Replay analysis in generic process modeling language

Nguyen, H.V.

Award date:
2011

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science
Architecture of Information System Group

Replay Analysis in Generic Process Modeling Language

Master thesis

Nguyen Hong Viet

Supervisors:
dr.ir. B.F.van Dongen
ir.Arya.Adriansyah

Eindhoven, January, 2010

Abstract

Process models are extensively used in many information systems not only as process descriptions, but also as means of analysis. Typically, models are assumed to be fully conforms to operational processes, but experience shows that this is not the case. To identify deviations between operational process and its specification, log replay analysis is performed. Given a process model and process executions recorded in an event log, log replay techniques confront the model to the log to extract useful information for further analysis such as conformance, performance. However, the techniques require models to be in specific modeling languages, which then limit their applicability.

In this thesis, we extend currently existing replay log techniques to be independent from any specific process modeling languages. We propose a unification approach based on a generic process modeling language that abstracts away the requirement of log replay to have a specific process modeling language. We propose conversion methods from several process modeling languages to the generic process modeling language that preserves essential information necessary for log replay. Based on existing replay techniques, we develop a log replay technique that works on the generic process modeling language and investigate its characteristics. To demonstrate the applicability of our approach, we have implemented it in the ProM[1] framework and tested it using simulated logs and process models.

Keywords: Log Replay, Process Analysis; Conformance checking;

[1] See http://www.processmining.org

Preface

This master thesis is the result of my graduation project which completes my master Computer Science & Engineering at the Eindhoven University of Technology. The project was performed internally at the Architecture of Information Systems group of the Mathematics and Computer Science department.

First of all, I would like to thank my graduation supervisor Boudewijn van Dongen and my graduation tutor Arya Adriansyah for their advices and supports during my master project.

Furthermore, I would like to thank my parent, my brother and my girlfriend for supporting me all the time. I also would like to thank Denny for helping me to deal with programming problems.

Hong Viet
Jan, 2011.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis context

Nowadays, many organizations, either small or large, are using information systems to support their operational processes. An operational process can be thought of a procedure containing a number of activities taken to achieve certain goals. Examples of operational processes are a "Laptop repair handling" process in a computer shop or "Order item processing" procedure in a web shop. Most of operational processes are described in the so-called process models which are expressed in term of process modeling languages. Process models usually describe activities and resources involved in operational processes. Currently, there is a large number of process modeling languages ranging from languages with relaxed semantic (e.g. Fuzzy model) to languages with strict semantic (e.g. Petri Net). Each language has its own advantages and disadvantages. Therefore, organizations often use more than one process modeling language to describe a single operational process.

As organizations always wish to improve their business, process analysis thus becomes an essential part for increasing the business performance. Analysis is performed to identify possible faulty behaviors, possible bottle necks or process patterns in a process or even to predict the outcome result of a process execution. However, most of currently existing analysis techniques work only for specific process modeling languages. This brings up problem when organizations want to perform analysis over process models. For instance, given two models of the same process specified by two different process modeling languages, one can ask a question which one conforms better to the given process. This kind of question cannot be answered using existing analysis techniques that are invented only for certain process modeling languages.

Instead of developing an analysis method for each process modeling language, this thesis proposes a unification approach that alleviates the problem by abstracting all process models to a common format. Analysis techniques will be then investigated based on this format.

1.2 Project goal

As discussed in the previous section, problem occurs when we wish to perform analysis of a process based on different process models. In this thesis, we look at one typical analysis technique called ***Log Replay***. *Log replay* is used to analyze performance of a process model or to check conformance between a given model and an event log. The *Log Replay* currently exists only on Petri Net, Fuzzy model and Flexible model. There are no replay techniques invented for other existing process modeling languages. Hence the goal of the project is concluded as follow:

***Goal of the project**: Enable the log replay on a variety of existing process modeling languages.*

1.3 Research questions

In order to obtain the project goal, a couple of research questions are formulated. Each research question is split up into several sub-questions. The answers of sub-questions lead up to the answer of the main question.

Rather than creating a replay method for each process modeling language, the proposal approach is to enable the log replay method based on a common type of process model which can abstract a wide variety of process models. Translations into this new format are necessary to support replay techniques on existing process modeling languages. Thus, the first step is to generalize different exiting process models into a common process format. This brings up the following first research question:

Research Question 1: What generalization techniques exist for process modeling languages?
This main question will be split up into the following sub-questions.

- **Sub research question 1.1**: What are the possible formats to which the process modeling languages can be generalized and what are their structures?
- **Sub research question 1.2:** What information is preserved and what information is lost while generalizing process models?
- **Sub research question 1.3:** How are the elements of existing models mapped to the elements of new process format?

The initial step for answering this question is to find out all possible formats which we can use for abstracting process models. Besides, understanding their structures, advantages and disadvantages will help us to pick up the most appropriate one. It should be noticed that model transformation does not always preserve information. Thus, we need to determine which information we have to lose and which one we can keep when we perform generalizing. Moreover, the generic formats and existing models have their own structures, notations and semantics. Therefore, it is important to do correct mapping between abstracted models and generic models such that the important behavior is preserved as much as possible.

After choosing a new model as a generic format, we need to apply the current existing replay techniques on that model. The second research question is then brought to us as follow:

Research Question 2: How can the currently existing reply techniques used on the chosen generalizing format?
The second question is also divided into three sub-questions. Answering these will provide us the solution for the main one.

- **Sub research question 2.1:** What are the current replay techniques and how do they work?
- **Sub research question 2.2:** Which is the most applicable replay technique for the chosen format and how it can be applied?
- **Sub research question 2.3:** What parameters are the most suitable for replaying on the generic format?

At the moment, there are a number of reply methods. However, all of them work only for some specific process models. Thus, it is essential to discover all available methods as well as their supporting process models. Gaining knowledge of how do they work, what are advantages and disadvantages will help us to choose the most applicable replay method. If the selected replay technique requires some pre-configured parameters then it is desirable to perform some experiments to find the most suitable parameter setting.

1.4 Research approach

The project goal can be achieved by executing a number of necessary steps in the following order

Step 1: Perform literature study and gather relevant information

The earliest step is to gain knowledge of some relevant subjects that are important to our project work, such as the basic concepts of process modeling languages, process mining, event log, log replay technique and ProM framework. The result of this literature study is presented in Chapter 2.

Step 2: Investigate currently existing replay methods

The next step is to perform an investigation on existing log replay techniques. We investigate how they can be implemented, what advantages, problems or limitations they have. Furthermore, we undertake an analysis to find out whether and how these replay techniques can be applied to the selected format. The result of this step is the answer for question: what is the most applicable replay technique. The detail of this step is provided clearly in Chapter 3.

Step 3: Generalize process modeling languages

The first phase of this step explained in Chapter 4 is to define the motivation to generalize process modeling languages. Also in this step, a couple of possible generic formats that can be used for abstract process models must be introduced. A comparison between those candidates needs to be taken in order to find the most appropriate one. Next, we investigate how to convert existing process models into selected generic format. At the end of this step, we implement conversions from process modeling languages into the selected generic format using process mining framework ProM.

Step 5: Design log replay on chosen generic format

In this step, we investigate how to perform log replay on the selected generic format. The replay solution of generic format should be implemented in ProM. The detail of this step is presented in Chapter 5.

Step 4: Perform case study experiments

The generic format might require pre-configured parameters for replay technique. Different settings lead to different replay result. In order to obtain the most optimal parameter setting, a validation needs to be taken by performing several case studies. This is demonstrated in Chapter 6.

Step 5: Conclude

The last chapter is the conclusion of this thesis. In this chapter, the project approach will be evaluated to see whether or not the project‘s goal is achieve. This chapter also includes a discussion concerning future works which are about to solve the remaining problems and to improve the result of project.

Chapter 2

Preliminaries

This chapter introduces basic conceptual foundations which will be used throughout this thesis. The chapter begins in Section 2.1 providing explanation about the process modeling languages. Also in this section, some particular modeling languages are described in more detail. In next section, Section 2.2, definition of semantic independent process modeling languages is given. Definition of Event log and Log replay are introduced in Section 2.3. Finally, in Section 2.4 we introduce the notion of process mining technique and ProM framework.

2.1 Process modeling languages

Many organizations use information systems to support their business operation. One of the core elements of information systems is process models. Process models describe, often in some graphic notations, how a certain process is composed from a number of different tasks, which resources are involved to carry out these tasks and which objects are influenced in the process. Process models can be used both within the context of IT deployment or for more business-oriented purposes. They are often expressed using process modeling languages. Business Process Modeling Notation (BPMN), Event-Driven Process Chains (EPC), Petri Net, Yet another Workflow Language (Yawl), BPEL (Business process execution language) and Fuzzy model are examples of process modeling languages. All of these process modeling languages vary in respect to their behavior, structure, graphic representation and semantics. In this thesis, we will not introduce all of them. Instead, we choose to describe four modeling languages related to our coming work, namely Petri net, Yawl, Fuzzy and EPC. We select Petri net because it is one of the mostly investigated process model. Moreover, Petri net currently support log replay technique that we are going to investigate. Fuzzy model is investigated as there is already a log replay invented for it. EPC and Yawl are selected since EPC is the most widely process modeling languages used in industry and Yawl is an expressive language to describe complex process models and provides comprehensive support for workflow pattern [1].

2.1.1 Petri net

Petri net [2] is a process modeling language described in the form of directed bipartite graph containing transitions, places and arcs. The role of transitions and place are defined as follow:

Transitions, which are denoted by rectangles in Petri net graph, correspond to activities that occur in the process execution. Transitions are also used to indicate the “silent” steps that take care of routing of control flow or to delay the activity execution.

Places, which are represented by circles, specify the pre/post- condition of a certain transition**.** Places also indicate the start and end of a process model. Generally, a Petri net can contain multiple starting/ending places.

Directed arcs link a place to a transition or vice versa, but never between transitions or between places. Places with outgoing arcs pointing to a transition are called input places of the transition. Places with incoming arcs running from a transition are called output places of the transition. Tokens denoted by black dots define execution behavior of the model. Multiple tokens can be held in any places, even in the same place. A transition is enabled as soon as there is at least one token in each of its input places. If a transition is enabled, it may be fired. When a transition fires, it consumes a token from each input place and produces a token in each output places. The distribution of tokens over places is called a marking of the net. Petri net use notion of initial marking to indicate from which transitions the process can start. Firing a transition creates a new marking by removing tokens in input places and adding new tokens in output places. The execution of Petri net is non-deterministic, i.e. when multiple transitions are enables at the same time, any one of them can be fired in any order. Therefore, this makes Petri suitable for modeling concurrent behavior.

Petri net captures mostly used workflow patterns, such as: sequential pattern, parallel split pattern, synchronization pattern, exclusive choice pattern and simple merge pattern. However, Petri net does not support OR split/join patterns. To model OR split/join constructs, a number of extra transitions which are not related to any activities in process are needed. These transitions are regarded as invisible transitions or silent steps. An example of a Petri net model containing basic control flow patterns is shown in Figure 2.1.

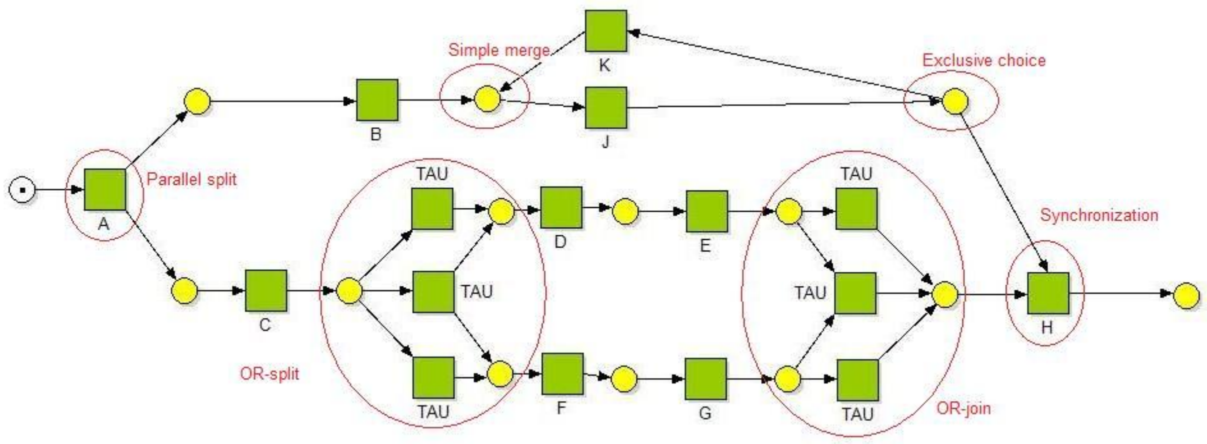


Figure 2.1: An example of Petri net

2.1.2 Yet another workflow language

Yet another workflow language (Yawl) [3] is specially developed to capture most of the existing workflow patterns [1]. Since Petri net supports almost basic control patterns, it is taken as a starting point for designing Yawl. Yawl extends Petri net by adding four new constructs, namely *cancellation set*, *OR-join*, *multiple instances activities* and *composite task*. These constructs aim at supporting some advanced workflow patterns that are not directly supported in Petri net, for instance synchronized merge, multiple merge, cancellation pattern, discriminator or patterns involving multiple instances. More detailed information about how Yawl supports those advanced patterns can be found in [3]. Although, Yawl is invented based on Petri Net, it cannot be regarded as an extended high level of Petri net. It is completely a new language with its own independent semantics. Generally, the Yawl model consists of tasks, conditions and flows.

Tasks, denoted by rectangle boxes with task's name inside, can be either atomic tasks or composite tasks. Atomic tasks represent activities occurring in the process. Composite tasks refer to other Yawl process model at low level in hierarchy. Especially, both composite tasks and atomic tasks can have multiple instances.

Conditions, denoted by circles, can be interpreted as places in Petri net. Conditions are also used to indicate the pre/post-condition before and after firing a task. Unlike Petri net, conditions can be implicitly represented in Yawl, i.e. they can be hidden in between tasks. Yawl model must contain exactly only one input condition marking the start of the process and only one output condition marking the end of the process.

In contrast to Petri net, the flows can run directly from one task to other task without passing through a condition in between. However, there is no flow in between conditions. Additionally, Yawl provides extra syntactically elements to intuitively capture other workflow patterns which are not available in Petri net, such as: simple choice, simple merge multiple choice and multiple merge. These patterns are semantically bound to the task's behavior, i.e. the behavior of parallel routing patterns and execution of activities are together represented on a same construct. Figure 2.2 shows an example of Yawl model with the same behavior as the Petri net in Figure 2.1.

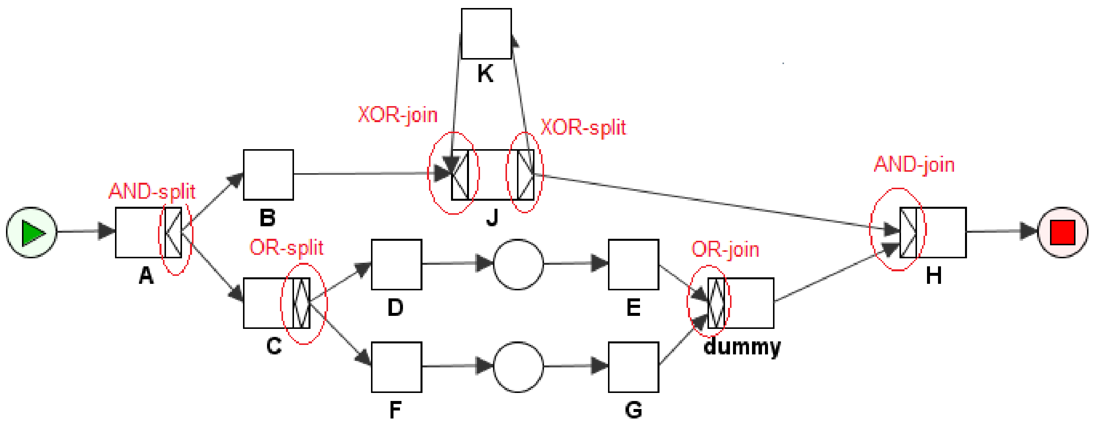


Figure 2.2: An example of Yawl

2.1.3 Event driven process chain

Event driven process chain (EPCs) [4] is another type of workflow language used for business process modeling. EPC is an industrial process mining language that is widely used by many companies for modeling, analyzing and redesigning business process. An EPC graph consists of three types of vertexes which are events, functions and connectors.

Events, represented by hexagons, describe in which states the process seize after executing a function or under what circumstance a function is enabled. In general, an EPC model must start with an event and end with an event. However, it is possible to have multiple starting events and multiple ending events.

Functions, denoted by rectangles, are similar to tasks in Yawl and transitions in Petri net. They are used to model activities executed in the process execution.

Connectors, denoted by circles, indicate the flow of control in EPC. Unlike Petri net and Yawl in which split/join behaviors are bound to the task's construct, in EPC these routing behaviors must be represented separately in the so-called connector constructs. There are three types of connectors, namely and-connector, exclusive-connector and or-connector. There is no explicit semantic to define a connector as a split or join. It depends on the number of incoming arcs and outgoing arc. For instance: an and-split connector has only one incoming arc and at least two outgoing arcs. Whereas, an and-join connector has at least two incoming arcs and only one outgoing arc. A connector can be either a split or join but cannot play both roles at the same time.

One should be noticed that there is no cycle containing connectors only in EPC model. Moreover, events are not allowed to precede xor-split connector and or-split connector. Every arc connects two different vertexes. Every vertex is on a path from a starting event to a final event. Events and functions must alternate along the control flow path, i.e. there is no path with two functions without an event in between and the other way around. An example of EPC is illustrated in Figure 2.3.

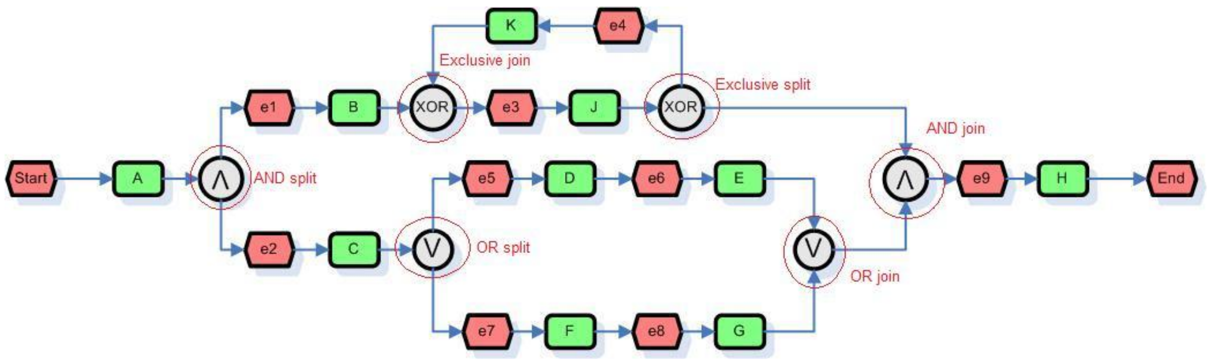


Figure 2.3: An example of EPC

2.1.4 Fuzzy model

Definition of Fuzzy model was first introduced in [5]. Fuzzy model is designed to visualize complex process models (e.g. spaghetti process models) in an understandable way. The idea of Fuzzy model is to abstract unimportant activities and aggregate highly related activitiess whose behavior are not interesting into a cluster. Fuzzy model is represented by a graph consisting of nodes and arcs. Nodes are specialized into cluster nodes and primitive nodes. While primitive nodes denoted by rounded rectangle represent activities in process execution, cluster nodes denoted by octagons aggregate events whose behavior is not of interest. Fuzzy model cannot be used to control process enactment on workflow system since it only provides an efficient and simplified visualization of processes. The main character of Fuzzy model is the relaxed execution semantics. Detailed description of relaxed execution semantics in term of workflow pattern can be found in [5]. In this thesis, we only summarize important execution semantics of Fuzzy model as follows:

- **Branch semantics**, every node in Fuzzy model has *AND-split* semantic. Whenever a node has been executed, it will enable all its successors. However, enablement does not enforce execution, thus the successors must not be executed. Moreover, if an enabled node has not been executed, it's enabling remains the same.

- **Join semantics**, every node in Fuzzy model has memory-less XOR-join semantic. A node is enabled as soon as one of its predecessors has been executed. Thus a node can be enabled multiple times by multiple predecessors. However, it does not keep track of how often it has been enabled.

- **Initialization,** the process can start at any arbitrary node in Fuzzy model. Therefore starting nodes can have multiple incoming arcs, i.e. there is no exclusive starting point.
- **Termination**, the process can terminate at any subset of node, i.e. there is no exclusive ending point. Furthermore, the process is considered to be terminated if there is no further execution of nodes in the model.

It is observed that, when a node in Fuzzy model has been executed, it enables all of its successors. Any subset of enabled successors can be then executed. However, all of them are not forced to be fired. Hence there is no explicit distinction between simple choice (XOR-split), multiple choice (OR-split) and parallel split (AND-split). In Figure 2.4, we show an example of Fuzzy model compared to the Yawl model in Figure 2.2. This is a simplified model, where task “K” has been abstracted and task C, D, F, E and G has been hidden inside a cluster node.

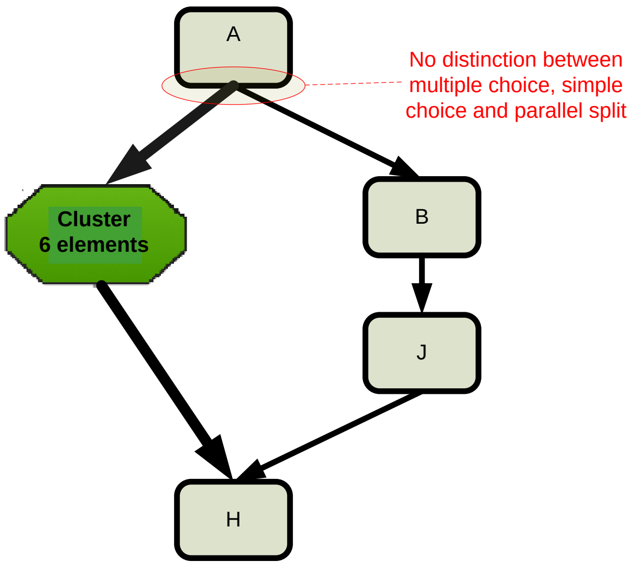

A
No distinction between multiple choice, simple choice and parallel split
Cluster 6 elements
B
J
H

Figure 2.4: An example of Fuzzy model

2.2 Generic process modeling languages

Generic process modeling languages are invented for generalizing existing process models. The idea of generalizing is to represent semantic independent from process modeling languages, i.e. it leaves away semantics of existing process models such that analysis independent from specific semantic can be performed. Generic process modeling languages are not intended to be yet another modeling notation, it is rather be a language that captures the core split/join behaviors of process models independent from modeling languages. There are two types of process models can be used to generalize existing process models, namely *Flexible model* and *Canonical Process Format.*

Flexible model has been developed within Technical University of Eindhoven in the Netherlands. The idea of flexible model is to perform log replay indirectly on any existing process models as long as the models can be expressed as a flexible model.

Canonical Process Format (abbreviated as CPF) has been developed in Queensland University of Technology in Australia. It is original used for supporting APROMORE [6] that is an advanced process model repository. APROMORE helps organizations manage a massive collection of process models by offering a rich set of advanced features such as maintaining, analyzing and exploiting the content of process models. In the coming subsequent sections, we will introduce definition of flexible model and canonical process format in more detail.

2.2.1 Flexible model

A flexible model [7] is a directed graph composed from tasks and arcs. Tasks denoted by rounded rectangles represent either activities or silent steps performed in a process execution. Moreover an activity can be represented by more than one task in a flexible model, i.e. duplicate tasks are permitted in a flexible model. Arcs are used to connect tasks to each other. The term “flexible” is used to emphasize that the model is able to accommodate different routing semantics ranging from relaxed to strict semantic. Flexible model can capture all kinds of split/join patterns including the OR split/join construct.

