Provide built-in support for a more extensive collection of the change patterns discussed **[1].**

- Formulate and implement the logic to verify the correctness **[3]** of a process model if a perturbation has been applied to it and halt the user from doing an operation if it violates the properties of business process models.
- To make the tool even more developer-friendly, provide built-in support for the existing functionality to be used as a library within the code, along with the CLI and DSL options for easier integration as an engine for business process optimizers.
- Provide support for the diagram update so that the perturbated process models can be viewed in GUI tools.
- PIX initiative (Section 2.4) will reuse the Perturbator as an easy solution that allows the rest of the PIX optimization components to generate multiple variants of different business processes to solve business process optimization problems.
- Contrary to the current research at the University of Tartu for process optimization based on Resource Allocation, the Risk-based Business Process Optimization deals with finding the optimal business processes resulting from skipping or adding new elements to a business process. Perturbator will be used as the tool in this optimization problem to generate possible alternates for calculation.

8 References

[1] Weber, B., Rinderle, S., & Reichert, M. (2007, June). Change patterns and change support features in process-aware information systems. In *International Conference on Advanced Information Systems Engineering* (pp. 574-588). Springer, Berlin, Heidelberg.

[2] *Experience the Impact of Change Patterns on the Modeling Process*. (n.d.). Business Process Management Research Cluster. http://bpm.q-e.at/wp-content/uploads/2015/04/InstructionsCPExpWithMoveWeb.pdf

[3] Rinderle, S., Reichert, M., & Dadam, P. (2004). Correctness criteria for dynamic changes in workflow systems—a survey. *Data & knowledge engineering*, *50*(1), 9-34.

[4] Dumas, M., La Rosa, M., Mendling, J., & Reijers, H. A. (2013). *Fundamentals of business process management* (Vol. 1, p. 2). Heidelberg: Springer.

[5] Dumas, M., van der Aalst, W., & Ter Hofstede, A. (Eds.). (2005). *Process aware information systems* (Vol. 1). Chichester: Wiley.

[6] Gschwind, T., Koehler, J., & Wong, J. (2008, September). Applying patterns during business process modeling. In *International Conference on Business Process Management* (pp. 4-19). Springer, Berlin, Heidelberg.

[7] *Chapter 1: Change Patterns Set*. (n.d.). Business Process Management Research Cluster. http://bpm.q-e.at/wp-content/uploads/2015/04/Explanation_CP_woMCP.pdf

[8] *The Process Improvement Explorer (PIX)*. (n.d.). Software Engineering Group. https://sep.cs.ut.ee/Main/PIX

Appendix

I. Glossary

BPM	Business Process Model
BPMN	Business Process Modelling and Notation
CLI	Command Line Interface
DSL	Domain-Specific Language
GUI	Graphical User Interface
P	Process
PAIS	Process-Aware Information Systems
PF	Process Fragment

II. Command Line Interface

Insert Element:

```
perturbator insert-element --insert-to <sourceFile> --element-type <elementName> --insert-at
<flowId> --output-file <outputPath>
```

Where,
<*elementName*>*:* name of the element to be added
<*sourceFile*>: an absolute path of the BPMN file to which element has to be added
<*flowId*>: ID of the sequence flow where the element has to be added
<*outputPath*>: an absolute path of the BPMN file where the output has to be written

Skip Element

```
perturbator delete-element --delete-from <sourceFile> --element-id <elementId> --output-file
<outputPath>
```

Where,
<*elementID*>*:* ID of the element to be removed
<*sourceFile*>: an absolute path of the BPMN file from which element has to be skipped
<*outputPath*>: an absolute path of the BPMN file where the output has to be written

Insert Process Fragment (Serial):

```
perturbator insert-process-fragment-serial --insert-to <targetFile> --insert-at <flowId> --
insert-from <sourceFile> --output-file <outputPath>
```

Where,
<*sourceFile*>: an absolute path of the BPMN file which is to be inserted
<*targetFile*>: an absolute path of the BPMN file to which insertion is to be done
<*flowId*>: ID of the sequence flow where process fragment is to be inserted
<*outputPath*>: an absolute path of the BPMN file where the output has to be written