Each task in a flexible model has an input set and an output set. Input set of a task consists of possible combinations of task’s predecessors, while its output set consists of possible combinations of the task’s successors. Unlike EPC in which routing behaviors are explicitly represented in separate nodes called connectors, routing behaviors are bound to task node in flexible model. Once a task is executed, it enables alternative combinations of successors from its input set. In addition, a task is only enabled if one of combinations from its output set is executed in front. Precise or relaxed split/ join behaviors depend on specifications of input/output sets.

In Figure 2.5, we illustrate a flexible process model containing some workflow patterns. As we can see, the split/join flows are specified by the declaration of the input/output sets. For instance, the output set of task *A* consists of exactly one set which contains both task *B* and *C*. This indicates the AND-split routing, i.e. after task *A* is executed, task *B* and *C* are both enabled. Whereas, the output set of *B* consists of two disjoint sets. One contains only task *M* and the other contains only task *N*. This specifies the XOR-split behavior in task *B*, i.e. either *M* or *N* will be carried out after the execution of *B*. The output set of task *C* declares an OR-split pattern, since it contains all alternative combinations of task *C*’s successors. Task *K* indicates an OR-join construct since its input set consists of all possible combinations of task *K*’s predecessors. The AND-join and XOR-join are specified by the input set of task *J* and *W,* respectively. Furthermore, a more relaxed synchronization and choice situation can also be expressed via the specification of input and output sets. For example, the output set of *D* indicates that execution of *D* enables either *G* or both *E* and *F*. This kind of choice is neither an OR-split nor a XOR-split, it is somewhat in between. Similarly, before task *G* can be enabled, *D* must be executed or both *E* and *F* must be executed. Thus flexible is capable of modeling different types of split/join semantics.

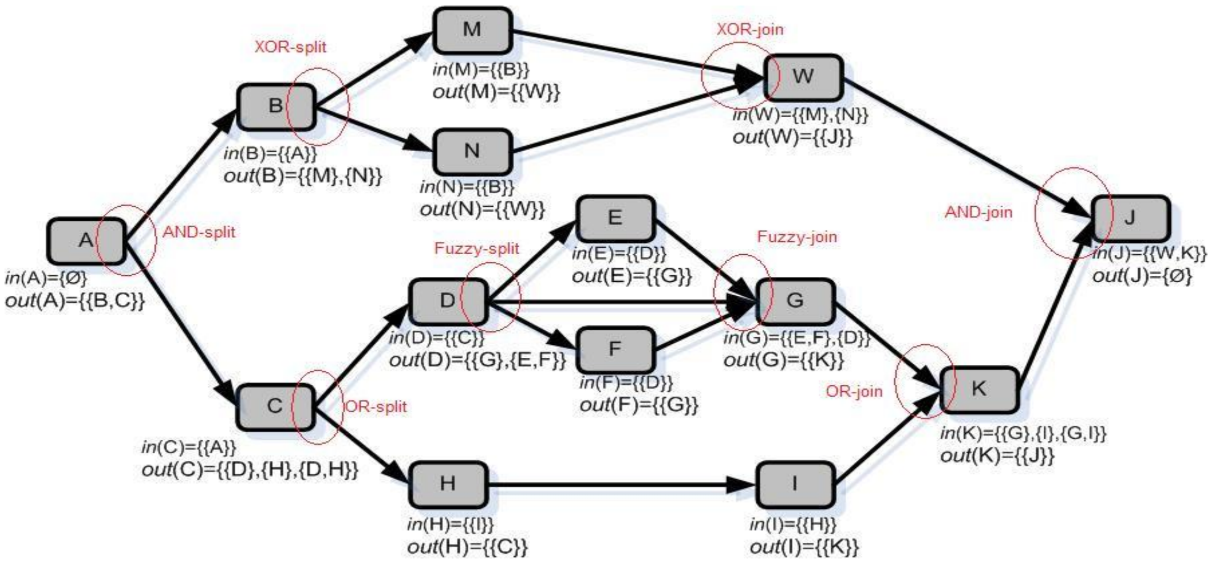


Figure 2.5: An example of Flexible model

2.2.2 Canonical process format

Canonical process format (CPF) [6, p.5-10] is a process format providing an unambiguous representation of business processes captured in different notations. The main idea of CPF is to represent only the structure characteristic of a process model that can be expressed in different kinds of process modeling languages. In general, a CPF model is a directed attributed graph consisting of two elements, nodes and edges both denoted by black circles.

Edges are similar to arcs in Petri net which are used to link nodes. Edges might contain an attribute, namely *“condition”* used to represent the conditions upon which a choice is made.

Nodes can be either of type Routing or Work. Routing nodes are used only for capturing the control flow in the process, i.e. no business perspective is performed in Routing nodes. Routing nodes are classified into Split, Join and State nodes. Split is specialized to XOR-split, OR-split and AND-split. Similarly, Join is composed from XOR-join, OR-join and AND-join. States are used to indicate the state before an event-driven decision is made or after a merge. Splits have exactly one incoming edge and at least two outgoing edges, Joins have at least two incoming edges and exactly one outgoing edge, whereas States can have multiple incoming and outgoing edges. With the existence of AND, OR and XOR split/join nodes, CPF is capable to capture core split/join behaviors.

Different from routing nodes, Work nodes capture information that is involved in business perspective. Work nodes contain at most one incoming edge and one outgoing edge and can be partitioned into Tasks and Events. Tasks model some activities performed in business process. Task nodes can be either atomic or composite tasks. Composite tasks refer to other CPF model in low level hierarchy. Events are used to signal something happening during the process execution. Event nodes can be divided into Message events (capturing a message being sent or receipt) and Timer events (capturing a timeout or a delay of a task). Work nodes are often associated with ResourceTypes and Objects. ResourcesTypes refer to something that carry out the tasks and can either be human (e.g. people in the organization) or non-human resources (e.g. automatic information systems). Objects refer to the business objects involved in the process and can be physical objects (e.g. document, paper invoice) or information objects (e.g. file, digital document).

A meta-model of the canonical process format defined using UML notation is shown in Figure A.1 in Appendix A. In Figure 2.6, we depict an example of canonical process format that describes the same behavior as the Petri net model in Figure 2.1.

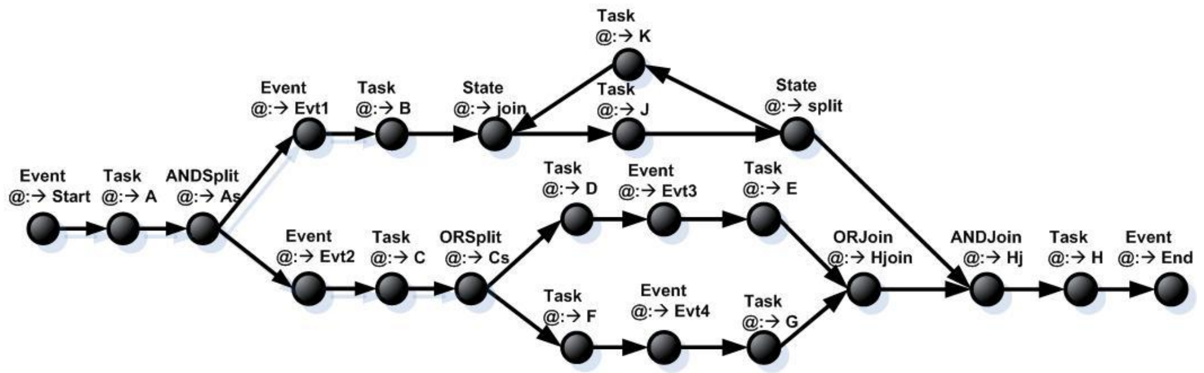


Figure 2.6: An example of CPF model

2.3 Event log

Information systems used by organizations usually recorded activities performed during process execution into event logs. Whenever a process is started, it will trigger a process instance which is called an event case. A process instance contains a trace of events that are executed for that case. Each event refers to an instance of a certain activity performed during the process execution. In addition, events are often associated with attributes such as timestamp attribute and resource attribute. Timestamp indicates the moment when the event is executed. Resource indicates something that carries the activity represented by the event and can be either a user or a system. Moreover, events in a case are ordered to specify in which routine the activities have occurred. For most of the cases, the order is defined according to the timestamp attribute of the event. Hence,an event log can be defined as a collection of event traces. Event log is usually taken as the starting point for process mining technique which will be introduced in the next section.

2.4 Process mining and ProM framework

Organizations use information systems to support and control their operational activities. Over the time, these systems record everything observed from real world into event logs. There are always questions like "how does the actual operational process look like?", "Does the process model conform to some specifications?", "are there any bottle-necks in the current process model?" etc. *Process mining* comes up to answer such kinds of question by looking at event logs. Process mining is defined as a technique that allows for extracting useful information from event logs. The basic idea of process mining is to discover process models or indentify the deviation by comparing observed events with some pre-defined models. As illustrated in Figure 2.7, process mining mainly focus on Process discovery, Conformance and Extension.

- **Discovery**, aims at constructing new process models from existing event log when there is no formal description of processes.
- **Conformance**, aims at analyzing the discrepancy between a log and a pre-defined model by comparing that model with observed events.
- **Extension**, aims at improving existing models based on some extracted information (e.g. improving the process's performance, detecting the bottle-necks in the models).

Log Replay is an analysis technique invented in Process mining area, and mainly focuses on Extension and Conformance.

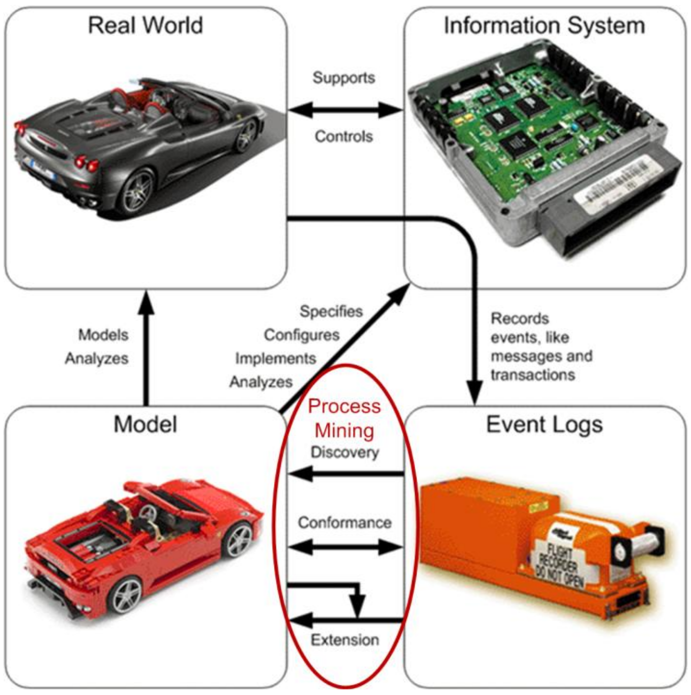


Figure 2.7: Process mining role

One of mostly used framework to support process mining is the process mining framework ProM [8]. ProM is plug-in-based software that allows researchers to apply a wide variety of process mining techniques in an extensive environment. ProM provides more than 250 plug-ins that support almost kinds of process mining purposes, for instance, process discovery, conformance checking, performance checking, process conversion and other research areas. ProM also provides an advanced visualization and verification capability. In most of the cases, the initial input of ProM is an event log. Currently, ProM accepts event logs in MXML format [9] and XES format [10]. More information about process mining in general, ProM framework and the most recent researches concerned in this area can be found in [11].

Chapter 3

Related Works

In this chapter, we introduce the notion of log replay technique. Additionally, we explain how the replay technique is implemented on specific process modeling languages.

3.1 Log replay

Log replay is an analysis technique allowing to simulate event traces in an event log comparing to a given process model. Log replay plays an important role in process mining. It can be used for many analysis purposes within process mining area, for instance, detecting the fitness, behavior, appropriateness between a process model and an event log or indentifying performance of a process model. Process models are described in form of many process modeling languages. However, log replay currently exists only on three modeling languages, namely Petri net, Flexible model and Fuzzy model. In the following sections, we explain the log replay techniques of those process models.

3.2 Petri net replay technique

3.2.1 Log replay on Petri net

The Petri net replay method was first introduced in [12]. Input for the replay is a Petri net and an event log .Before the replay can start, events in log should be mapped to transitions in Petri net such that each event is mapped to a transition that corresponds to the activity represented by this event. Since duplicate transitions are allowed in Petri net, an event might be associated with more than one transition in Petri net. Transitions which are not related to any activities are regarded as *invisible* transitions used only for routing purpose. The log replay will simulate every event trace in the log. Replaying a trace starts with the setting of initial marking for Petri net. Events from each case will be replayed one by one in a chronological order. The present of tokens play an important role because they specify execution behavior of transitions in Petri net. They indicate whether or not a transition is executed with proper pre- condition.

Once an event is replayed, one of its related transitions [2] is fired. When a transition is fired, a token is removed from each of its input places and a token is produced in each of its output places. Firing of a transition can be proper or improper depending on whether or not the transition is enabled in advance. Generally, a transition is enabled whenever there is at least one token in each of its input places. However, a Petri net may contain sequences of currently enabled invisible transitions preceding the related transition. Executing these sequences will enable the execution of the related transition. Thus, within the context of Petri net replay, a transition is said to be enabled if it is

[2] Related transition: a transition is mapped from an event and corresponds to the activity represented by this event.

enabled either by current marking or by preceding invisible transitions. In case that related task cannot be enabled directly by current marking, it should be checked whether it can be enabled by sequences of invisible tasks before considering it as a failed task (i.e. a task is executed improperly).

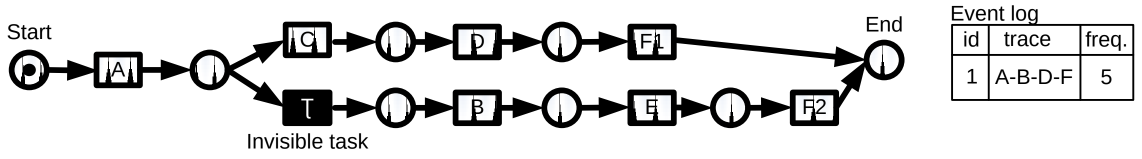
Whenever an event is replayed, elements from set of related transitions will be executed according to heuristic rules defined in [12]. First, we denote the set of related transitions of the currently replayed event as T. The following possibilities are considered based on heuristic.

- If all elements from T are not enabled by all means, an arbitrary transition will be fired by adding artificial tokens into its input places. We refer these tokens as *missing tokens*. Hence the transition is executed unsuccessfully, i.e. the current event is replayed improperly.
- If T contains only one transition that is enabled either by current marking or by sequences of preceding invisible transitions or both, eventually it will be executed successfully. In the first case, the transition will be fired directly if it is enabled by current marking only. In the second case in which the related transition is enabled only by multiple sequences of invisible transitions, heuristic chooses the shortest sequence to enable the related transition. If transition is enabled by both current marking and sequences of invisible tasks, it will be executed immediately rather than executing invisible transitions.
- If set T contains more than one enabled transition, the most suitable transition will be chosen based on the method already presented in [12, p.90-94].

Furthermore, Petri net replay technique may create a number *missing tokens* and *remaining tokens* after all events in a case are replayed. Missing tokens indicate number of tokens added artificially to execute transitions without proper pre-condition. Remaining tokens are the tokens not consumed completely and remain in process model after log relay is finished. The existence of missing tokens and remaining tokens are important to calculate Petri net conformance fitness metric as defined in [12, p.70].

3.2.2 Example

To illustrate the Petri net replay technique, we present an example in which an event log is replayed on a Petri net as shown in Figure 3.1. The event log in this example contains only one event trace *A-B-D-F* with frequency value of 5. The Petri net contains an invisible task and two duplicate tasks, *F1* and *F2*. They both correspond to the same event *F* in the event trace. The replaying steps of trace *A-B-D-F* are shown in Figure 3.2.



Event log

id	trace	freq.
1	A-B-D-F	5

Figure 3.1: Petri net replay example

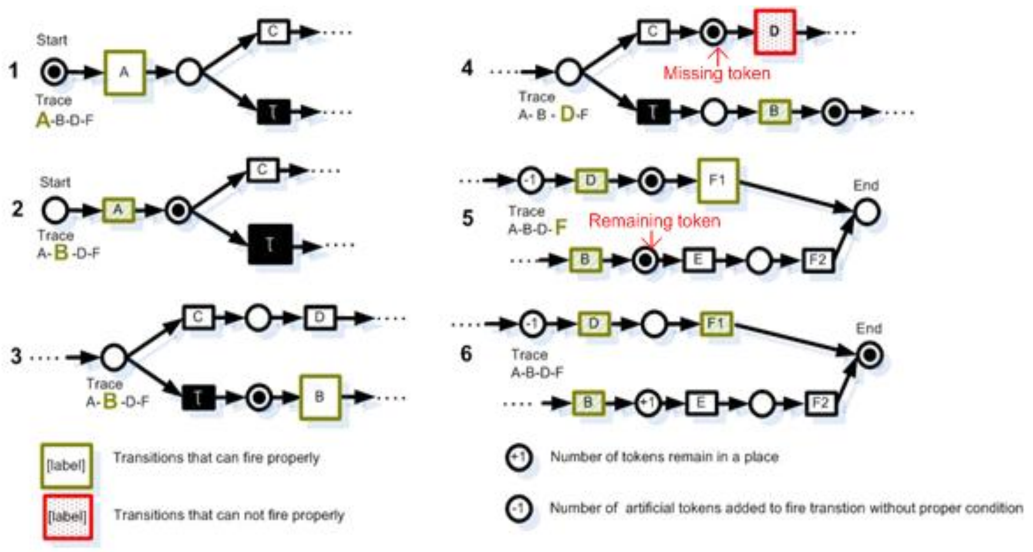

Start
1
A
C
T
Trace
A-B-D-F
2
4
Missing token
D
5
Remaining token
End
F1
F2
B
E
6
Transitions that can fire properly
Transitions that can not fire properly
Number of tokens remain in a place
Number of artificial tokens added to fire transtion without proper condition

Figure 3.2: Replaying steps of trace A-B-D-F on the model given in Figure 3.1

We observe that after event *B* is replayed, heuristic fires task *D* incorrectly and hence task *F1* is executed next instead of task *F2*. This produces five missing tokens in input place of task *D* and five remaining tokens in output place of task *B* after the log replay is finished. The projected replay result is shown in Figure3.3.

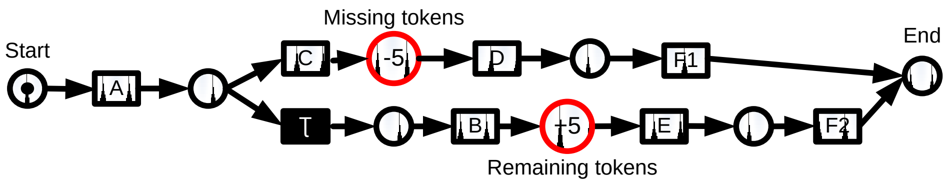


Figure 3.3: Result of replaying the event log in Figure 3.1

3.2.3 Limitation

Problems might occur when Petri net contains either invisible or duplicate tasks. To deal with such invisible and duplicate tasks, local heuristics are used to find the optimal decision for replaying events, for instance, the shortest sequence of invisible transition is always selected to enable execution of a transition that corresponds to the currently considered event and currently enabled transitions must be chosen over un-enabled transitions from list of duplicate transitions. Since the design decision is optimized locally, from the global point of view, there is no guarantee if the best optimal replay result is always retrieved. There could be a case that firing transitions violating heuristic rules (e.g. executing the longer sequence of invisible transitions or firing un-enabled transitions rather than firing enabled transitions) would produce an executing sequence of tasks that is compliant better to the observed behavior.

As an example, we replay the event trace *A-B-C* on the Petri net as shown in Figure 3.4. This Petri net contains two duplicate tasks *B1* and *B2* both corresponding to the same event *B*. Using classical approach to replay this event trace, there is one transition *(C)* that fires without proper pre-condition. This is because the replay algorithm chooses to execute *B1* rather than *B2* after the execution of *A*. When event *C* is replayed, an extra token must be added to execute *C* without proper pre-condition. The existence of missing token leads to a conformance fitness metric less than

1 in this example. Therefore, executing sequence *A-*τ *-B2-C* is supposed to be selected as it produces no missing tokens. However, this sequence is never executed due to local heuristic rules.

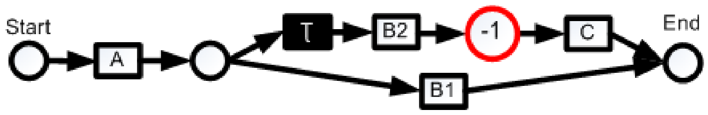


Figure 3.4: A Petri net with two duplicate tasks to be replayed with trace A-B-C

There is another serious problem concerning heuristic decision of invisible tasks which relate to OR-Split construct. In Petri net, an OR-split construct is realized by a set of invisible transitions to model all the possible combinations. Consider an example of Petri net OR-split construct given in Figure 3.5.

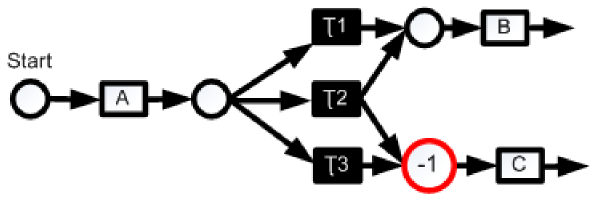


Figure 3.5: A Petri net OR-split construct to be replayed with trace A-B-C

The problem occurs when trace *A-B-C* is replayed on this Petri net. After event *A* is replayed correctly, heuristic approach is applied to select the shortest sequence of invisible transitions to enable task *B*. We realize that any of sequences *A-*$\tau 1$ and *A-*$\tau 2$ can be fired randomly since all of them have the same shortest length of invisible transitions and can both enable the transition B. However, firing $\tau 1$ would create an undesirable situation in which one missing tokens are added to execute *C* without proper pre-condition. Whereas, execution of $\tau 2$ would lead to a better optimal result since no missing tokens and remaining tokens are created. Preferably, $\tau 2$ should be executed rather than executing $\tau 1$. However, the choice between these two invisible transitions is non-deterministic by using classical Petri net replay method.

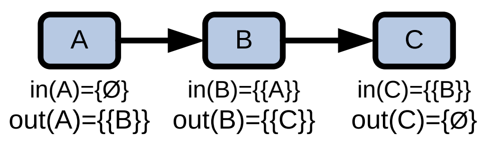
Therefore, the optimal replay result cannot be ensured using currently applied local heuristic. To obtain the global optimal replay result, it is necessary to perform further analysis for each design decision choice during the replay. However, the current Petri net replay technique is not invented to support that.

3.3 Flexible model replay technique

Flexible model replay technique was first presented in [13]. Before we describe it in detail, notion of unsatisfied events and A* algorithm are introduced first since they are essential in the flexible replay technique.

3.3.1 Unsatisfied events

Given a Flexible model and a set of events to be replayed, an event in the set is said to be unsatisfied if it is replayed incorrectly, i.e. its corresponding task in the flexible model fires without proper condition. Consider an example in which a trace *A-C* is replayed on the flexible model shown in Figure 3.6. The enabling of task *C* requires an execution of *B* in advance. However, no events associated with task *B* appear in front of event *C* in the trace. Thus, during replaying trace *A-C*, task *B* never be executed and hence task *C* cannot be enabled. To replay event *C*, the related task *C* has to be fired without prior execution of task *B*. Therefore, we regard event *C* as an unsatisfied event. The existence of unsatisfied during replay significantly affects the *conformance task ratio fitness metric* for flexible model [7, p.8]. Note that the definition of unsatisfied events is similar to events that cause missing tokens in Petri net replay.



Event log

id	trace	freq.
1	A-C	2

Figure 3.6: Trace A-C is replayed on the given Flexible model with C is an unsatisfied event

3.3.2 A * algorithm

A* algorithm [14] is a searching algorithm based on heuristics to find a least-cost path from a source node to a set of target nodes in a weighted directed graph. The algorithm uses cost function f to determine the order in which the search visit the nodes in the graph. Let *n* be a node in the graph, function $f(n)$ is defined as the sum of $g(n)$and $h(n)$, where $g(n)$ is the least cost to go from the source node to node *n*, $h(n)$ is an admissible heuristic function returning the estimated cost to go from node n to the closet target node. In order to find the optimal path from source to target, function $h(n)$ should not overestimate the actual cost to reach the closet target. The basic idea of A* algorithm is that from the node currently visited, one of its successors *n* with the least $f(n)$ is visited next. The algorithm stops whenever a target node *y* is visited (i.e. $h(y) = 0$) or no more nodes to be explored.

3.3.3 Log replay on Flexible model

The flexible model replay technique takes a flexible model and an event log as inputs. Before the replay can start, each event from the log must be mapped to a corresponding task in flexible model. The flexible replay technique will simulates each event trace from the log one after one. For each event trace, the replay algorithm creates a search tree containing an optimal path that reflects the replayed events in the trace. The flexible model replay algorithm works by iteratively exploring successors of nodes, starting from the source node. Each time of iteration, we keep track of paths from initial node to each successor. To determine which successor is going to be explored in next iteration, the algorithm calculates the cost $f(n)$ of every successor and then selects the one whose $f(n)$ is the smallest.

The log replay starts by creating an initial node first. After that the successors are generated based on the following "***move***" types.

- ***Move on model only with invisible task***: If there are some currently enabled invisible tasks in the model, then this action will execute each of these invisible tasks without replaying any events from the log. Every *move on model only with invisible task* will create a new node instance in search tree. The purpose of *move on model only with invisible task* is to identify the events that cannot be observed in reality. Executing these invisible tasks might enable the task to which the currently considered event in the trace corresponds.

- ***Move on model only with real task***: If there are some currently enabled real tasks in the model, then this action will execute each of these real tasks without replaying any events from log. Every *move on model only with real task* will create a new node instance in search tree. The purpose of *move on model only with real task* is to identify the events that actually happen in reality but not logged. Executing these real tasks might enable the task to which the currently considered event in the trace corresponds.

- ***Move on log only:*** The currently considered event is removed from the log without firing any tasks from the model. Some activities are logged more than one in event log. Therefore this move will indentify the redundant events in the log. Each move on log only will create a node instance in search tree.

- ***Move on both log and model:*** The currently considered event is replayed and its related task is fired. If the current event is related to more than one task in the model. The *move on both log and model* will happen for each of the related tasks of the current event. The replayed event can be unsatisfied if the related task is fired improperly otherwise it is satisfied. Every *move on both log and model* will also create a node instance in search tree.

The algorithm will select a successor node whose cost value $f(n)$ is the smallest for the next iteration. In the next iteration, the same procedure is applied. The log replay stops whenever the target node is reached. The target node is usually obtained either by removing the last event from the log, by replaying the last event correctly or by replaying the last event incorrectly. At the end, we obtain an optimal path in which the number of unsatisfied event is the smallest. Each edge from this path corresponds to a "move" type.

Before we describe the cost function $f(n)$ for each currently visited node *n*, a list of cost parameters involved in $f(n)$ is introduced first in Table 3.1:

Parameter	Description
RE	Cost of replaying an event successfully.
E	Cost of a single event that still left in the case
Us	Cost of replaying an unsatisfied event.
UnA	Cost of an unhandled arc, i.e. arc connects to a task that is enabled after executing an event.
Minv	Cost of executing an invisible task
Mr	Cost of executing a real task only without replaying the current selected event in the log
Rem	Cost of removing an event from the log

Table 3.1: Description of parameters required for Flexible model -based replay

Then $f(n)$ is defined as the cost of the path running from source node to target node and passes through current node n.

$f(n)$ is formulated as:

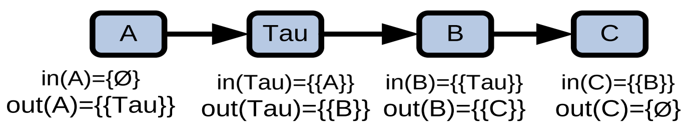
- $f(n) = g(n) + h(n)$ **(1)**
- $h(n)$ is the estimate cost to go from current node n to the target node. Let *L* be the number of events left in the event case at node n then $h(n) = L * E$.
- $g(n)$ is the least cost to reach the current node n from the initial node. Under node n, let *A* be the number of events replayed successfully so far. Let *B* be the number of created un-handled arcs. Let *C* be the number of executed invisible tasks. Let *D* be the number of visible tasks executed without replaying any events from the case. Let *F* be the number of events removed from the case so far and let G be the number of unsatisfied events replayed. Then $g(n)$= *A*RE + B*UnA + C*Minv + D*Mr + F*Rem+G*Us.* Note that number of events replayed successfully is increased by *move on both log and model.* Similarly, number of unsatisfied event is increased by *move on both log and model.* The number of executed invisible tasks is increased by *move on model only with invisible task.* The number of executed visible tasks is increased by the *move on model only with real task*. And the number of removed events is increased by *move on log only*.

- The cost value $f(s)$ of the source node *s* is always equal to $h(s)$ (i.e. $g(s)$=0). And the cost value $f(t)$ of the target node *t* is always equal to $g(t)$ (i.e. $h(t) = 0$).

Before the replay can start, cost parameters listed in Table 3.1 must be provided with certain cost values. For better understanding of flexible model replay method, an example is demonstrated in the subsequent section.

3.3.4 Example

Given a Flexible model, an event log and values required for cost parameters as shown in Figure 3.7, a search tree is constructed during replaying trace *A-B-C* as shown in Figure 3.8.



Event log

id	trace	Freq.
1	A-B-C	2

Parameter setting

Parameter	Value
RE	1000
E	1000
UnA	1
Minv	1
Mr	2000
Rem	3000
Us	1500

Figure 3.7: Trace A-B-C is replayed on the flexible model with pre-defined parameters

After initial node S is created, *move on model only* can happen on *A* since *A* is already enabled. This creates the node instance 3. *Move on log only* removes event *A* and creates instance 1. *Move on both log and model* replays event *A* and fires related task *A* simultaneously and results in instance 2.

A is said to be satisfied because related task *A* has been fired properly. Replay algorithm then calculates cost value $f(n)$ of each created instance using formula **(1)** described in Section 3.3.3. Instance 2 with smallest cost value is then selected for next consideration.

We then apply the same procedure to find successors of instance 2. Event *B* in the log is going be replayed. *Move on log only* removes event *B* and results in instance 2.1. Observed that execution of *A* enables the invisible task Tau, moreover task *A* is always enabled at anytime. Thus *move on model* now can happen both on invisible task Tau and again on task *A*. *Move on both log and model* will replay event *B* and execute related task *B*. Event *B* is regarded as unsatisfied since task *B* has been fired improperly (i.e. missing execution of *Tau* in prior). We observe that *move on model* on *Tau* creates an instance (instance 2.4) whose cost value $f(n)$ is the smallest, thus instance 2.4 is selected for next iteration.

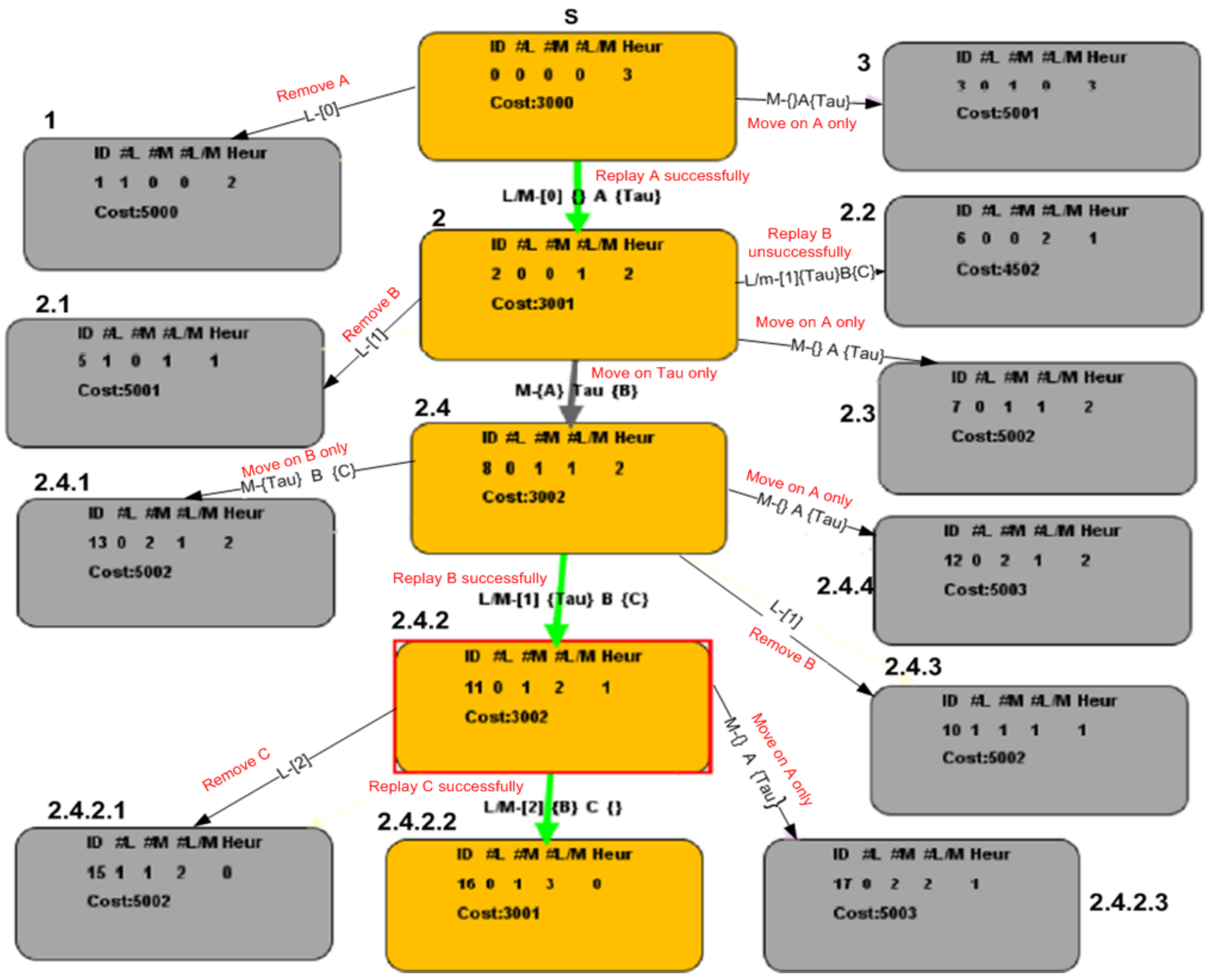


Figure 3.8: Search tree generated after replaying A-B-C contains an optimal path

The algorithm continues the same procedure until last event from the case is considered. At the end, we obtained an optimal path which reflects the replay of each event from trace *A-B-C*. We observe that the first edge from this path corresponds to replaying event A correctly. The second edge corresponds to executing an invisible task. The third edge corresponds to replaying event *B* correctly and the last edge corresponds to replaying event C correctly. In this path, all replayed events are satisfied.

The initial node instance is always selected at the beginning of search tree. Apparently, the cost value $f(n)$ of initial node is always equal to its heuristic cost value $f(n)$. In our example the cost of source node S is $f(S) = h(S) =$ *number of remaining event in the log* * E=3*1000=3000. At every currently explored node instance, the algorithm will try to implement all kinds of **"move"** described above to create the successors. Note that *move on log only* happens at most one time as we can only remove one event at a time. Number of occurrences of *move on model only* depends on the number of currently enabled tasks, while *move on both log and model* depends on the number of duplicate tasks related to currently considered event. The successor node with smallest cost value $f(n)$ is always selected for next iteration. We continue the same procedure for next selected instance until the last event is replayed.

3.3.5 Limitation

In contrast to Petri net replay in which the design decision is made based on local optimization, flexible model replay will form different potential scenarios for each event to be replayed and performs further analysis over those potential scenarios. Thus, global optimal scenario is always guaranteed to be found.

However, there is a serious problem existing in flexible model replay method. This problem concerns the fact that number of investigated instances during replay might be increased significantly. The current flexible model does not have notion of starting tasks. Therefore, all flexible tasks which don't have predecessors will be treated as initial tasks. Initial tasks have empty input set[3] and they are always enabled at any time. Since they are always enabled, *move on model only* with every initial task will always be taken at every currently explored node instance. For instances, in our example, *move on model only* with *A* happens at every selected instance along the optimal path. This creates a large number of new instances caused by moving only on *A*. Additionally, when duplicate tasks are considered; each one of them forms a different scenario that requires further analysis to choose the optimal one. Obviously, this also creates a lot of new instances. Thus, for the replay between a model containing huge number of initial tasks, huge number of duplicate tasks and a long log trace, the algorithm will explore a large number of instances. Hence it decreases the performance of log replay.

There is another problem concerning unhandled arc cost. First of all, we introduce the purpose of unhandled arc cost. The idea of unhandled arc cost is to avoid a special situation described as follows. Given a flexible model in Figure 3.9, both sequences *A*- $\tau 1$-*C-D* and *A*- $\tau 2$ -*C-D* are executed correctly during replaying trace *A-C-D*. However, execution of $\tau 1$ also enables task B. Thus when trace *A-C-D* finishes replay, *B* remains enabled, i.e. this leads to an improper completion. It is similar to the existence of *remaining tokens* in Petri net. Hence $\tau 2$ should be executed rather than $\tau 1$. To enforce $\tau 2$ to be selected, we should add unhandled arc cost to the calculation of evaluation

[3] For a certain node s, if its input set is *in(s) = {∅} (Empty input set)* then s is always enabled. If input set is *in(s) =∅(input set is null),* then it is never enabled.

cost $f(n)$. The total cost $f(n)$ to execute $\tau 2$ is then smaller than the total cost to execute $\tau 1$ because $\tau 1$ creates 3 unhandled arcs whereas $\tau 2$ creates only 2. Consequently, $\tau 2$ is selected.

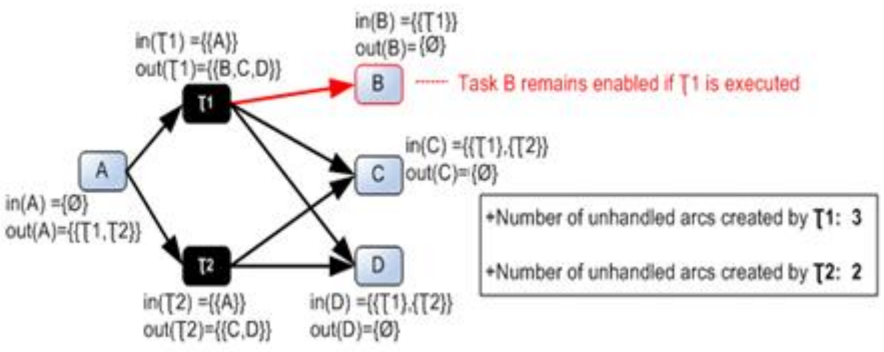


Figure 3.9: An example of flexible model which proves the need of unhandled arc cost

However, the existence of unhandled arc cost would produce another serious problem. As an example, consider the flexible model given in Figure 3.10 and an event trace *A-B-C*. Both tasks *B1* and *B2* in the model represent the same event *B*. The expected executing sequence for replaying trace *A-B-C* should be *A-B1-C*. However, the actual obtained sequence is the trace *A-B2-C* in which *C* has been executed unsuccessfully, i.e. event *C* is unsatisfied. This is because the execution of B1 creates a massive number of unhandled arcs which is larger than the unsatisfied event cost ***Us***. Thus cost to execute the trace *A-B1-C* is bigger than cost to execute the trace *A-B2-C*. Hence the algorithm chose to execute *A-B2-C* which is unexpected.

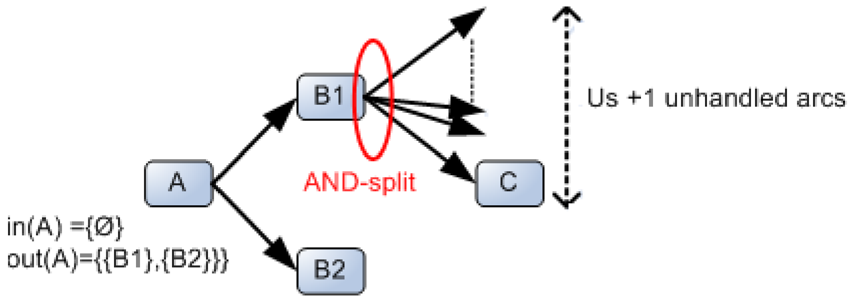


Figure 3.10: An example of flexible model which shows the problem of using unhandled arc cost

3.4 Fuzzy model replay technique

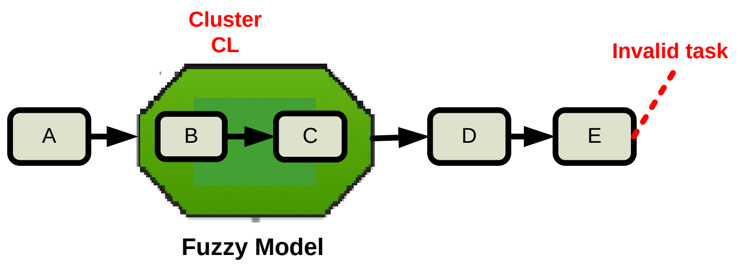
3.4.1 Log replay on Fuzzy model

The method for log replay on fuzzy model was first introduced in [5, p.199-203]. Inputs for fuzzy model replay technique are an event log and a fuzzy model. Since fuzzy model supports activity abstraction, activities whose behavior is not of interest will be discarded from model. Therefore, events in the log whose activity abstracted are considered as unmappable events. The log replay starts with a pre-processing phase in which unmappable events are removed from the log. Every unmappable event is counted as a *deviation*. The term *deviation* refers either to an event whose activity is abstracted or to an event whose corresponding task in fuzzy model is executed without proper pre-condition. The number of deviations between a model and an event log are important to calculate the fuzzy conformance metric [5, p.199].

The basic idea of fuzzy model replay technique is to subsequently transform sequence of events to be replayed into an executing sequence of corresponding tasks in fuzzy model. Events whose related tasks are contained in cluster nodes will be mapped to their corresponding cluster nodes. The replay algorithm then executes each node from the executing sequence one by one. Each time a node (primitive node or cluster node) is executed, it will be classified as valid or invalid. If the current node has been enabled by previous executed nodes or it is the same cluster node observed directly before, it is said to be valid. Otherwise, it is said to be invalid. All invalid nodes are regarded as *deviations*. The detailed explanation of how to detect the validity of executed nodes is presented clearly in [5, p.199-203]. In the next subsequent section, we illustrate a simple example of fuzzy model replay technique.

3.4.2 Example

Consider an example in which we replay the trace *A-B-C-E* on a fuzzy model shown in Figure 3.11. Since task *B* and *C* are grouped in cluster node *CL*. Trace *A-B-C-E* will be transformed to the executing sequence *A-CL-CL-E*. The replay begins with execution of task *A*. The execution of task *A* is valid as task A is enabled at any time during a process execution. After *A* has been executed, cluster node *CL* from model is enabled. Thus this makes the execution of first cluster node *CL* from the sequence valid. The execution of second cluster node *CL* is valid too because it is exactly the cluster *CL* observed directly before. Execution of *CL* then enables *D* only. However, the next node to be executed is node *E*. In this case, the execution of *E* is invalid since there is no execution of D before to enable task E. Therefore the replay of trace *A-B-C-E* results in one *deviation* (i.e. E)



.Trace to be replayed: **A-B-C-E**
.Mapped sequence of related tasks: **A-CL-CL-E**

Figure 3.11: Trace A-B-C-E is replayed on the given Fuzzy model

3.4.3 Limitation

There are several problems existing in the current fuzzy model. One of problems is that fuzzy model replay is able to replay an invalid event trace correctly due to its relaxed execution semantics. As an example, consider an example in which we replay the trace *A-B-C-D* in a Fuzzy model which describes the same behavior as the one described in the Petri net depicted in Figure 3.12.

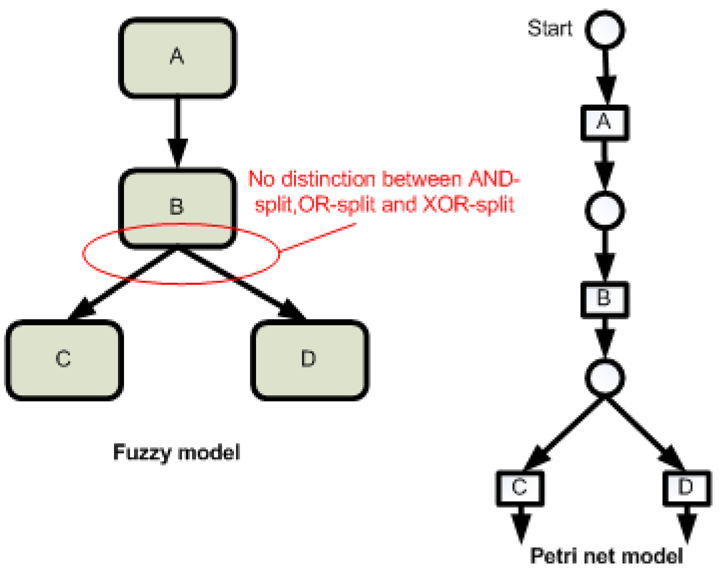


Figure 3.12: Fuzzy model converted from Petri net

Assume that this Petri net conform 100% to reality. Replaying the trace *A-B-C-D* in the Petri net creates a fitness value less than 100%. Thus, this trace is regarded as invalid trace. However, due to relaxed execution semantics of fuzzy model, this trace does not deviate from the fuzzy model during the replay.

There is another problem concerning invisible tasks. Conceptually, Fuzzy model does not have notion of invisible tasks. This is because fuzzy model can only be obtained by directly discovering from event log. Obviously, invisible tasks are not recorded in event log. Therefore, the current fuzzy replay technique was not designed to deal with invisible tasks. If any invisible tasks from any other process models can be mapped to Fuzzy nodes, they will be regarded as real tasks. This brings up the following problem. Consider a fuzzy model depicted in Figure 3.13 in which the trace *A-B* is replayed. The fuzzy model contains a task Tau that is indeed an invisible task. However, task Tau is treated as a real task in Fuzzy model. Hence during replaying trace *A-B*, the corresponding sequence *A-B* from the given Fuzzy model is executed with one deviation, i.e. execution of B is invalid. This leads to a fuzzy conformance metric less than 1 which is supposed to be 1.

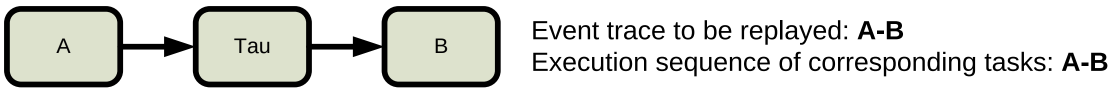

A
Tau
B
Event trace to be replayed: **A-B**
Execution sequence of corresponding tasks: **A-B**

Figure 3.13: A Fuzzy model which contains an invisible task Tau

Chapter 4

Generalizing process modeling languages

The broad application of business process models has stimulated organizations to create dozens of process models based on different types of modeling languages. Apparently, an issue comes up on how to deal with such large volumes of models, particularly when one needs to consults, customize and re-use models. Over the years, there are a lot of techniques developed for process model analysis. However, most of them look at process models in isolation, i.e. they work only on a specific model rather than being applicable to a wide variety of process modeling languages. This problem is including the log replay technique which has been invented for specific process modeling languages. It is always desirable to perform replay on any types of modeling languages for process analysis purpose. Thus, in this chapter, we present an approach for replaying an event log on various process modeling languages. Also in this chapter, we introduce two generic process formats that are essential in our replay approach.