Insert Process Fragment (Conditional):

```
perturbator insert-process-fragment-conditional --insert-to <targetFile> --insert-from
<sourceFile> --output-file <outputPath> --branch-start <initialFlowId> --branch-end
<finalFlowId>
```

Where,
<*sourceFile*>: an absolute path of the BPMN file which is to be inserted
<*targetFile*>: an absolute path of the BPMN file to which insertion is to be done
<*initialFlowId*>: ID of the sequence flow where the split gateway is to be added
<*finalFlowId*>: ID of the sequence flow where join gateway is to be added
<*outputPath*>: an absolute path of the BPMN file where the output has to be written

Insert Process Fragment (Parallel):

```
perturbator insert-process-fragment-parallel --insert-to <targetFile> --insert-from
<sourceFile> --output-file <outputPath> --branch-start <initialFlowId> --branch-end
<finalFlowId>
```

Where,
<*sourceFile*>: an absolute path of the BPMN file which is to be inserted
<*targetFile*>: an absolute path of the BPMN file to which insertion is to be done
<*initialFlowId*>: ID of the sequence flow where the split gateway is to be added
<*finalFlowId*>: ID of the sequence flow where join gateway is to be added
<*outputPath*>: an absolute path of the BPMN file where the output has to be written

Skip Process Fragment:

```
perturbator delete-process-fragment --delete-from <sourceFile> --output-file <outputPath> --
branch-start <initialFlowId> --branch-end <finalFlowId>
```

Where,
<*sourceFile*>: an absolute path of the BPMN file which is to be inserted
<*initialFlowId*>: ID of the sequence flow where the split gateway is to be added
<*finalFlowId*>: ID of the sequence flow where join gateway is to be added
<*outputPath*>: an absolute path of the BPMN file where the output has to be written

Execute batch commands from text file, defined using the Domain Specific Language:

```
perturbator execute-from-file --command-file <commandPath>
```

Where,
<*commandPath*>: an absolute path of the text file containing commands written in DSL discussed in Section 4.4.2

III. Definition of Domain-Specific Language

```
Model: commands*=PerturbatorCommand;
```

```
PerturbatorCommand: InsertElementCommand | DeleteElementCommand |
InsertSerialCommand | InsertConditionalCommand | InsertParallelCommand |
DeleteFragmentCommand;
```

```
InsertElementCommand: 'insert' elementType=STRING 'to process in file'
path=STRING 'at flow' flow=STRING 'and output in' outputPath=STRING;
```

```
DeleteElementCommand: 'delete element with id' elementId=STRING 'from
process in file' path=STRING 'and output in' outputPath=STRING;
```

```
InsertSerialCommand: 'serial insert process from' extractPath=STRING 'to
process in file' path=STRING 'at flow' flow=STRING 'and output in'
outputPath=STRING;
```

```
InsertConditionalCommand: 'conditionally insert process from'
extractPath=STRING 'to process in file' path=STRING 'between'
startingFlow=STRING 'and' endingFlow=STRING 'and output in'
outputPath=STRING;
```

```
InsertParallelCommand: 'parallel insert process from' extractPath=STRING
'to process in file' path=STRING 'between' startingFlow=STRING 'and'
endingFlow=STRING 'and output in' outputPath=STRING;
```

```
DeleteFragmentCommand: 'delete process fragment from process in file'
path=STRING 'between' startingFlow=STRING 'and' endingFlow=STRING 'and
output in' outputPath=STRING;
```

IV. License

Non-exclusive license to reproduce thesis and make thesis public

1. I, Zohaib Ahmed Butt, herewith grant the University of Tartu a free permit (non-exclusive license) to reproduce, for preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, "Programmatic Perturbation for Business Process Models," supervised by Marlon Dumas and Orlenys Lopez Pintado.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons license CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive license does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Zohaib Ahmed Butt
12/05/2021