4.1 Replay approach on variety of modeling languages.

Log replay allows for re-playing observed events against a given process model. Currently, log replay exists only on some specific modeling languages, such as Petri net, fuzzy model and flexible model. It could always happen when one takes any existing process models (e.g. EPC, Yawl) and wishes to perform the log replay on them. This may require a new log replay method developed for such process models. However, creating a new log replay is difficult and takes a lot of efforts. Therefore, rather than developing a new replay technique for each process modeling language, we take the idea of unified analysis approach which is similar to the one presented in [12, p.199]. The idea of this approach is to develop a replay method on a generic process format that provides abstraction of existing process models. This unification approach is described in Figure 4.1.

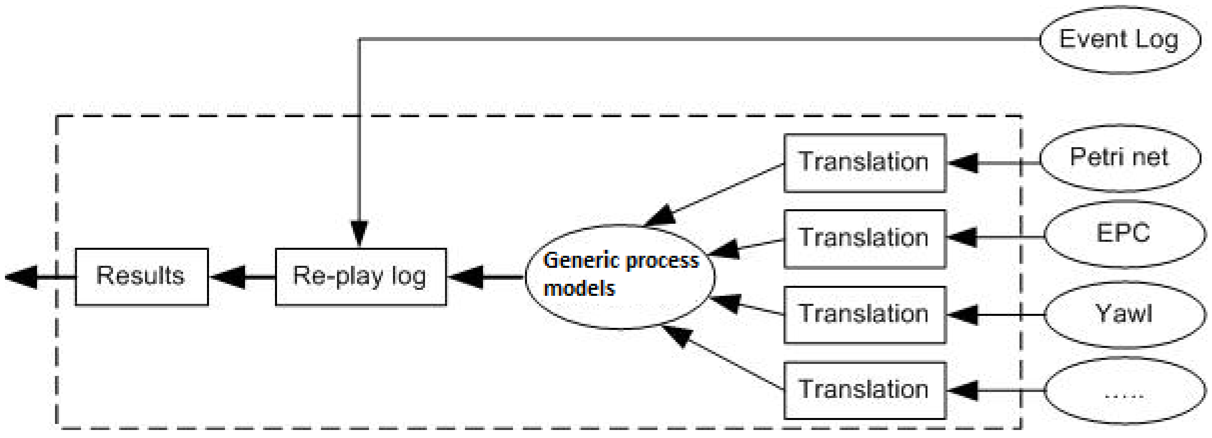


Figure 4.1: A unification approach for Log Replay

This approach allows any existing process models to be replayed indirectly as long as they can be expressed in the form of a generic process model. Generic process model is essential to enable log replay on a wide variety of modeling languages. As the idea is to perform log replay independently from specific semantics, generic process model is required to leave away specific semantics and to capture the core split/join behaviors of existing process modeling languages. Thus, flexible model and canonical process format introduced in Chapter 2 appear to be the two potential candidates for our unification approach.

It is important to realize that flexible model and CPF model can be obtained using several approaches, e.g. by discovering directly from event log, by modeling them manually or by converting existing process models. In current practice, Flexible model can only be obtained by modeling it manually, while CPF model can be obtained from other process models using various conversional methods [6]. With the lack of conversion methods available for flexible model, CPF seems to be a better choice. Moreover, CPF is optimized to supports a wide range of analysis purposes [6, p.2-4] such as quality analysis, correctness analysis, performance analysis, pattern-based analysis or similarity search. Therefore, we choose CPF as a process modeling language to generalize existing process models.

4.2 Conversion aspect

CPF models are obtained by mapping elements of other process models to elements of CPF model. Currently there are conversion methods available for six different modeling languages, namely EPC, BPMN, Protos, Yawl, WF-net and BPEL, as introduced in [6, p.7-10]. Apparently, mapping rules described in [6] are not complete. They do not explain in detail how specific constructs of existing process models can be mapped to CPF model. Therefore, in this section we explain mapping rules in more detail and improve them as necessary to tackle several specific cases. However, we do not describe mapping rules of all six modeling languages mentioned above. Instead, we choose to explain the ones of Yawl and EPC as Yawl can capture almost workflow patterns [1] and contains some advanced constructs (e.g. cancelation set, multiple-instances tasks, composite tasks) while EPC is the mostly used modeling languages in industry. Additionally, we design the mapping rules for Petri net and Flexible model.

4.2.1 Petri net to CPF

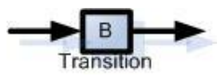
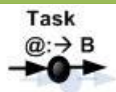
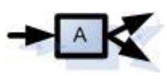
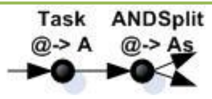
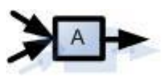
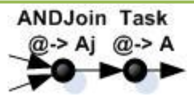
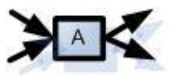
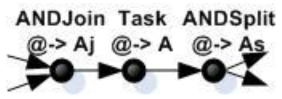
A typical Petri net graph consists of arcs, transitions and places. Places can be specialized into input places, output places and normal places. Input places and output places are used to signal the beginning and the end of a process, respectively. Normal place captures a state before an event-decision is made or pre/post-condition of a transition. Furthermore, places can be partitioned into either sese[4]- places or non-sese places. Aside from representing activities, transitions can be used to route the control flow in process execution. Hence, we classify transitions into *sese-transitions*, *AND-join transitions*, *AND-split transitions* and *AND-join AND-split transitions*. While sese- transitions are only used to model activities, the others can also be used to express the split/join behaviors, such as AND-join transitions indicate the synchronization, AND-split transitions indicate parallel branching and AND-join AND-split transitions indicate both synchronization and branching.

[4] sese: single entry single exist referring to a model element with at most one incoming arc and one outgoing arc

Table B.1 in Appendix B illustrates how Petri net elements are mapped to canonical elements. It is observed that all of the Petri net elements can be converted to canonical elements. For instance Petri net arcs are simply mapped to canonical edges. Sese-places are mapped to canonical events whereas non-sese places are mapped to canonical states. In Petri, AND-join/split routing and task behavior are semantically bound to a single transition node. However, in canonical representation, we need to separate the task from its parallel routing behavior. Therefore, Petri net transitions are mapped to canonical tasks while transition's parallel split and synchronization must be mapped to canonical ANDSplit node and canonical ANDJoin node, respectively.

In table 4.1, we show the conversion from Petri net to CPF in more detail by illustrating canonical representation of ten Petri net constructs. The first construct represents mapping of a Petri net sese-transition. In this case, we simply map it to a canonical task. The second construct illustrates mapping of an AND-split transition. An AND-split transition is mapped to a canonical task followed by a canonical ANDSplit node. Similarly, an AND-join transition in the third construct is mapped to a canonical ANDJoin node followed by a task. The fourth construct represents mapping of a special type of transition which involves both synchronization and parallel branching. In this situation, the transition must be mapped separately to an ANDJoin node followed by a task which is then followed by an ANDSplit node.

A sese-place which indicates something happening during the process execution is mapped to a canonical event as shown in the fifth construct. However, non-sese place which captures simple choice, simple merge or both must be mapped to a canonical state. A mapping of a non-sese place is exampled in the sixth construct. Sese-input place and sese-output place must be mapped to a canonical event as shown in the seventh and the eighth construct. Non-sese input places (or non-sese output places) require an extra canonical state to indicate the state in which the event-driven is made or to capture the final state before the process ends. The last two constructs show examples in which a non-sese input place is mapped to an event followed by a state while a non-sese output place is mapped to a state followed by an event.

	Petri net constructs	Canonical representations
1		
2		
3		
4		

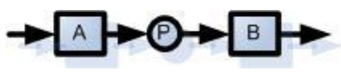
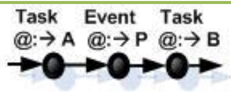
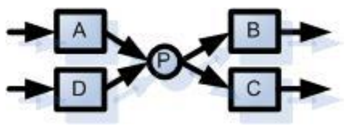
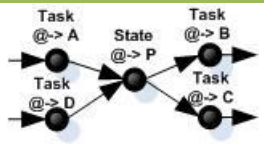
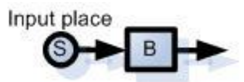
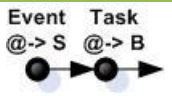
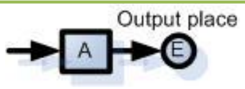
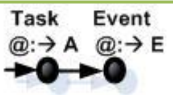
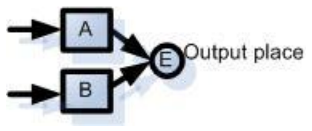
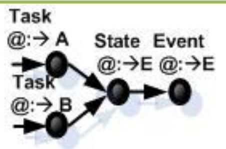
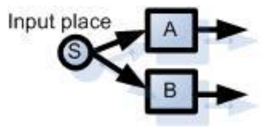
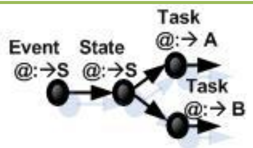
5		
6		 Task @-> A Task @-> D State @-> P Task @-> B Task @-> C
7		
8		
9		
10		

Table 4.1: Canonical representation of ten Petri net constructs

Syntactically, Petri net does not have any elements to directly support multiple merge and multiple join, whereas canonical process format do have the so-called ORSplit and ORJoin to represent such patterns. To express OR-split/join behaviors in a Petri net, a network of extra silent steps must be added (as shown in Figure 4.2.a and 4.2.b). Conceptually, multiple split and multiple join in Petri net must be replaced by a canonical ORSplit and a canonical ORJoin, respectively. However, identifying Petri net OR-split/join patterns is not an easy step in practice. For ease, we do not address such patterns specifically and perform straightforward mapping for Petri net elements as described in Table4.1. As an example, Figure 4.3 illustrates direct mappings of Petri net OR-split and OR-join into corresponding canonical notations.

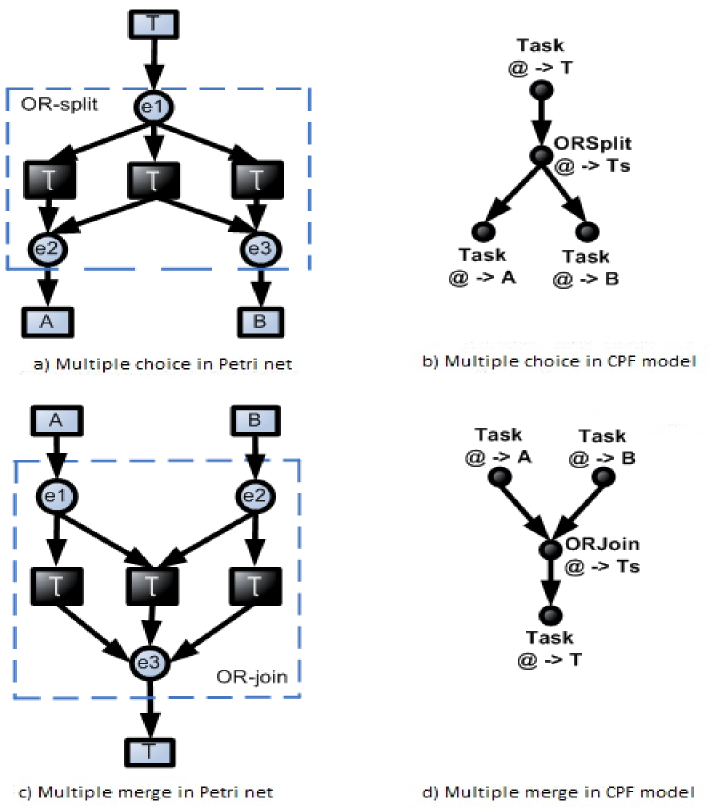

OR-split
e1
e2
e3
A
B
Task @ -> T
ORSplit @ -> Ts
Task @ -> A
Task @ -> B
a) Multiple choice in Petri net
b) Multiple choice in CPF model
OR-join
ORJoin @ -> Ts
c) Multiple merge in Petri net
d) Multiple merge in CPF model

Figure 4.2: OR-split/join modeled in Petri net compared to CPF notation

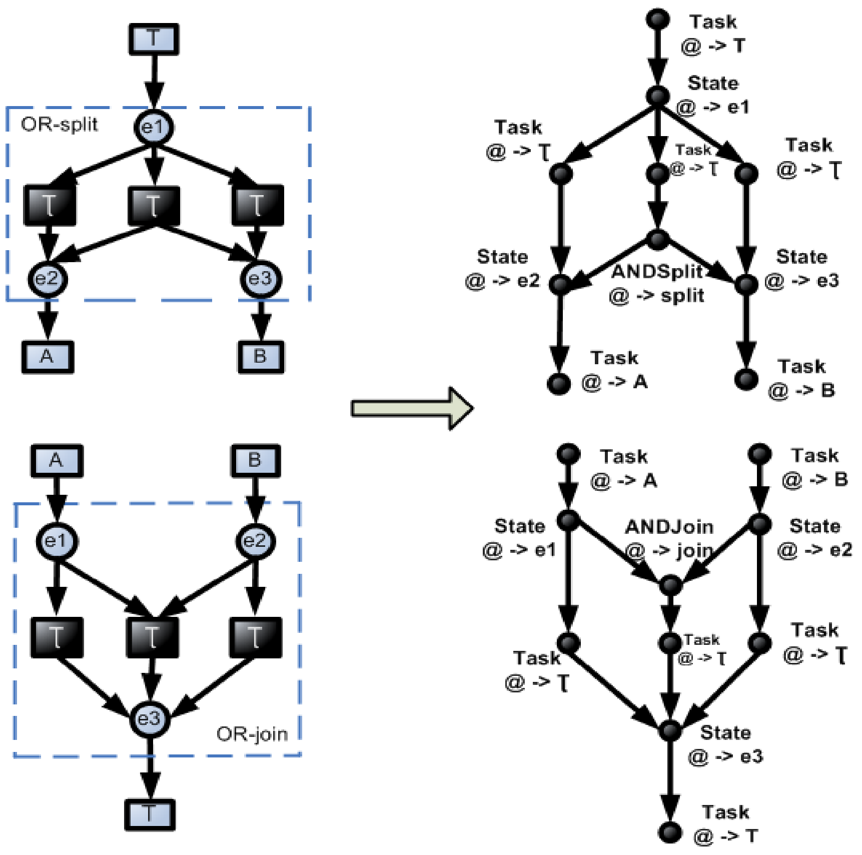


Figure 4.3: Straightforward mapping of OR-split/join from PN to CPF

Based on [1], simple choice can be classified into *exclusive choice* and *deferred choice*. The difference between these two types is concerning the moment when the routing decision is made. In exclusive choice, the choice is decided immediately by the workflow system, whereas the deferred choice delays the moment of choice as latest as possible and depends on outside environment. Figure 4.4 provides examples of exclusive choice and deferred choice in both Petri net and canonical format notation.

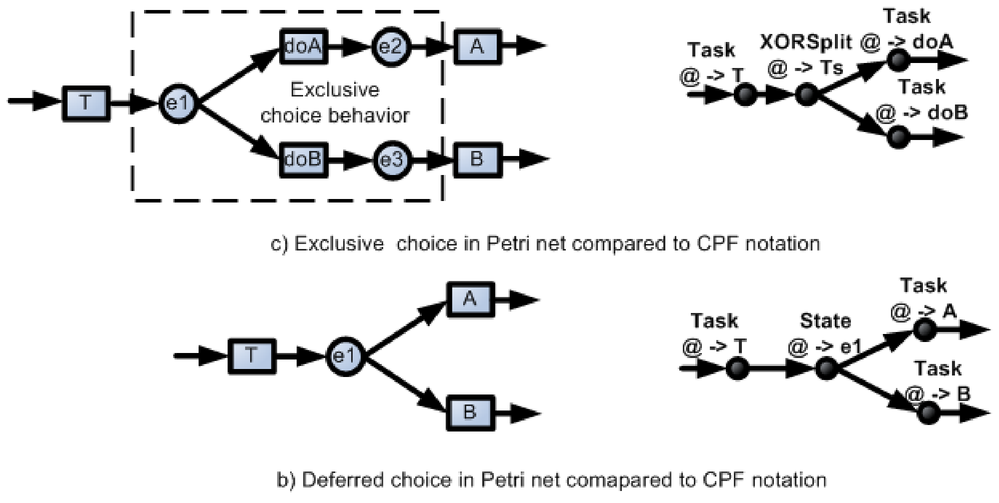

T
e1
doA
e2
A
doB
e3
B
Exclusive choice behavior
Task @ -> T
XORSplit @ -> Ts
Task @ -> doA
Task @ -> doB
c) Exclusive choice in Petri net compared to CPF notation
T
e1
A
B
Task @ -> T
State @ -> e1
Task @ -> A
Task @ -> B
b) Deferred choice in Petri net comapared to CPF notation

Figure 4.4: Simple choice modeled in Petri net compared to CPF notations

States in CPF are used to capture the deferred choice in process execution, thus mapping of Petri net's deferred choice to CPF is done directly by replacing Petri net place with a canonical state as shown in Figure 4.4.b. In general, exclusive choice pattern in Petri net should be replaced by a canonical XORSplit node as illustrated in Figure 4.4.a. However it is not easy to indentify this pattern in implementation. Without any additional information apart from the model, it is not possible to indentify transitions that only exist to help routing of a process. For instance, it is impossible to indentify the role of task *"doA"* and *"doB"* which is used to enforce the simple choice becomes exclusive choice in our example. Therefore, the mapping will be done straightforwardly by directly converting the related Petri net elements into the corresponding CPF elements as illustrated in Figure 4.5. We observe that the place *"e1"* which signals the choice in Petri net is mapped to a canonical state which is used to indicate the deferred choice in CPF model. Therefore, the exclusive choice in Petri net is converted to a deferred choice in CPF. This would be a drawback in which information about the moment of choice cannot be preserved after the mapping. Another drawback is that initial marking of Petri nets will be lost after mapping since CPF does not store information about initial marking.

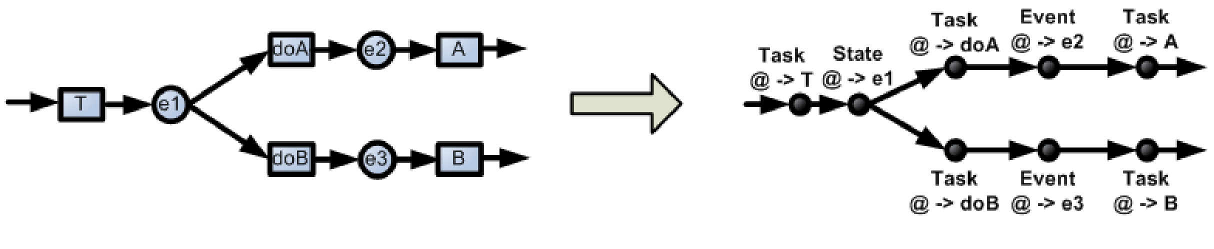


Figure 4.5: Straightforward mapping of XOR-split from PN to CPF

4.2.2 Flexible model to CPF

Flexible model is described in form of a directed graph consisting of task nodes only. Flexible tasks represent either activities or silent steps happening in a process execution. By definition, flexible model can capture almost core split/join behaviors. It is even able to capture a more relaxed synchronization/choice pattern. Consider an example in which a flexible model with more relaxed split/join constructs is shown in Figure 4.6.

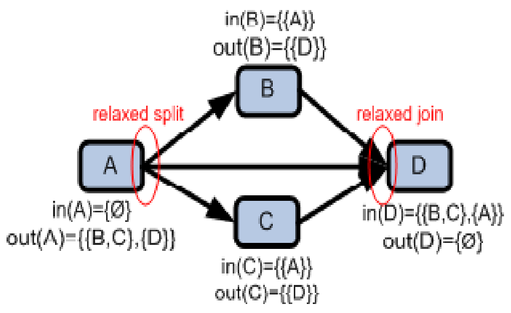


Figure 4.6: A flexible mode with relaxed split and join pattern

Observed that execution of *A* enables either *D* or both *B* and *C*. This type of split is neither multiple choice (OR-split) nor exclusive choice (XOR-split). This is considered as a more relaxed choice whose behavior is in between OR-split and XOR-split. Similarly, firing *D* is enabled by either execution of *A* only or by *B* and *C* together. Unfortunately, current CPF model was not designed to capture such kind of split/join patterns. Therefore, the mapping to CPF model assumes the original flexible models do not contain such type of relaxed split/join constructs.

Table B.2 (Appendix B) describes how flexible model elements can be mapped to canonical elements. Flexible arcs are mapped directly to canonical edges. Split/join behaviors in flexible model are bound to task nodes and specified by task' input/output set. In CPF model, they must be represented in routing nodes. Thus, flexible tasks are mapped to canonical tasks while their split/join behaviors are mapped separately into canonical split/join nodes.

In table 4.2, we explain the mapping in more detail by illustrating canonical representation of eight flexible constructs. Flexible tasks which do not contain split/join behavior are mapped directly to canonical tasks as shown in the first construct. The next three constructs represent mapping of flexible tasks which contain only a split. In this case, flexible task must be mapped to a canonical task followed by a canonical split. Type of canonical split depends on the type of split behavior bound in the flexible task. If a flexible task contains only a join, it must be mapped to a canonical join followed by a canonical task as illustrated in construct 5, 6 and 7. If a flexible task contains both split and join, it must be mapped to a canonical join followed by a canonical task which is then followed by a canonical split. The last construct shows a mapping example of a flexible task which contains both join and split behaviors (e.g. OR-join and AND-split).

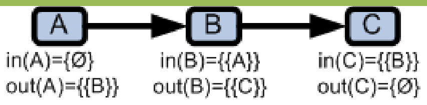
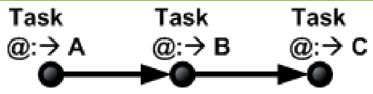
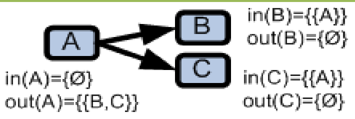
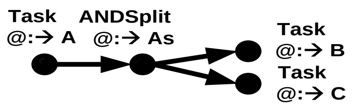
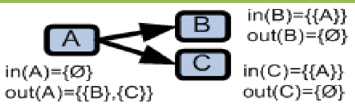
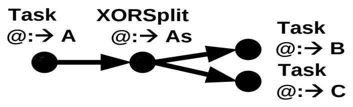
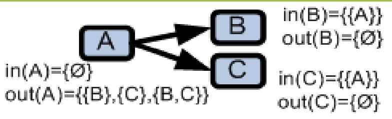
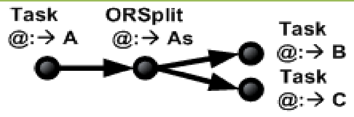
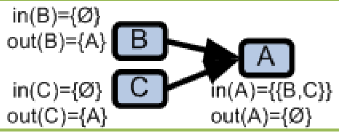
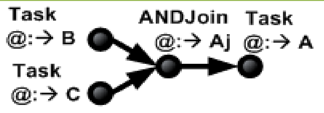
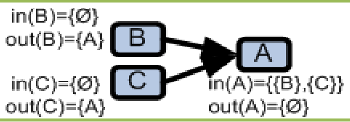
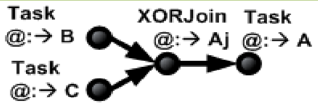
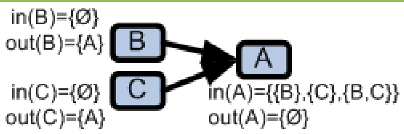
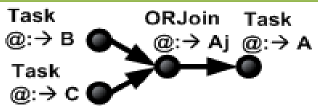
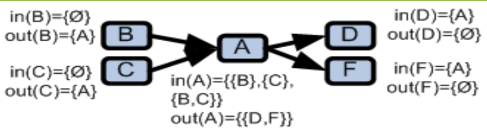
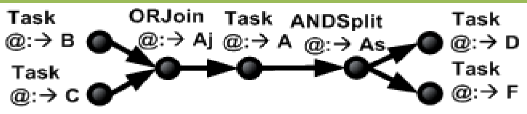
	Flexible constructs	Canonical representations
1	 A; B; C; in(A)={Ø}; out(A)={{B}}; in(B)={{A}}; out(B)={{C}}; in(C)={{B}}; out(C)={Ø}	
2	 A; B; C; in(B)={{A}}; out(B)={Ø}; in(A)={Ø}; out(A)={{B,C}}; in(C)={{A}}; out(C)={Ø}	
3	 A; B; C; in(B)={{A}}; out(B)={Ø}; in(A)={Ø}; out(A)={{B},{C}}; in(C)={{A}}; out(C)={Ø}	
4	 A; B; C; in(B)={{A}}; out(B)={Ø}; in(A)={Ø}; out(A)={{B},{C},{B,C}}; in(C)={{A}}; out(C)={Ø}	
5	 in(B)={Ø}; out(B)={A}; B; A; in(C)={Ø}; out(C)={A}; C; in(A)={{B,C}}; out(A)={Ø}	
6	 in(B)={Ø}; out(B)={A}; B; A; in(C)={Ø}; out(C)={A}; C; in(A)={{B},{C}}; out(A)={Ø}	
7	 in(B)={Ø}; out(B)={A}; B; A; in(C)={Ø}; out(C)={A}; C; in(A)={{B},{C},{B,C}}; out(A)={Ø}	
8	 in(B)={Ø}; out(B)={A}; B; in(C)={Ø}; out(C)={A}; C; A; in(A)={{B},{C}, {B,C}}; out(A)={{D,F}}; D; F; in(D)={A}; out(D)={Ø}; in(F)={A}; out(F)={Ø}	 Task @:→ B; Task @:→ C; ORJoin @:→ Aj; Task @:→ A; ANDSplit @:→ As; Task @:→ D; Task @:→ F

Table 4.2: Canonical representation of eight Flexible constructs

4.2.3 Yawl to CPF

Yet Another Workflow Languages is represented by a graph consisting of *tasks*, *conditions* and *flows*. Yawl conditions are specialized into: input conditions, output conditions and normal conditions. A condition can be either sese or non-sese. In Yawl, control flow patterns such as simple choice, simple merge, multiple choice, multiple merge, parallel branching and synchronization, are often bound together in task node. In order to perform the correct mapping, we need to distinguish different routing types inside task nodes. We categorize Yawl tasks into four main types: *Atomic Task, Multiple Instances Atomic Task (*abbreviates as *MI Atomic Task*), *Composite Task* and *Multiple Instances Composite Task (MI Composite Task)*. Each of them can associate with different routing patterns, for instance ANDJoin- atomic task is an atomic task containing only synchronization while XORJoin-ORSplit atomic task contains both simple merge and multiple split. There is a list of different types of Yawl task described in Table B.6 (Appendix B).

Table B.3 (Appendix B) provides a general view of how the Yawl's elements are mapped to canonical elements. Yawl flows are mapped directly to canonical edges. A Yawl condition can be mapped either to a canonical event if it is sese, or to a canonical state if it is non-sese. Yawl atomic tasks are mapped to canonical Tasks. A composite task must be exploited such that its sub-net should be mapped to corresponding canonical notation. This is necessary since replay technique needs to traverse all transitions during replaying. Routing behaviors inside tasks must be separated and mapped to canonical split/join nodes.

Table 4.3 produces an insight into the mapping from Yawl elements to CPF elements by illustrating the canonical representation of nineteen Yawl constructs. The first two constructs show that both atomic and multiple-instances atomic tasks are be mapped directly to canonical tasks. The reason we map multiple instances atomic tasks to canonical tasks is because CPF does not provide any elements which can syntactically represent multiple-instances characteristics in an explicit way. Therefore, information about multiple-instance characteristic in Yawl cannot be preserved when mapping to CPF.

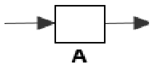
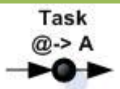
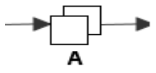
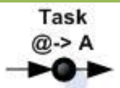
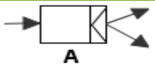
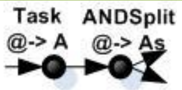
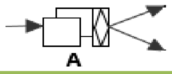
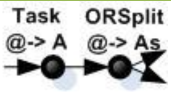
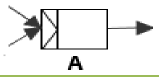
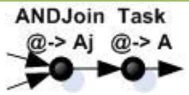
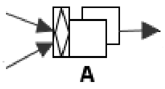
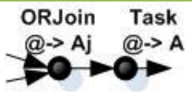
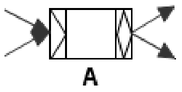
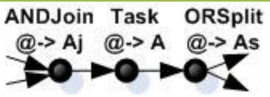
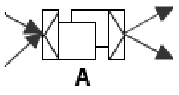
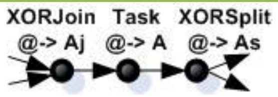
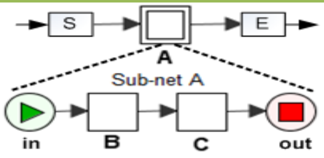
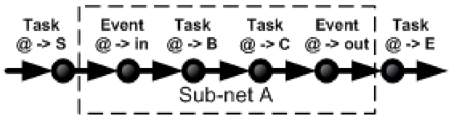
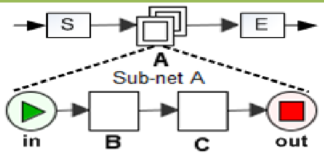
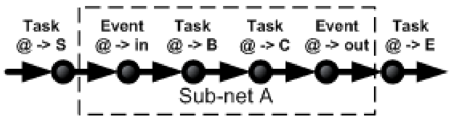
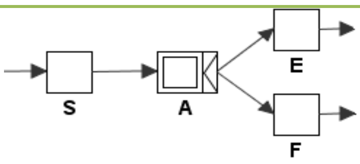
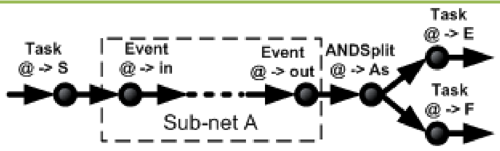
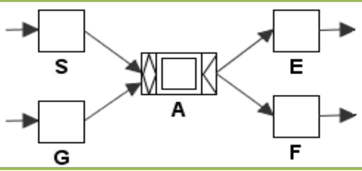
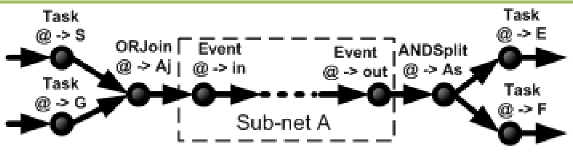
In Yawl, splits and joins must be represented inside task nodes but in canonical process format we need to separate the task behavior from its routing behavior. Therefore, a yawl task associated with a join must be mapped to a canonical join followed by a canonical task. The type of canonical join depends on the type of join behavior associated inside the original yawl task. A task associated with a split must be mapped to a canonical task followed by a canonical split. A yawl task associated with both join and split must be mapped to a canonical join followed by a task which then links to a split. In the next six constructs, we demonstrate examples of mapping (multiple-instances) atomic tasks involving join and/or split.

Since Log replay needs to replay the entire process model including sub-net models, Yawl composite tasks must be exploited completely in such a way that all elements of sub-net should be mapped to corresponding CPF nodes. If the composite tasks contain split/join behavior, these routing must be separated from task behavior and mapped to canonical split/join nodes. Similarly, multiple-instances characteristic of composite task cannot be preserved in CPF model. In the next five constructs shows examples of mapping composite tasks which possibly contain split/join behavior.

A Condition in Yawl is used to signal the start or the end of a process, or to capture something happening in the middle of process execution. If the condition is sese, we simply map it to a canonical event as shown in the construct 14, 16 and 17. If the condition is a non-sese input condition, we map it to an event followed by a state as shown in the construct 18. If the condition a non-sese output condition, it is mapped to a state followed by an event as shown in the last construct. If the condition is only used for routing purpose, e.g. capturing the deferred choice or simple merge, it should be mapped to a state only as shown in the construct 15.

Yawl supports a special construct called *cancelation sets* in which the execution of a task will remove tokens from nodes irrespective to how many tokens there are. Figure 4.7.a shows the *cancelation sets* is denoted by a dashed rounded rectangle. Enabling of task *"Cancel"* does not depend on tokens in the dashed area. All remaining tokens in this area are removed at the moment task *"Cancel"* is executed. Unfortunately, the currently canonical process format does not support this kind of construct. Thus, information concerning when and what to be discarded within the process execution cannot be preserved during mapping to CPF.

Figure 4.7.b only depicts the corresponding CPF model converted from the Yawl model on the left hand side using the mapping rules described in Table 4.4.

	Yawl constructs	Canonical representations
1		
2		
3		
4		
5		
6		
7		
8		
9		 Task @ -> S Event @ -> in Task @ -> B Task @ -> C Event @ -> out Task @ -> E Sub-net A
10		 Task @ -> S Event @ -> in Task @ -> B Task @ -> C Event @ -> out Task @ -> E Sub-net A
11		 Task @ -> S Event @ -> in Event @ -> out ANDSplit @ -> As Task @ -> E Task @ -> F Sub-net A
12		 Task @ -> S Task @ -> G ORJoin @ -> Aj Event @ -> in Event @ -> out ANDSplit @ -> As Task @ -> E Task @ -> F Sub-net A

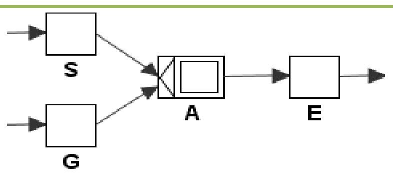
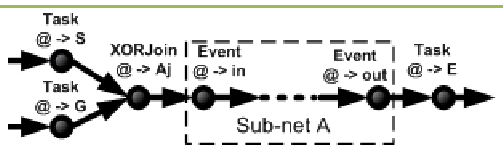
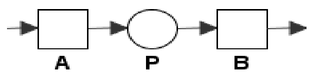
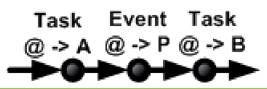
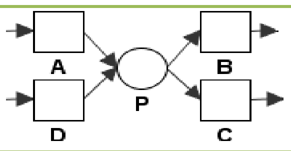
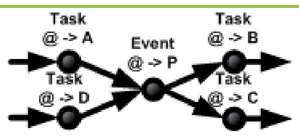
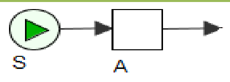
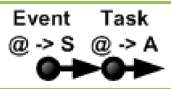
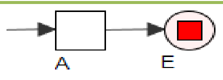
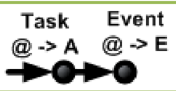
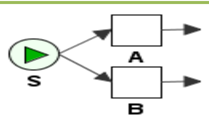
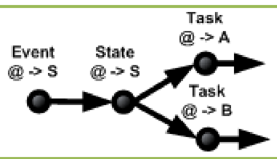
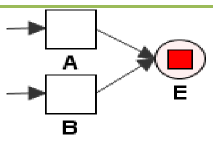
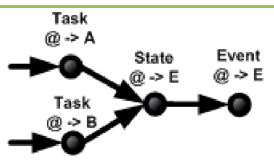
13		 Task @ -> S Task @ -> G XORJoin @ -> Aj Event @ -> in Event @ -> out Task @ -> E Sub-net A
14		
15		 Task @ -> A Event @ -> P Task @ -> B Task @ -> D Task @ -> C
16		
17		
18		
19		

Table 4.3: Canonical representations of 19 Yawl constructs

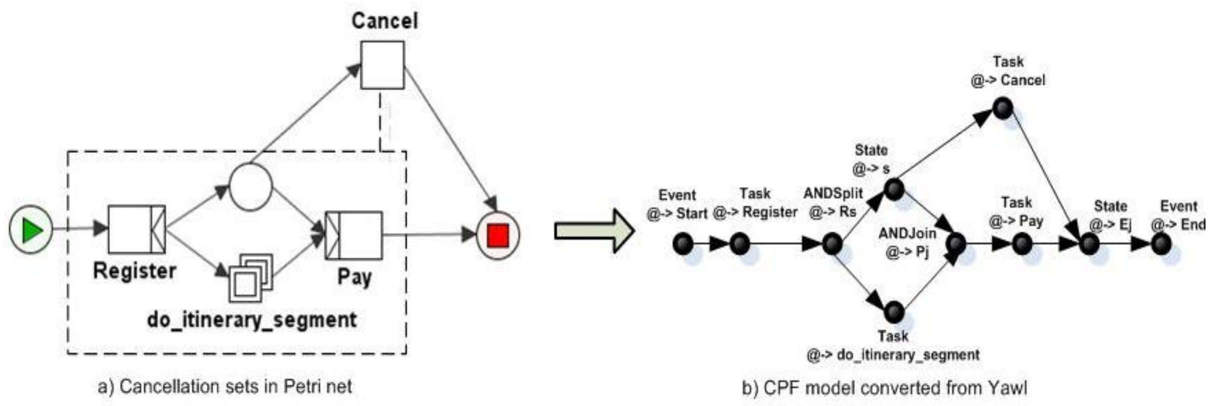


Figure 4.7: Cancelation sets modeled in Yawl and the corresponding CPF model

4.2.4 EPC to CPF

Event-driven process chain is an industry process modeling language represented by an EPC diagram. EPC diagram contains only three modeling elements to capture process control flow such as *functions*, *events* and *logical connectors*. Table B.4 in Appendix B describes the mapping rules from EPC elements to CPF elements. EPC functions are used only to represent activities and do not involve any routing patterns. Therefore, functions should be mapped to canonical tasks. In EPC, the flow of process is captured by logical connectors. Thus, EPC connectors should be mapped to canonical splits and joins. Events in EPC describe under what circumstance a function works or which state a process results in and they also indicate the beginning and the end of a process.

EPC events are usually mapped to canonical events if they do not immediately follow an (X)OR-split. Since in EPC, the events following an (X) OR-split are used to indicate the condition upon which the choice is made. Therefore, events that follow immediately an (X)OR-split must be mapped to canonical edges and the labels of the events are mapped to the labels of the canonical edges.

Table 4.4 describes mapping in more detail by illustrating canonical representation of nine EPC constructs. The first three constructs represent mapping of EPC split-connectors. The next three constructs represents mapping of EPC join-connectors. The seventh construct illustrates mapping of an EPC function. The eighth construct illustrate mapping of an EPC event that does not immediately follow an (X)OR-split. The last construct shows mapping of events that follow immediately an XOR – split.

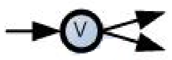
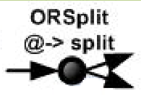
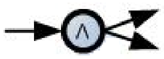
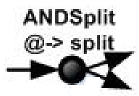
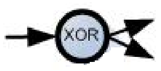
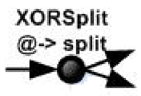
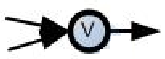
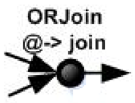
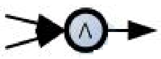
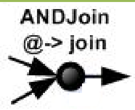
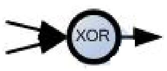
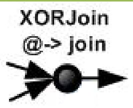
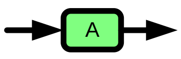
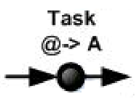
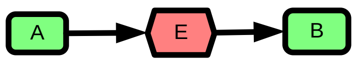
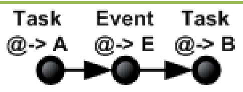
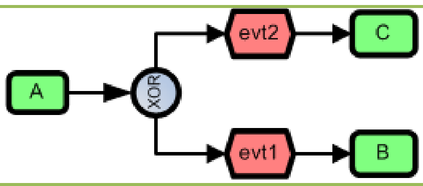
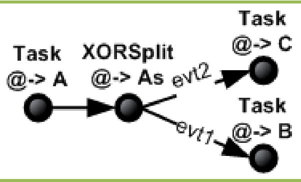
	EPC constructs	**Canonical representations**
1		
2		
3		
4		
5		
6		
7		
8		
9		 Task @-> A XORSplit @-> As evt2 evt1 Task @-> C Task @-> B

Table 4.4: Canonical representation of nine EPC constructs

It should be noticed that there might be a situation in which ending events follow an (X) OR-split. In this case, if we apply the mapping rules described in Table 4.4, the resulting CPF model will contain open edges pointing to nowhere. Although this situation rarely happens in reality, this should be avoided. Thus, for this specific case, a proposal is to add extra “silent” tasks following (X)OR-split as shown in Figure 4.8.

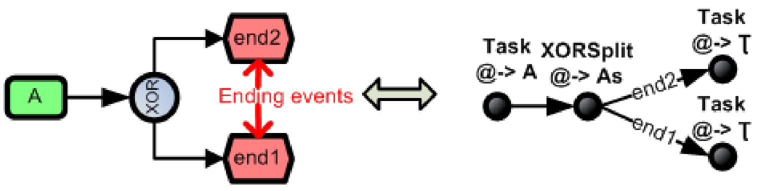


Figure 4.8: Canonical representation of an EPC model with 2 ending events following an XOR-split

4.2.5 Conversion to CPF in general

According to [6], canonical elements are identified from an analysis of commonalities among various process modeling languages. Thus canonical process format is able to capture structural characteristics of many process models. Any existing process models that are common in the majority of modeling languages can be mapped to CPF model using the following rules:

1. Each element from original model which represents an activity or a silent step is mapped to a canonical task. If the element involves routing behavior, its routing behavior must be separated and mapped to canonical split/join nodes.
2. Each element which indicates only routing behavior (e.g. EPC connectors) is mapped to a canonical split or join depending on what kind of routing is described in the original element.
3. Each element that indicates something happening during the process execution or describes under what circumstances the process works is mapped to a canonical event or state. If the element has at most one incoming and/or outgoing arc, it is mapped to an event. If the element has multiple incoming and/or outgoing arcs, it is mapped to a canonical state.

Those rules make it possible to create an equivalent CPF model of a process model described in many types of process modeling languages. However, resulting CPF model may not be able to reflect correctly the complete behavior of the original model since some characteristics of the original models cannot be preserved in canonical process format such as multiple-instances tasks, cancelation tasks or exclusive choice of Petri net.

4.2.6 Conversion from CPF to other process models

In the previous sections, we have explained the mapping rules of four different process models (e.g. Petri net, Yawl, EPC and Flexible model) into an equivalent canonical process format. We observe that each element of these process models has a corresponding canonical element. However, the other way around is not true. As we can see from Table B.5 (Appendix B) which shows the mapping rules of CPF elements, not all of the canonical elements can be mapped to a corresponding element in each of these process models. Hence mapping from CPF model to the original process models may be more difficult than mapping from the original models to CPF model.

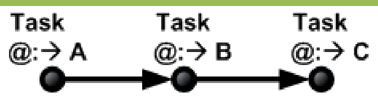
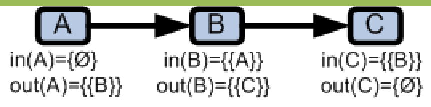
Furthermore, three of these process models require some strict structural constraints. Structural constraints indicate requirements which the structure of a certain process model must satisfy. For example, Petri net requires that there must be exactly one place in between any consecutive transitions. Places are not allowed to connect to each other and every transition must have at least one input place and one output place. Similarly, Yawl requires that there must be exactly one task in between any two consecutive conditions and there must be exactly one input condition and one output condition. Moreover, tasks without any predecessors are not allowed in Yawl model. In EPC, functions and events must be alternate along a path. Events are not allowed to precede a (X) OR-split and cycles containing logical connectors only are not allowed in an EPC graph. In contrast, canonical process format does not have such requirements. Its structure varies depending on the structure of the process models which it captures. Therefore, mapping from CPF to such process models (i.e. Petri net, Yawl, and EPC) is difficult and takes a lot of efforts to guarantee the structural constraints of target models.

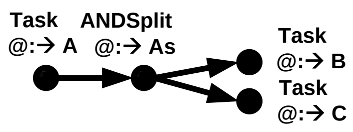
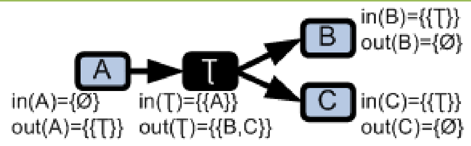
Apparently, flexible model contains only one node type (Task node) and it does not require any specific structural constraints. Hence, the mapping from CPF to flexible model seems to be easier and can be performed straightforwardly in such a way that all canonical elements including those which don’t have any corresponding flexible elements are mapped to flexible tasks. The mapping rule is described as follows:

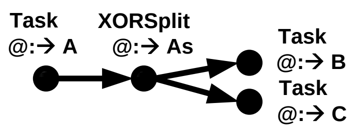
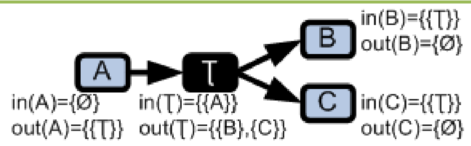
1. Each canonical task is mapped to a flexible task. No split and join behavior are involved in the resulting flexible task.
2. Each canonical split or join is mapped to a flexible task which is regarded as an invisible task. The split or join behavior is specified by the output set or input set of the resulting flexible task.
3. Each canonical event including message event and timer event is forced to be mapped to a flexible invisible task. The resulting flexible task does not contain split and join behavior.
4. Each canonical state is mapped to a flexible task and regarded as an invisible task. The XOR-split and/or XOR-join behavior of the canonical state is specified by the output and/or input set of the resulting flexible task.

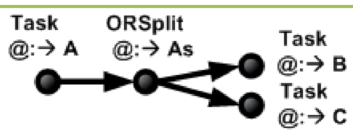
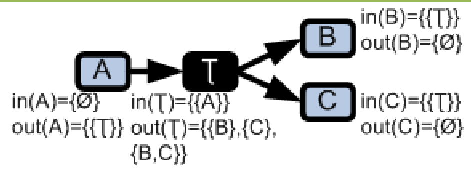
In table 4.5, we describe the mapping in more detail by illustrating flexible representation of nine canonical constructs. The first construct represents the mapping of canonical tasks without split/join behavior. In the construct 2, 3 and 4, we show the mapping of canonical split nodes. The next three constructs (5, 6 and 7) illustrate the mapping of canonical joins. The mapping of canonical event and state are shown in construct 8 and 9, respectively.

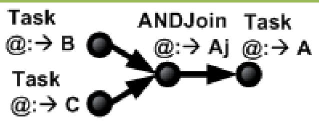
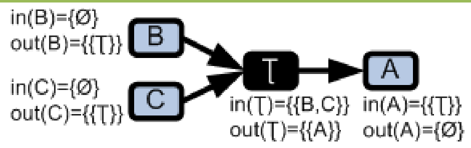
	Canonical constructs	Flexible representations

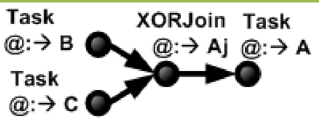
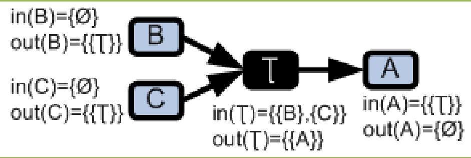
1		 A
		B
		C
		in(A)={Ø} out(A)={{B}}
		in(B)={{A}} out(B)={{C}}
		in(C)={{B}} out(C)={Ø}

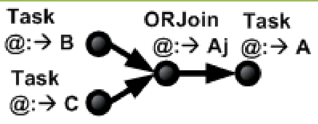
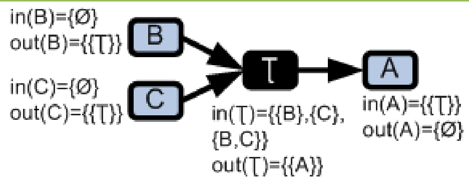
2		 A
		T
		B
		C
		in(A)={Ø} out(A)={{T}}
		in(T)={{A}} out(T)={{B,C}}
		in(B)={{T}} out(B)={Ø}
		in(C)={{T}} out(C)={Ø}

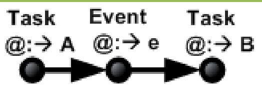
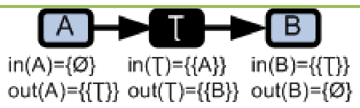
3		 A
		T
		B
		C
		in(A)={Ø} out(A)={{T}}
		in(T)={{A}} out(T)={{B},{C}}
		in(B)={{T}} out(B)={Ø}
		in(C)={{T}} out(C)={Ø}

4		 A
		T
		B
		C
		in(A)={Ø} out(A)={{T}}
		in(T)={{A}} out(T)={{B},{C}, {B,C}}
		in(B)={{T}} out(B)={Ø}
		in(C)={{T}} out(C)={Ø}

5		 B
		C
		T
		A
		in(B)={Ø} out(B)={{T}}
		in(C)={Ø} out(C)={{T}}
		in(T)={{B,C}} out(T)={{A}}
		in(A)={{T}} out(A)={Ø}

6		 B
		C
		T
		A
		in(B)={Ø} out(B)={{T}}
		in(C)={Ø} out(C)={{T}}
		in(T)={{B},{C}} out(T)={{A}}
		in(A)={{T}} out(A)={Ø}

7		 B
		C
		T
		A
		in(B)={Ø} out(B)={{T}}
		in(C)={Ø} out(C)={{T}}
		in(T)={{B},{C}, {B,C}} out(T)={{A}}
		in(A)={{T}} out(A)={Ø}

8		 A
		T
		B
		in(A)={Ø} out(A)={{T}}
		in(T)={{A}} out(T)={{B}}
		in(B)={{T}} out(B)={Ø}

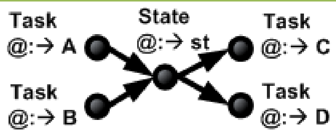
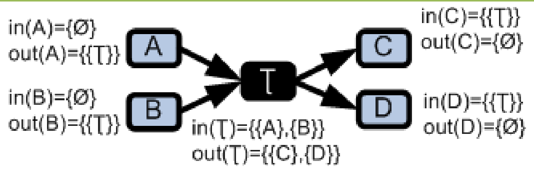
9	 Task @:→ A
	Task @:→ B
	State @:→ st
	Task @:→ C
	Task @:→ D
		 A
		B
		T
		C
		D
		in(A)={Ø} out(A)={{T}}
		in(B)={Ø} out(B)={{T}}
		in(T)={{A},{B}} out(T)={{C},{D}}
		in(C)={{T}} out(C)={Ø}
		in(D)={{T}} out(D)={Ø}

Table 4.5: Flexible representation of nine Canonical constructs

Chapter 5

Log replay on canonical process format

In Chapter 4, we define the motivation to generalize existing process modeling languages into canonical process format. In this chapter, we discuss the log replay technique in canonical process format. First of all, we specify some requirements for replay on CPF. Secondly, we introduce several replay approaches that can be applied in CPF model and then we choose a solution that is most suitable for CPF. At the end, we illustrate the implementation of the CPF replay approach in ProM framework.

5.1 Requirement

Firstly, there must be an event log and a pre-defined CPF model available for log replay. Since replay technique is implemented in ProM6, the format of event log must be either MXML or XES. Secondly, we must undertake a mapping between events in log and tasks in CPF model. Each event will be mapped to a related task in a way that they represent the same activity. An event can be mapped to multiple task nodes. Events that do not correspond to any tasks should be removed from event log. Tasks that do not have any related events must be mapped to *invisible* tasks or *unmapped visible* tasks. Invisible tasks are used for routing purpose or for delaying the real tasks in process model while unmapped visible tasks refer to activities which are not recorded in event log.

5.2 Approach

There are several approaches which can be used to enable log replay on CPF model. The first approach is to replay event log directly on CPF model. However, there is a serious problem with this approach. The canonical process format was originally developed without any execution semantics, for example it does not use the notion of initial marking, notion of tokens or notion of input/output sets (which is available in flexible model). As we realize that execution semantics are very important for replaying an event log since it indicates from which tasks the process can start, which tasks will be enabled by which tasks or which tasks are executed successfully, etc. In addition, the structure of CPF model is not enforced to follow specific structural constraints, e.g. event must occur in between tasks, events and tasks should alternate along a path or process model must start and end at events. The idea of CPF model is to capture only the core split/join behaviors and structure characteristic of existing process modeling languages. Therefore, CPF structure is flexible depending on what kind of process model it captures. In order to perform the replay directly on CPF model, it is necessary to extend the CPF with specific execution semantics such that they are suitable in current CPF structure. This would create a lot of difficulties and take hard efforts. Thus, this approach seems not to be a good choice for the moment.

The other approach is to replay event log indirectly via an intermediate notation. We have shown in Section 3 that log replay currently exists on Petri net, fuzzy model and flexible model. Hence, log replay can be performed indirectly by first converting CPF model to either Petri net, fuzzy model or flexible model and then apply the existing log replay methods of these modeling languages. The problem of how to replay an event log on CPF model becomes how to convert CPF model into Petri net, fuzzy model or flexible model.

As we have indicated in Section 3.4.3, fuzzy model does not use notion of invisible tasks, any invisible tasks from other process models if possible will be mapped to real tasks in fuzzy model. Current fuzzy replay method was not invented to deal with invisible tasks. Apparently, without execution of invisible tasks during replaying an event log, number of unsatisfied events will be increased. Furthermore, there are no concrete conversion methods available for mapping from exiting process models to fuzzy model. Thus, implementing log replay on CPF via Fuzzy model seems not to be a good direction.

In Section 4.2, we have stated that mapping from CPF model to flexible model is easier than mapping to Petri net. Besides, there is no conversion method invented for mapping from CPF model to Petri net at the moment, whereas a method for mapping to flexible model from CPF model has been introduced. Moreover, the current Petri net replay method contains some problems that can be solved by flexible model replay technique. Thus, enabling replay on CPF model via flexible model seems to be the promising approach.

5.3 Solution

We choose to perform log replay indirectly on CPF model by first converting it to an equivalent flexible model then apply the existing flexible model replay technique. However, flexible model does not use notion of *starting task node*. Every node without predecessors is regarded as initial node. Initial nodes with empty input set are always enabled. Therefore, flexible model replay will always execute initial nodes at every selected instance in search tree. This explores a massive number of new created instances given a model with huge number of initial tasks and a long event trace. Moreover, the existence of value ***UnA*** (an unhandled arc cost) in the calculation of cost function $f(n)$ potentially leads to a bad result as illustrated in Section 3.3.5. Therefore, we extend the current flexible model replay technique to the following extents:

Notion of starting task nodes will be used. Starting task nodes are the nodes without predecessors for which the input set is empty (i.e. in (s) =∅). This is different from a node for which input set contains an empty set (i.e.in (s) = {∅}). By default, every starting task node is enabled at the beginning of process. Once it is executed either by **move on model only** or **move on both log and model**, it is no longer enabled under the term of starting task node. If starting task nodes are not executed, their enabling remains available.

Unhandled arc cost (UnA) is removed from the calculation of cost function $f(n)$ for any node n in search tree. Precisely, unhandled arc cost (UnA) is not involved in the caluculation of subpart $g(n)$ of $f(n)$. The new cost function $f(n)$ for each explored node n in search tree is then defined as: $f(n)= g(n) + h(n)$, where:

- $h(n) = L * E$**,** where *L* is the number of events left in the case at node n and *E* is the cost of an event left in the case.
- $g(n)$ *= A*RE + C*Minv + D*Mr + F*Rem + G* Us,* where *A* is the number of events replayed correctly so far, *C* is the number of executed invisible tasks, *D* is the number of visible tasks executed without replaying any events from the case, *F* is the number of events removed

from the case so far and *G* is the number of events replayed incorrectly (i.e. unsatisfied events). *Re* is the cost of replaying an event correctly, *Minv* is the cost of executing an invisible task, *Mr* is the cost of executing a visible task, *Rem* is the cost of removing an event from the case and *Us* is the cost of replaying an event incorrectly.

Obviously, the estimate heuristic cost function $h(n)$ remains unchanged. The only change is in the calculation of $g(n)$ in which the number of unhandled arcs is not concerned any more. By this way, we can prevent the problem illustrated in Figure 3.10. However, to avoid improper completion as shown in Figure 3.9, we extend the replay algorithm in such a way that the decision for selecting the next node to be explored is prioritized as follows. The one with smallest value $f(n)$ is always selected first. If there is more than one candidate whose value $f(n)$ is the smallest, we choose the one containing the smallest number of created unhandled arcs so far. If there is more than one node satisfying both conditions, we pick up one randomly for next iteration.

As an example, we illustrate the replay of trace $A-C-D$ on a flexible model (Figure 5.1) with the extended flexible model relay technique. The resulting search tree is shown in Figure 5.2

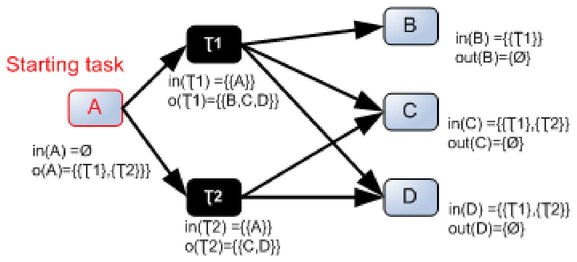


Figure 5.1: Flexible model contains a starting task node A

We observe that the costs to execute $A-\tau1$ and $A-\tau2$ are the same, i.e. the value of cost function $f(n)$ generated at *instance1* and *instance2* are the smallest ($f(ins1) = f(ins2) = 3001$). However, execution of task $\tau1$ creates three unhandled arcs whereas execution of task $\tau2$ produces only two unhandled arcs. Consequently, node *instance2* is selected for next exploration. This results in the optimal execution sequence $A-\tau2-C-D$ as expected. Moreover, if the starting task is executed, it will never been executed again under the term of starting task, neither by ***move on model only*** nor ***move on both log and model***. For instance, once the staring task A is executed, execution of *A* will not be happened again along every paths starting from initial instance. Therefore, a large number of new created instances caused by executing starting task *A* are decreased. This certainly increases the performance of replay algorithm.

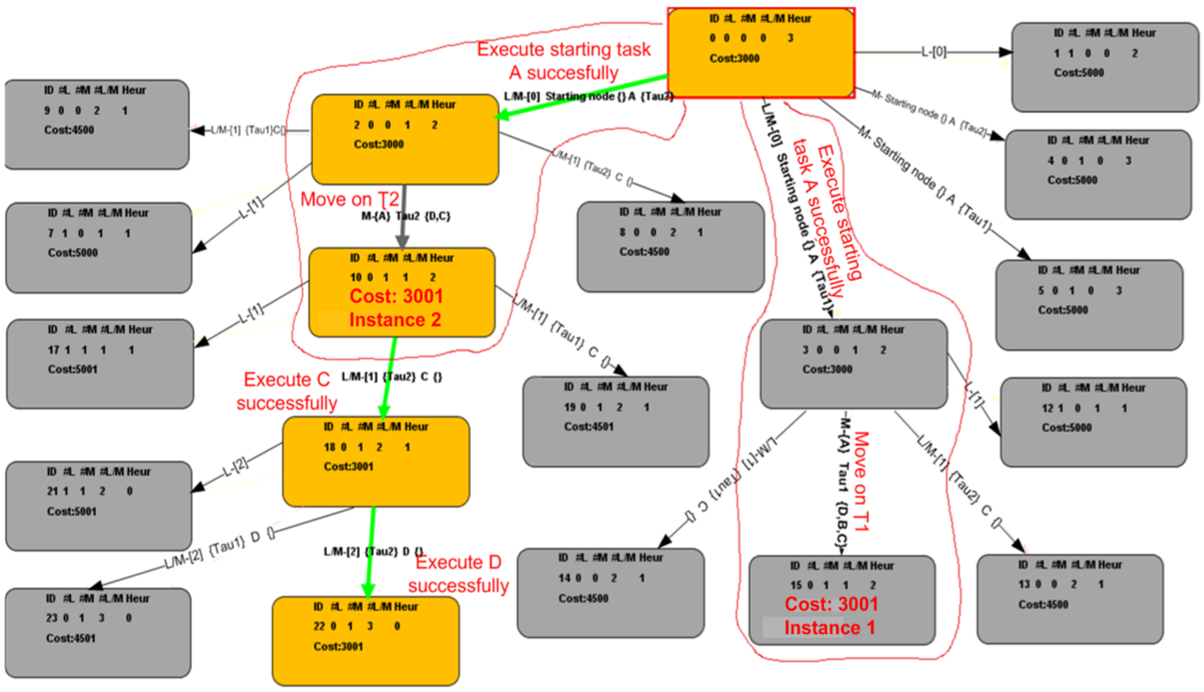


Figure 5.2: Search tree generated by replaying A-C-D contains an optimal path

5.4 Limitation

Although the use of starting task node can reduce a number of new created instances which need to be explored and hence improve the performance. It cannot fix the problem completely. This is because the problem still occurs if the replayed model contains a large volume of duplicate tasks. Each one of the duplicated tasks generates a different scenario of task execution, thus the replay algorithm still create a large number of new instances needed to be explored in search tree to find the most optimal scenario.

5.5 Implementation

Our CPF replay approach is implemented in ProM framework. Besides, we also implemented plug-ins that convert existing process modeling languages to CPF model. Figure 5.3 shows the screenshot of choosing plug-ins for converting Petri net, fuzzy model, EPC and flexible model into CPF.

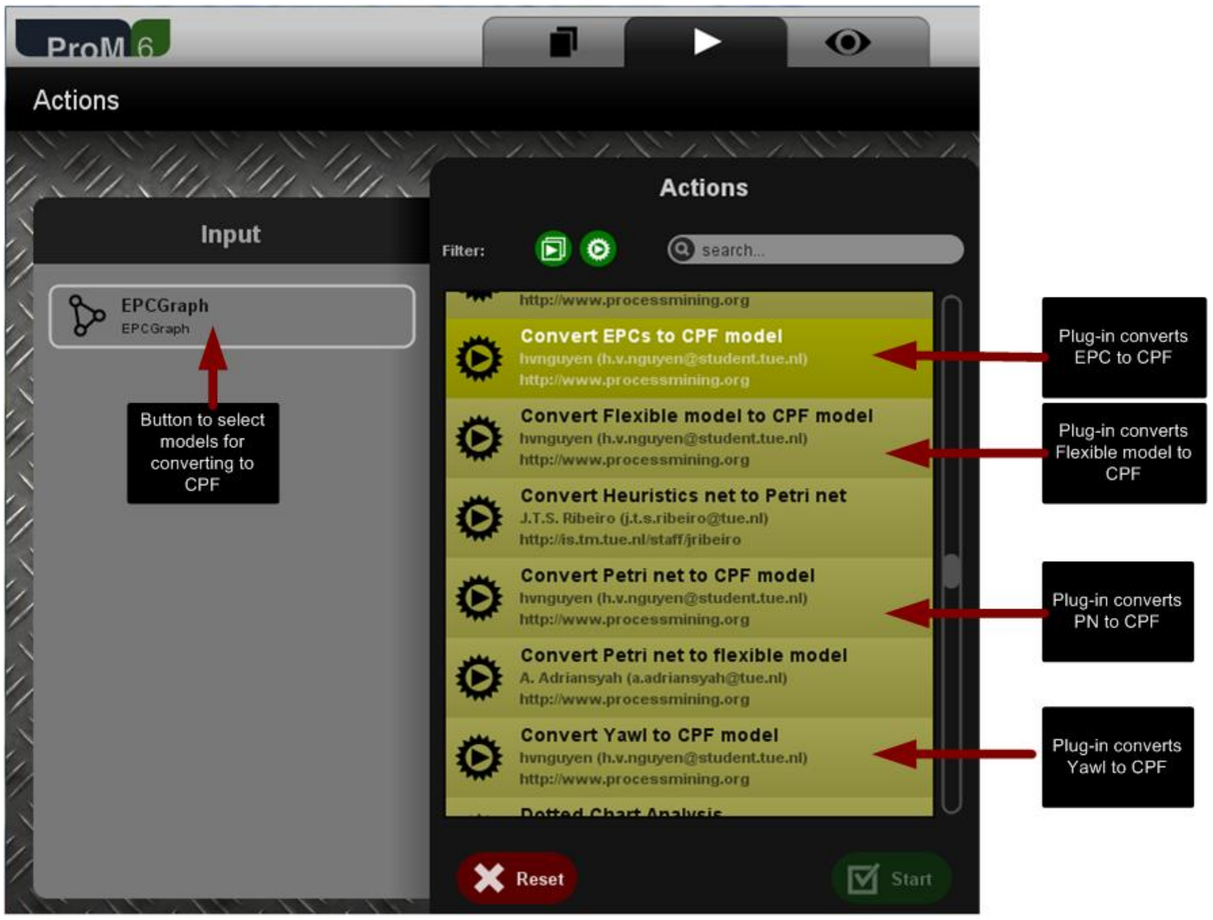

ProM 6
Actions
Input
EPCGraph
EPCGraph
Button to select models for converting to CPF
Actions
Filter:
search...
http://www.processmining.org
Convert EPCs to CPF model
hvnguyen (h.v.nguyen@student.tue.nl)
http://www.processmining.org
Convert Flexible model to CPF model
hvnguyen (h.v.nguyen@student.tue.nl)
http://www.processmining.org
Convert Heuristics net to Petri net
J.T.S. Ribeiro (j.t.s.ribeiro@tue.nl)
http://is.tm.tue.nl/staff/jribeiro
Convert Petri net to CPF model
hvnguyen (h.v.nguyen@student.tue.nl)
http://www.processmining.org
Convert Petri net to flexible model
A. Adriansyah (a.adriansyah@tue.nl)
http://www.processmining.org
Convert Yawl to CPF model
hvnguyen (h.v.nguyen@student.tue.nl)
http://www.processmining.org
Dotted Chart Analysis
Reset
Start
Plug-in converts EPC to CPF
Plug-in converts Flexible model to CPF
Plug-in converts PN to CPF
Plug-in converts Yawl to CPF

Figure 5.3: Screenshot of choosing a plug-in for converting to CPF

We implemented replay on both a single case and a whole log. Figure 5.4 shows the screenshot of choosing a plug-in for replaying a case or a whole log. It is required that a CPF model and an event log must be available in ProM in advance. The CPF model can be obtained by importing from outside or converting from other process model using the plug-ins described in Figure 5.3. The event log must be imported from outside. We choose to perform replay on a single event case first using *"Replay a case on CPF model"* plug-in. After selecting an event log and a CPF model, we click on "Start" button to begin the replay. Once the "Start" button is clicked, the conversion from CPF model to an equivalent flexible model is taken implicitly using mapping rules described in Section 4.2.6.

The initial step of the replay is to map events in log into tasks in CPF model as illustrated in Figure 5.5. Tasks which are not related to any events should be mapped to "*invisible*" or "*visible but unmapped*".

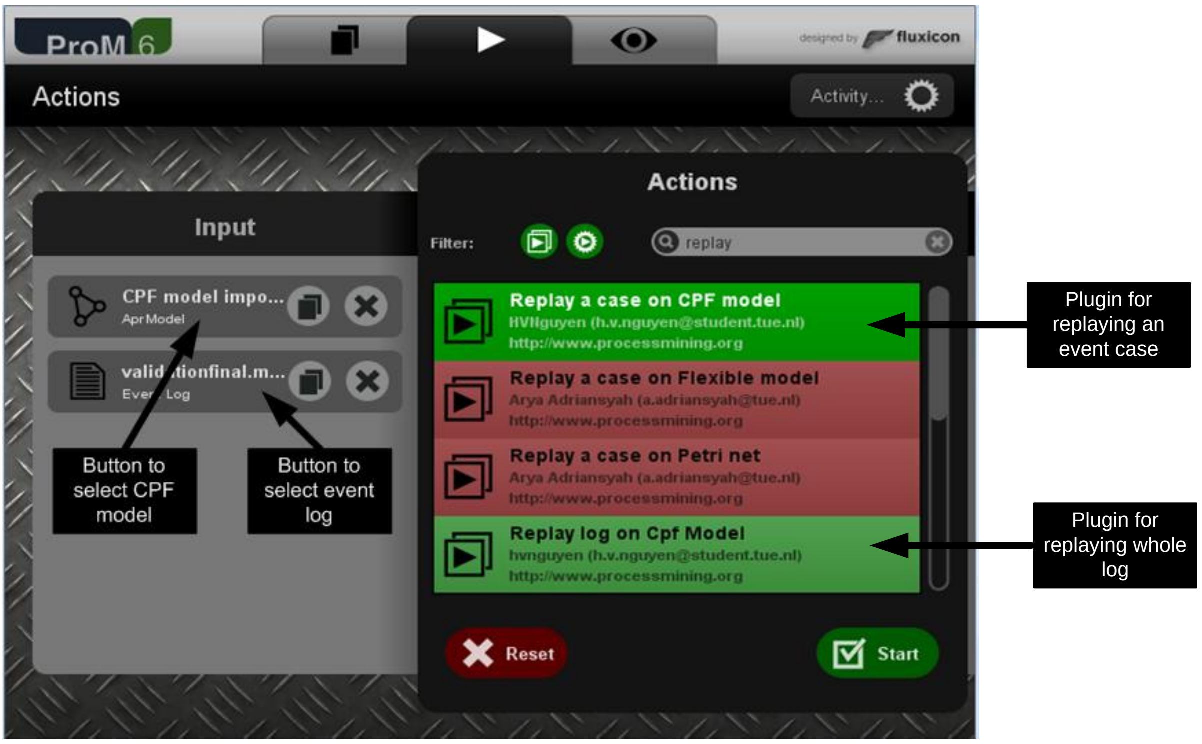
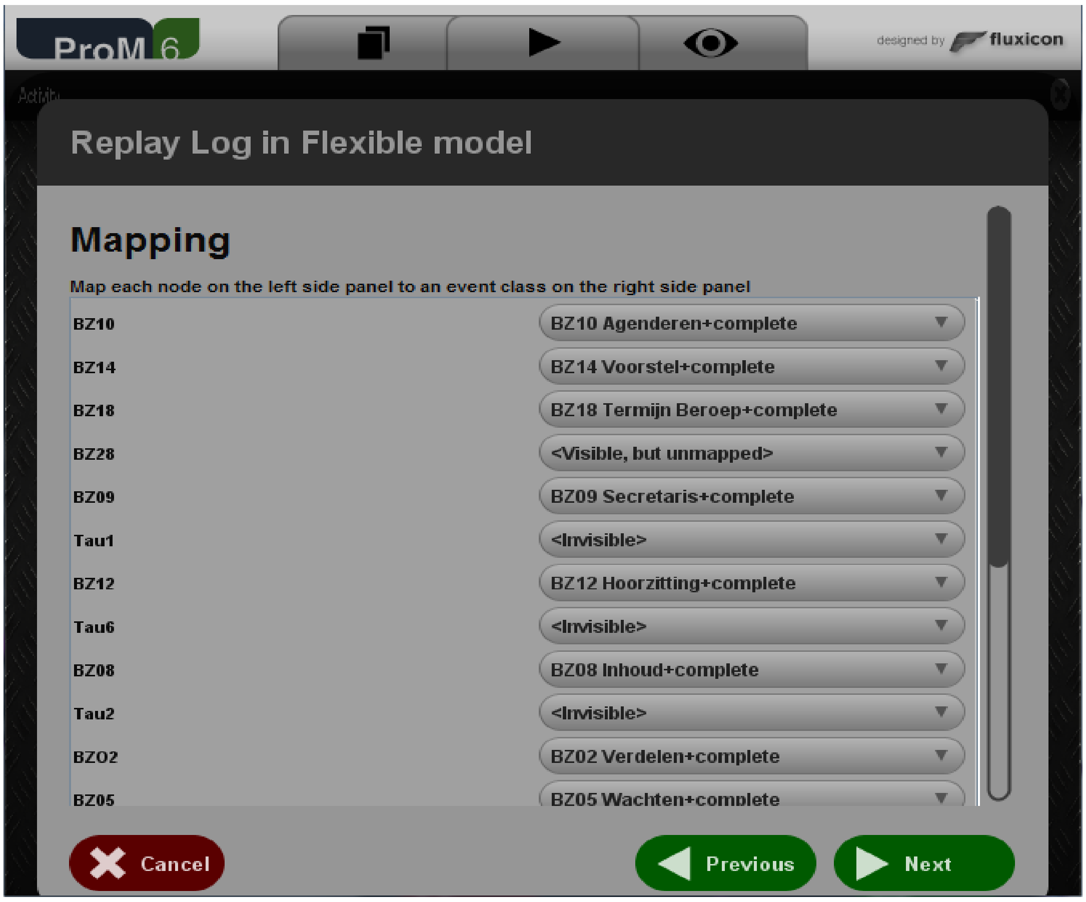

ProM 6
designed by fluxicon
Actions
Activity...
Input
Actions
Filter:
replay
CPF model impo...
Apr Model
validationfinal.m...
Event Log
Button to select CPF model
Button to select event log
Replay a case on CPF model
HVNguyen (h.v.nguyen@student.tue.nl)
http://www.processmining.org
Replay a case on Flexible model
Arya Adriansyah (a.adriansyah@tue.nl)
http://www.processmining.org
Replay a case on Petri net
Arya Adriansyah (a.adriansyah@tue.nl)
http://www.processmining.org
Replay log on Cpf Model
hvnguyen (h.v.nguyen@student.tue.nl)
http://www.processmining.org
Plugin for replaying an event case
Plugin for replaying whole log
Reset
Start

Figure 5.4: Screenshot of choosing replay plug-ins

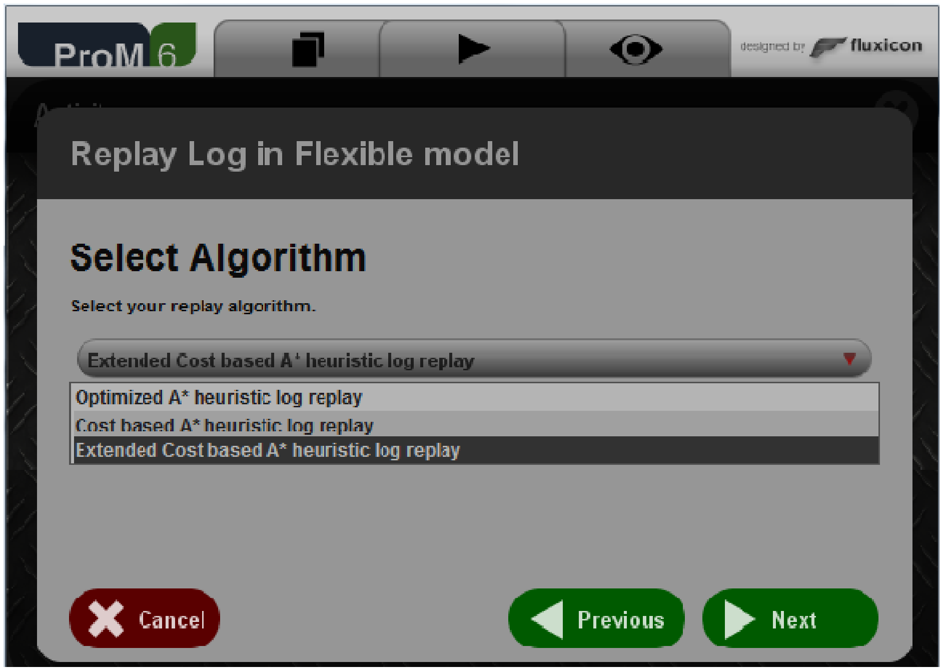


Map each node on the left side panel to an event class on the right side panel

BZ10	BZ10 Agenderen+complete
BZ14	BZ14 Voorstel+complete
BZ18	BZ18 Termijn Beroep+complete
BZ28	<Visible, but unmapped>
BZ09	BZ09 Secretaris+complete
Tau1	<Invisible>
BZ12	BZ12 Hoorzitting+complete
Tau6	<Invisible>
BZ08	BZ08 Inhoud+complete
Tau2	<Invisible>
BZ02	BZ02 Verdelen+complete
BZ05	BZ05 Wachten+complete

Figure 5.5: Screenshot of mapping step

In the second step, we choose the desired replay algorithm from a list box as shown in Figure 5.6. For our replay approach, we select *"Extended Cost based A* heuristic log replay"* algorithm. This algorithm makes use of starting task nodes and removes unhandled arc cost from calculating total cost $f(n)$ for each instance to be evaluated in search tree.


Replay Log in Flexible model

Select Algorithm

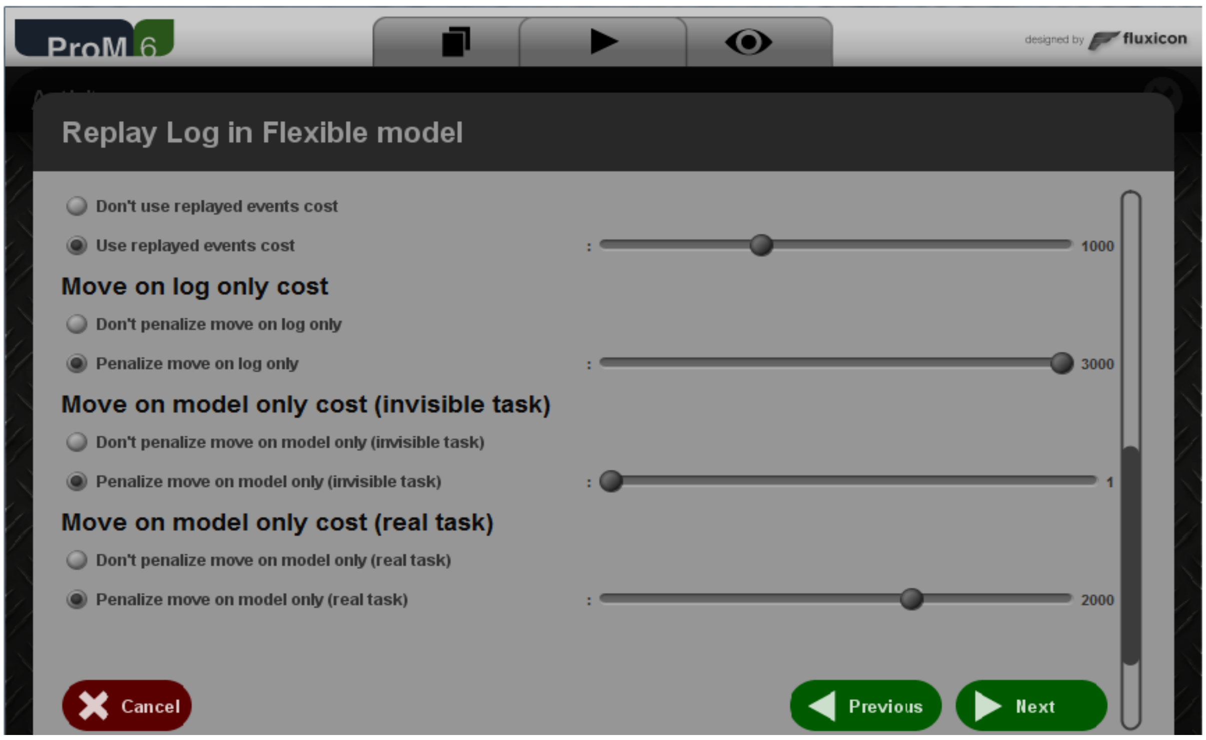
Select your replay algorithm.

Extended Cost based A* heuristic log replay

Optimized A* heuristic log replay
Cost based A* heuristic log replay
Extended Cost based A* heuristic log replay

Figure 5.6: Screenshot of choosing replay algorithm step

In the next step, we configure the cost of parameters required for calculating cost $f(n)$ such as unsatisfied event cost (Us), event to be replayed cost (E), replayed event cost (RE), move on log only cost (Rem), move on model only with real task cost (Mr) and move on model only with invisible task cost (Minv). Furthermore, we can set up the maximum number of instances can be explored in search tree during replay. Figure 5.7 shows the cost configuration panel in which we adjust the slide bar to change the cost value of each parameter. Next we choose an event case to be replayed as shown in Figure 5.8.



Replay Log in Flexible model

- Don't use replayed events cost
- Use replayed events cost : 1000

Move on log only cost

- Don't penalize move on log only
- Penalize move on log only : 3000

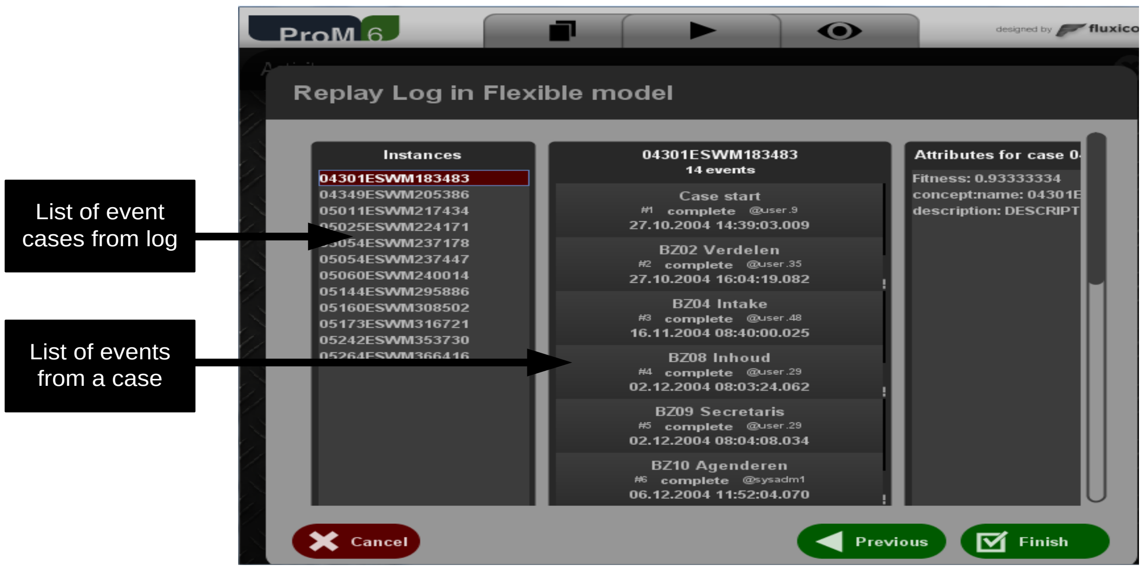
Move on model only cost (invisible task)

- Don't penalize move on model only (invisible task)
- Penalize move on model only (invisible task) : 1

Move on model only cost (real task)

- Don't penalize move on model only (real task)
- Penalize move on model only (real task) : 2000

Figure 5.7: Screenshot of cost configuration step



Replay Log in Flexible model

Instances
04301ESWM183483
04349ESWM205386
05011ESWM217434
05025ESWM224171
05054ESWM237178
05054ESWM237447
05060ESWM240014
05144ESWM295886
05160ESWM308502
05173ESWM316721
05242ESWM353730
05264ESWM366416

04301ESWM183483
14 events

Case start
#1 complete @user.9
27.10.2004 14:39:03.009

BZ02 Verdelen
#2 complete @user.35
27.10.2004 16:04:19.082

BZ04 Intake
#3 complete @user.48
16.11.2004 08:40:00.025

BZ08 Inhoud
#4 complete @user.29
02.12.2004 08:03:24.062

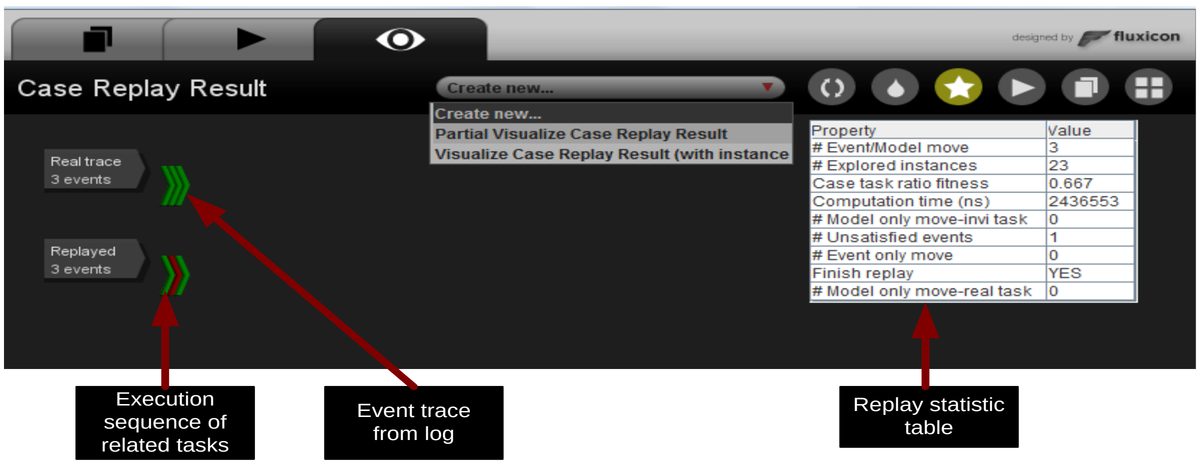
BZ09 Secretaris
#5 complete @user.29
02.12.2004 08:04:08.034

BZ10 Agenderen
#6 complete @sysadm1
06.12.2004 11:52:04.070

Attributes for case 0
Fitness: 0.93333334
concept:name: 04301E
description: DESCRIPT

Figure 5.8: Screenshot of selecting an event case step

Result of the replaying a case is illustrated in Figure 5.9. This view shows two traces on vertical axis. The above trace represents sequence of replayed events in the case and the below trace represents sequence of executed tasks comparing to the replayed events. Each trace is shown as a stream of wedged-shaped segments. Each segment in the event sequence represents an event in the case and colored in green, extending from left to right in their given order. Each segment in task sequence represents an executed task in CPF model or a removed event in the case and has different colors. Red segment corresponds to an invalid task, i.e. task is executed improperly, whereas green segment corresponds to a valid task. Purple segment signals *move on model only* with real task and grey segment signals *move on model only* with invisible task. Yellow segment indicates an event is removed from the log. There is also a table containing case statistic of replaying a case, such as number of unsatisfied events, number of *move on model only* with real/invisible task, number of removed events, case task ratio fitness, computation times, etc. We can also view result of replaying a case in form of a search tree (as shown in Figure 5.10) by selecting "Visualize Case Replay Result (with instance)" from drop box. The search tree contains an optimal path denoted by yellow nodes. Edges along optimal path correspond to execution of tasks and have different colors. The meaning of color is similar to the meaning of segment's color such as: red edges indicate invalid tasks whereas green edges indicate valid tasks, etc.



Property	Value
# Event/Model move	3
# Explored instances	23
Case task ratio fitness	0.667
Computation time (ns)	2436553
# Model only move-invi task	0
# Unsatisfied events	1
# Event only move	0
Finish replay	YES
# Model only move-real task	0

Figure 5.9: Screenshot of replaying result

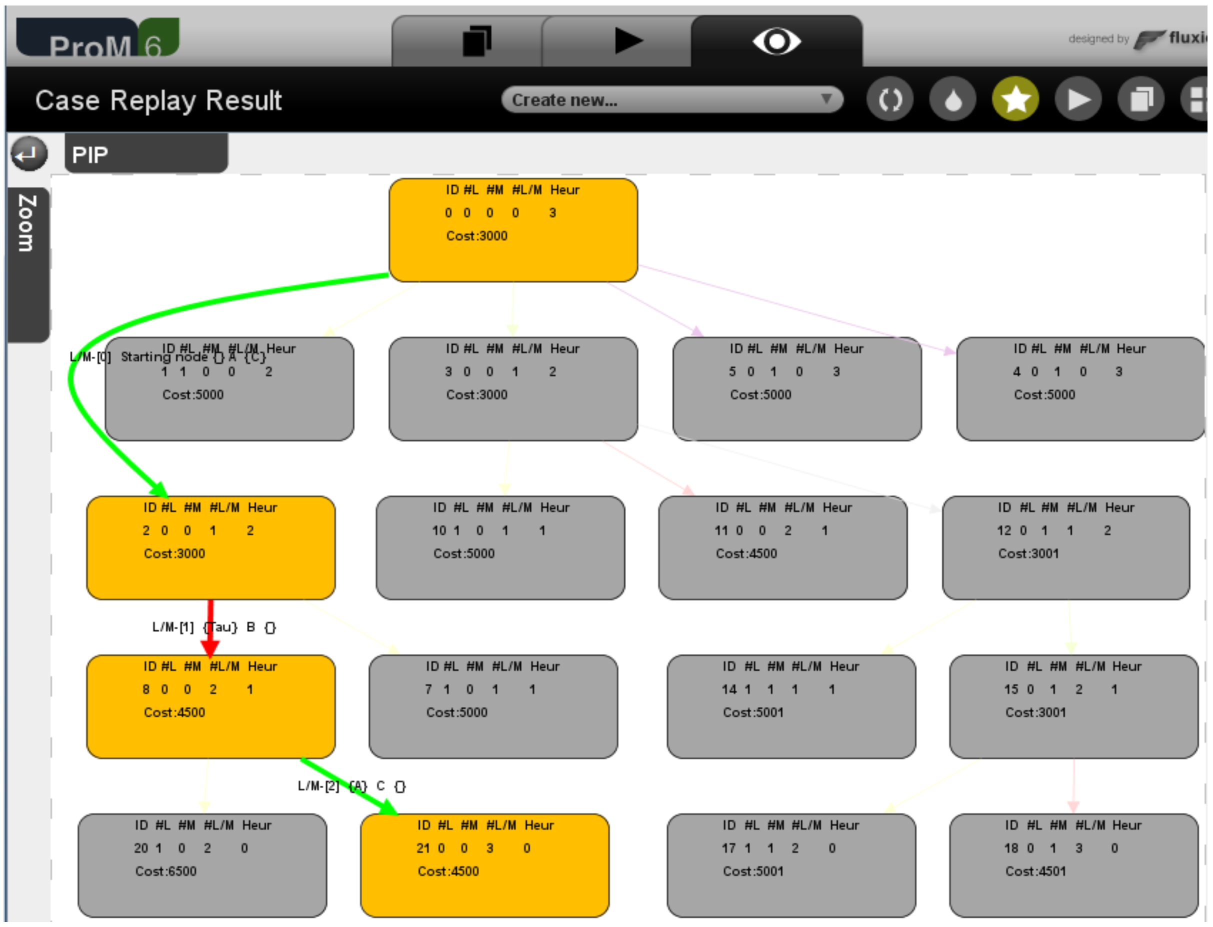
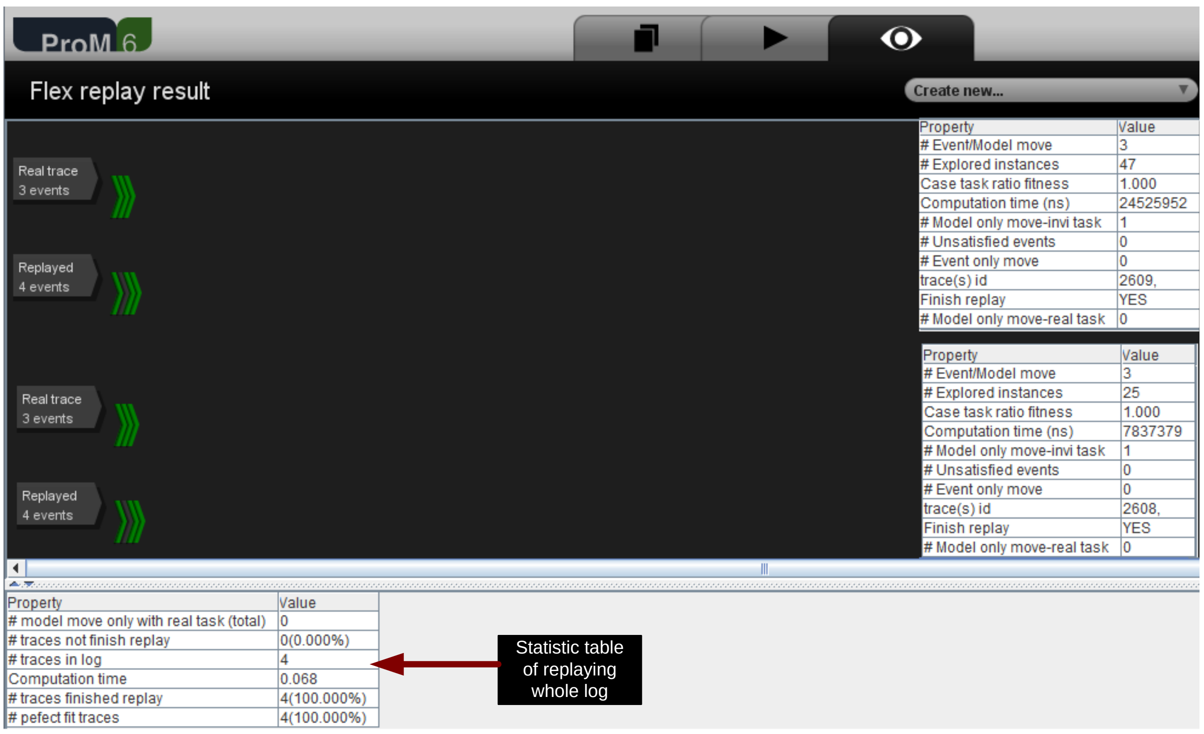

ProM 6
designed by fluxi
Case Replay Result
Create new...
PIP
Zoom
ID #L #M #L/M Heur 0 0 0 0 3 Cost:3000
L/M-[0] Starting node {} A {C}
ID #L #M #L/M Heur 1 1 0 0 2 Cost:5000
ID #L #M #L/M Heur 3 0 0 1 2 Cost:3000
ID #L #M #L/M Heur 5 0 1 0 3 Cost:5000
ID #L #M #L/M Heur 4 0 1 0 3 Cost:5000
ID #L #M #L/M Heur 2 0 0 1 2 Cost:3000
ID #L #M #L/M Heur 10 1 0 1 1 Cost:5000
ID #L #M #L/M Heur 11 0 0 2 1 Cost:4500
ID #L #M #L/M Heur 12 0 1 1 2 Cost:3001
L/M-[1] {Tau} B {}
ID #L #M #L/M Heur 8 0 0 2 1 Cost:4500
ID #L #M #L/M Heur 7 1 0 1 1 Cost:5000
ID #L #M #L/M Heur 14 1 1 1 1 Cost:5001
ID #L #M #L/M Heur 15 0 1 2 1 Cost:3001
L/M-[2] {A} C {}
ID #L #M #L/M Heur 20 1 0 2 0 Cost:6500
ID #L #M #L/M Heur 21 0 0 3 0 Cost:4500
ID #L #M #L/M Heur 17 1 1 2 0 Cost:5001
ID #L #M #L/M Heur 18 0 1 3 0 Cost:4501

Figure 5.10: Screenshot of search tree

We can also perform replay on a whole log using *"Replay log on CPF model"* plug-in. The steps to perform replay on a log are similar to the ones of replaying a case except that we do not need to select a case for replaying. Instead, the whole event log is relayed automatically. Furthermore, the final result does not contain a search tree. The log replay result is shown in Figure 5.11 in which each replayed case is represented by a pair of segment traces and a statistic table. Additionally, there is a table containing information about log replay statistic such as: the total task ratio fitness, number of events perfectly fitted, number of events finished replay, computational time, etc.



Property	Value
# Event/Model move	3
# Explored instances	47
Case task ratio fitness	1.000
Computation time (ns)	24525952
# Model only move-invi task	1
# Unsatisfied events	0
# Event only move	0
trace(s) id	2609,
Finish replay	YES
# Model only move-real task	0

Property	Value
# Event/Model move	3
# Explored instances	25
Case task ratio fitness	1.000
Computation time (ns)	7837379
# Model only move-invi task	1
# Unsatisfied events	0
# Event only move	0
trace(s) id	2608,
Finish replay	YES
# Model only move-real task	0

Property	Value
# model move only with real task (total)	0
# traces not finish replay	0(0.000%)
# traces in log	4
Computation time	0.068
# traces finished replay	4(100.000%)
# pefect fit traces	4(100.000%)

Figure 5.11: Screenshot of log replay result

Chapter 6

Evaluation

In Chapter 5, we introduced an approach to enable log replay on canonical process format. The idea of this approach is to perform log replay indirectly by converting CPF model to an equivalent flexible model and apply the existing flexible model replay technique which is extended with some new features. Given a CPF model and an event log, we can obtain lots of useful information especially related to the conformance by replaying the log on the CPF model. Based on [7], several conformance metrics are derived from log replay. In this thesis, we analyze one important conformance metric, namely *task ratio fitness* [7, p.8]. Thus, notion of task ratio fitness is introduced first in this chapter. Next, we describe an analysis performed on cost parameters which are required in the extended flexible replay to find the optimal task ratio fitness value. Finally, we present a case study to evaluate the task ratio fitness.

6.1 Task ratio fitness

Task ratio fitness is defined as the percentage of events that are unsatisfied after replaying an event log. Task ratio fitness is categorized into *case task ratio fitness* and *log task ratio fitness*. Given an event case to be replayed, the extended flexible model replay technique creates a search tree in which an optimal executing sequence of tasks is found. This optimal path contains a smallest number of invalid tasks (i.e. tasks are executed unsuccessfully). An invalid task corresponds to an unsatisfied event. Therefore, the number of unsatisfied events is equal to the number of invalid tasks found in the optimal sequence. The *case task ratio fitness* and the *log task ratio fitness* are then defined as follows:

- **Case task ratio fitness** $f_c^{rat} \in [0,1]$ is defined as: $f_c^{rat} = 1 - \frac{|E_c^{us}|}{|E_c|}$ where E_c^{us} is the set of unsatisfied events found from the optimal sequence after replaying the event case *c and* E_c is the set of events occurring in the event case *c.* f_c^{rat} indicates the percentage of unsatisfied events after replaying the case *c.*
- **Log task ratio fitness** $f^{rat} \in [0,1]$ is defined as: $f^{rat} = \frac{\sum_{c \in C} f_c^{rat}}{|C|}$ where *C* is the event log. f^{rat} indicates the average sum of all *case task ratio fitness* of each case in the event log *C.*

6.2 Parameter setting analysis

The extended flexible model replay technique requires configuring values of cost parameters such as cost of an event left in the case (*E*), cost of an event replayed correctly (*RE*), cost of an event replayed incorrectly (*U*s), cost of executing an invisible task (*Minv*), cost of executing a visible task

(*Mr*) and cost of removing an event from the case (*Rem*). We observe that different settings of cost parameters lead to different task ratio fitness values. In order to achieve the optimal task ratio fitness value, we need to find the most suitable setting of cost parameters. To do so, we define several constraints required for the cost parameters as follows:

1) $\underline{0 < E \wedge E \leq RE}$

The cost of an event left in the case *(E)* must be less than or equal to the cost of replaying an event correctly *(RE)*. In order to understand how we got this inequality, we study an example in which we replay event trace *A-B-C* in an arbitrary flexible model. Assume that we obtain a partial search tree as shown in Figure 6.1. *Node1* in the search tree is reached by replaying event *A* correctly. The best possible scenario to reach the preferred target node from *node1* is to replay the event *B* and *C* properly. The lowest cost to reach the preferred target node from *node1* is now equal to two times of the cost of replaying an event correctly *(2*RE).* The estimate cost at *node1* is calculated by multiplying the number of events left in the case with the cost of an event left in the case (i.e. $h(node1) = 2 * E$). In order for A * algorithm to be able to find an optimal path, the estimate cost $h(n)$ of each explored node n in the search tree should not overestimate the lowest cost to reach its preferred target. That is $2 * E \leq 2 * RE$ .This implies $E \leq RE$ **.**

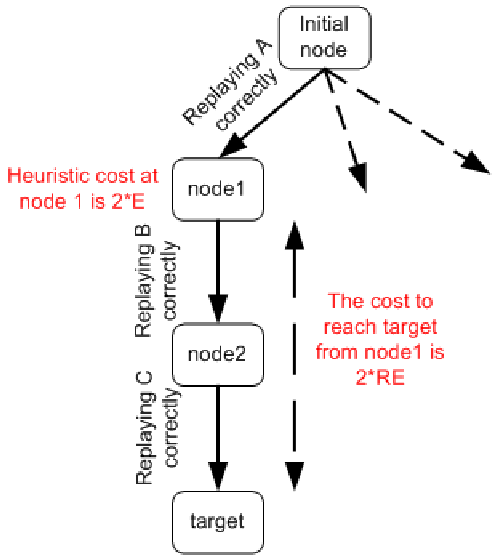


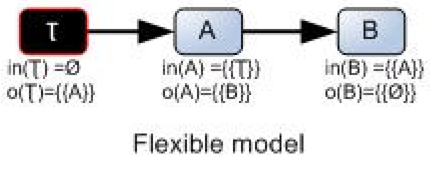
Figure 6.1: A partial search tree generated during replaying trace A-B-C

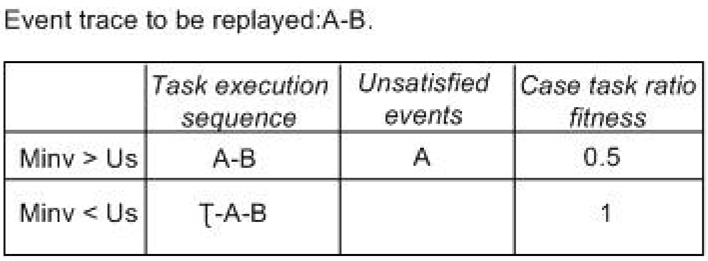
The cost *E* must be bigger than zero. If *E* equal to zero*,* the estimate cost $h(n)$ of each explored node n except the target node in the search tree will be always equal to zero (since $h(n)$=*number of events left in the case*E*). This should not be allowed in A* algorithm. Therefore, we must have $E > 0$ **.**

2) $\underline{0 < Minv \wedge n * Minv < Us},$ *where n is the maximum possible number of invisible tasks executed consecutively to enable execution of a real task in a flexible model.*

Firstly, we observe that the cost of executing an invisible task *(Minv)* must be smaller than the cost of replaying an event incorrectly *(Us).* Consider an example in which we replay the trace *A-B* in a flexible model containing one starting invisible task as shown in Figure 6.2, provided that $Minv > Us$.

During replaying trace *A-B,* the expected task executing sequence should be $\tau - A - B$ with *case task ratio fitness* value of 1. However, the resulting executing sequence is *A-B* with *case task ratio fitness value* of 0.5. The replay algorithm chooses to execute task *A* unsuccessfully rather than executing invisible task τ since cost *Us* is smaller than cost *Minv*. This makes the event *A* unsatisfied. Therefore, to enforce the execution of invisible task τ over execution of task *A,* the cost *Minv* must be smaller than the cost *Us.*

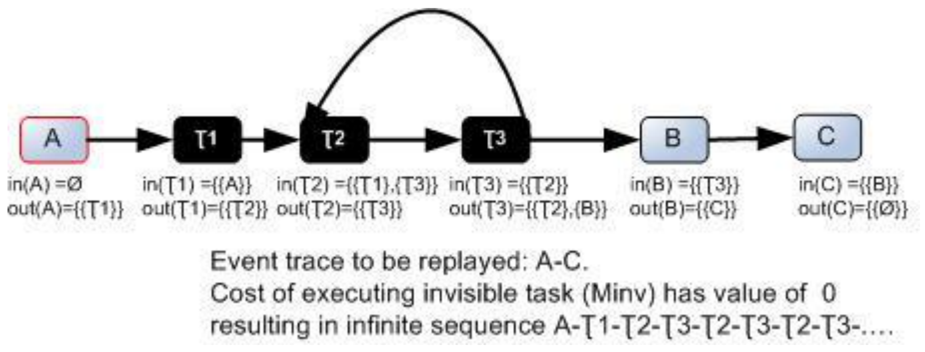



Event trace to be replayed:A-B.

	Task execution sequence	*Unsatisfied events*	*Case task ratio fitness*
Minv > Us	A-B	A	0.5
Minv < Us	Ţ-A-B		1

Figure 6.2: Event trace A-B is replayed in the flexible model with Minv bigger than Us

Secondly, the cost *Minv* must be bigger than zero. If *Minv* is equal to zero, this could lead to an unexpected situation in which the replay algorithm chooses to execute a loop of consecutive invisible tasks indefinitely. As an example for this problem, we replay the trace *A-C* in a flexible model as illustrated in Figure 6.3. After sequence $A - \tau 1 - \tau 2 - \tau 3$ is executed successfully, replay algorithm chooses to execute $\tau 2$ rather than executing *C* unsuccessfully or *moving on model only* with B. This is because the cost of replaying an invisible task is zero which is always smaller than the cost of replaying an event incorrectly and the cost of executing a real task.



Event trace to be replayed: A-C.
Cost of executing invisible task (Minv) has value of 0
resulting in infinite sequence A-Ţ1-Ţ2-Ţ3-Ţ2-Ţ3-Ţ2-Ţ3-....

Figure 6.3: Event trace A-C is replayed with Minv value of zero

Therefore, the replay algorithm executes an infinite sequence of invisible tasks indefinitely (i.e. $A - \tau 1 - \tau 2 - \tau 3 - \tau 2 - \tau 3 - \tau 2 - \ldots$). This situation must be avoided. If *Minv* is larger than zero, at some point along the executing sequence, the total cost to execute a number of consecutive invisible tasks becomes larger than the cost of replaying an unsatisfied event and cost of moving on a real task. This forces the replay algorithm to quit executing loop of consecutive invisible tasks. Hence*, Minv* must be bigger than zero (i.e. *Minv >0*)

There exists another serious problem when the cost *Minv* is bigger than zero. Assume that we replay event trace *B-C* in a flexible model (Figure 6.4) with *Minv* value of 2 and *Us* value of 5. The expected *case task ratio fitness* value is 1 achieved by executing the sequence $\tau 1-\tau 2-\tau 3-B-C$ successfully. However, the cost to execute three consecutive invisible tasks ($\tau 1, \tau 2$ and $\tau 3$) is larger than the cost of replaying event *B* incorrectly (i.e. $3*2>5$). Consequently, the replay algorithm chooses to execute *B* unsuccessfully rather than executing sequence of invisible tasks. This creates an unexpected executing sequence $B-C$ with *B* is an unsatisfied event. Hence the obtained *case task ratio fitness* has value of 0.5 which is different from the expectation. In order to replay event *B* correctly (i.e. task *B* is fired successfully), the cost of replaying an unsatisfied event must be bigger than the cost of executing three invisible tasks consecutively (i.e. $Us>3*Minv$).

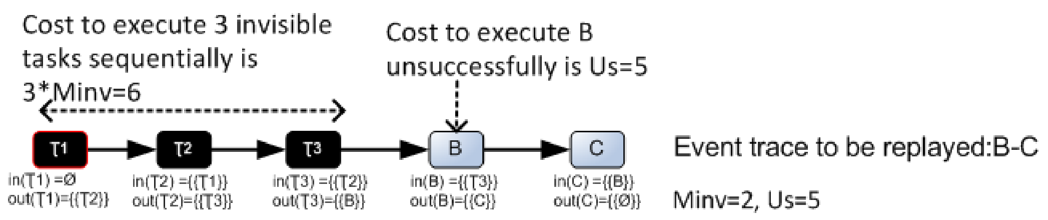
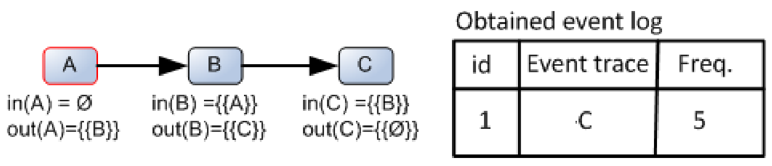

Cost to execute 3 invisible tasks sequentially is 3*Minv=6
Cost to execute B unsuccessfully is Us=5
T1
T2
T3
B
C
Event trace to be replayed:B-C
in(T1) =Ø out(T1)={{T2}}
in(T2) ={{T1}} out(T2)={{T3}}
in(T3) ={{T2}} out(T3)={{B}}
in(B) ={{T3}} out(B)={{C}}
in(C) ={{B}} out(C)={{Ø}}
Minv=2, Us=5

Figure 6.4: Event trace B-C is replayed with Minv value of 2 and Us value of 5

We observe that this serious problem might occur when we need to execute a complex long sequence of invisible tasks to enable execution of a real task. Let *n* be the maximum possible number of invisible tasks executed successively to enable a real task in an arbitrary flexible model. The described problem can be avoided by setting the cost of replaying an unsatisfied event bigger than the cost of executing *n* sequential invisible tasks (i.e. *Us> n*Minv*). *Furthermore,* it is essential to construct an algorithm to find the value of *n* before the replay can start. However, constructing such algorithm is difficult and beyond the scope of this thesis.

3) $Mr < Us$

In reality, there could always be the case that some activities were executed but not logged due to some unknown reasons (e.g. technical problems or human mistakes). This constraint makes it possible to identify those missing activities in the log during replay. Consider an example in which we have a valid process model described in form of flexible model (Figure 6.5). This process model conforms perfectly to the reality. An event log is obtained by simulating the process model. Due to some reasons, executions of task *A* and *B* are not logged. Hence event *A* and *B* are missing and only C was recorded in the log as illustrated in Figure 6.5.



Obtained event log

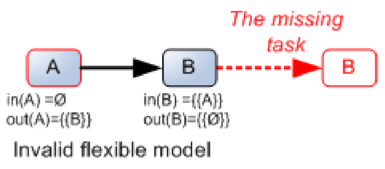
id	Event trace	Freq.
1	C	5

Figure 6.5: An event log is obtained by simulating the flexible model

We replay the obtained event log in the flexible model. In the case that cost of replaying an event improperly (*Us*) is smaller than cost of executing a real task (*Mr*), task *C* will be executed unsuccessfully (i.e. event *C* is unsatisfied). This leads to the *task ratio fitness* value of 0.5 which is supposed to be 1 in reality. Since *A* and *B* were actually executed in the process and hence C was fired successfully. Therefore, in order to fire *C* properly during the replay, task *A* and *B* must be executed in prior. Task *A* and *B* are regarded as missing events during replay. This can only be guaranteed if replay algorithm performs the step *moving on model only with real task* two times which means the cost *Us* should be bigger than two times of cost *Mr* (i.e. $2*Mr < Us$). In a more complex situation, we might need to execute more real tasks to enable execution of a certain task. Let *n* be the minimum number of consecutive real tasks that might need to be executed to enable a certain task. The cost *Us* should be bigger than *n* times of cost *Mr* (i.e. $n*Mr < Us$). Generally, the value of *n* is often decided by the users before the replay can start.

4) <u>*Rem > Us+Re*</u>

Cost of removing an event should be bigger than sum of cost of replaying an unsatisfied event and cost of executing a real task only. Consider an example in which we have an invalid flexible model and a valid event log as shown in Figure 6.6. The invalid model does not conform perfectly to reality such as an execution of task *B* which actually happens in reality is missing in the model. The valid log is obtained from reality and contains only one trace *A-B-B* with frequency of 100.



Valid event log obtained from reality

id	*Trace*	*Freq.*
1	*A-B-B*	*100*

Figure 6.6: Valid log is replayed in an invalid flexible model

The expected executing sequence after replaying a case is *A-B-B* in which the last event *B* is unsatisfied (i.e. *task ratio fitness* value is 0.5). Assume that we obtain a partial search tree during replaying *A-B-B as shown in Figure 6.7*. *Node1* is reached by replaying event *A* and first *B* correctly. Let *n* be the number of events replayed so far and *k* be the number of events left in the case at *node1*.

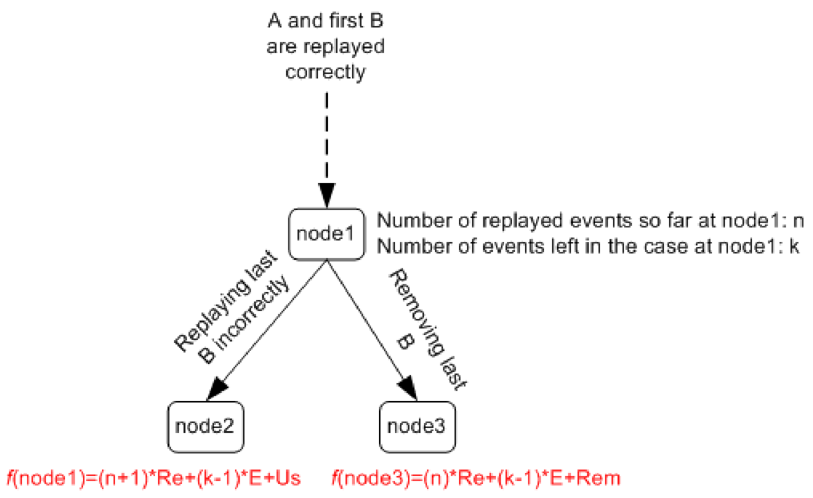

A and first B are replayed correctly
node1
Number of replayed events so far at node1: n
Number of events left in the case at node1: k
Replaying last B incorrectly
Removing last B
node2
node3
f(node1)=(n+1)*Re+(k-1)*E+Us
f(node3)=(n)*Re+(k-1)*E+Rem

Figure 6.7: A partial search tree obtained during replaying trace A-B-B

Node2 is created by replaying the last *B* incorrectly and *node3* is created by removing last *B* from the log. The evaluation cost of *node2* is $f(node2) = (n+1)*\mathrm{Re} + (k-1)*E + Us$ and the evaluation cost of *node3* is $f(node3) = (n) * \text{Re} + (k-1) * E + Rem.$ To enforce the replay algorithm to execute last event *B* unsuccessfully, the evaluation cost of *node2* must be smaller than the cost of *node3*. That is $(n+1) * \text{Re} + (k-1) * E + Us < (n) * \text{Re} + (k-1) * E + Rem.$ This implies *Rem > Us+Re.*

If the cost of removing an event is smaller than the sum of cost of replaying an unsatisfied event and cost of executing a real task only, the resulting executing sequence is *A-B* in which the last event *B* is not replayed (i.e. it is removed from the log). This leads to the *task ratio fitness* value of 1 which is not correct since the model is invalid. Therefore, to avoid such situation, the cost *Rem* should be bigger than the sum of *Us* and *Re*.

6.3 Case study

In this section, we describe a case study which involves a real life process as shown in Figure 6.8. This process deals with a complaint handling process of a town hall in the Netherlands. We also have a real life log recorded from this process.

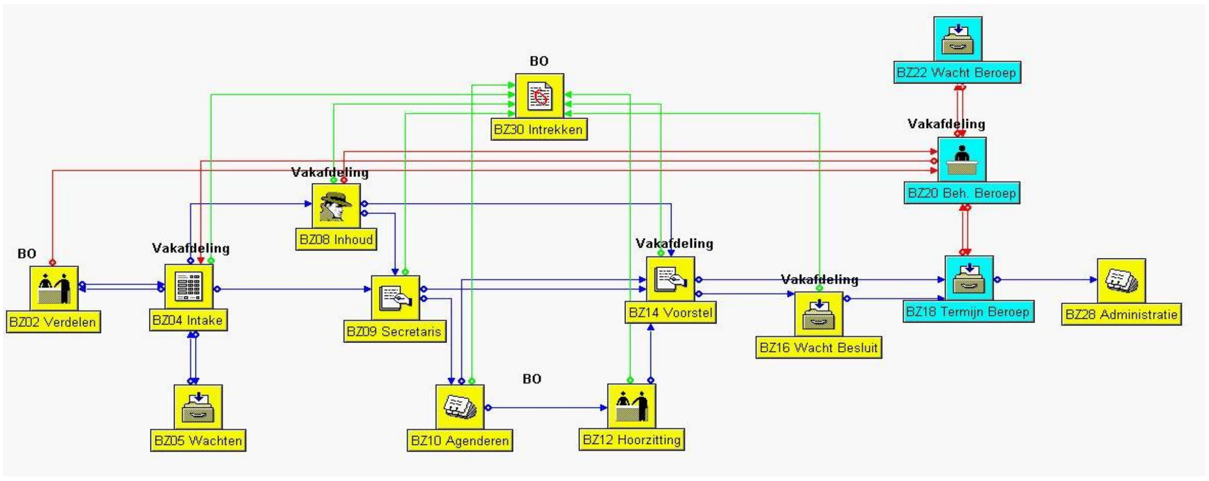


Figure 6.8: the original description of the complaint handling process

Using process mining discovery technique on this log, we come up with process model expressed in form of flexible model as depicted in Figure C.1 (Appendix C). Note that all split and join in flexible model are of type exclusive choice (XOR-split) and simple merge (XOR-join). We are going to replay the log in this flexible model to evaluate the task ratio fitness value, provided that all cost parameters are configured based on the constraints presented in Section 6.2. The idea is to replay each case from the log and compare the result obtained from replaying with value given by our intuition. However, we don't replay all cases. Instead, we select the first three cases for analyzing. The configuration of cost parameters is given in Table 6.1.

Cost parameter	Value
Cost of replaying an event correctly(RE)	*200*
Cost of an event left in case(E)	*200*
Cost of moving on a invisible task(Minv)	*1*
Cost of moving on a real task(Mr)	*250*
Cost of removing an event from log(Rem)	*550*
Cost of an unsatisfied event(Us)	*300*

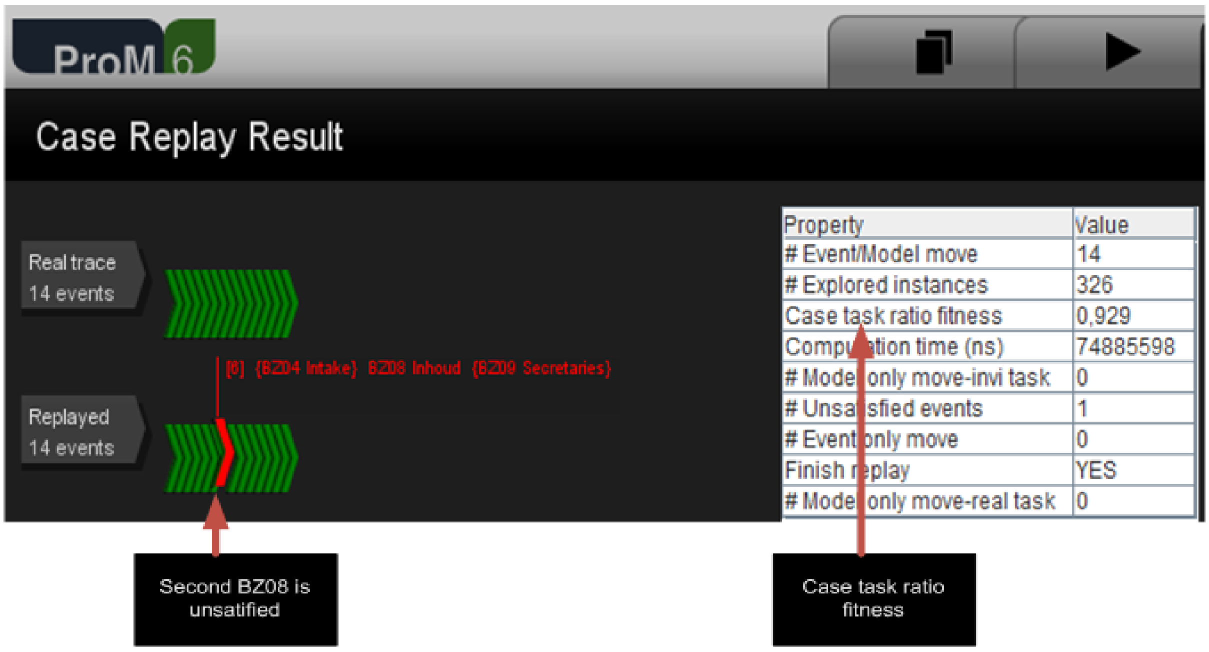
Table 6.1: Cost parameters setting

Using the extended flexible model replay algorithm (i.e. Extended Cost based A* heuristic log replay), we obtain the following result for each replayed event case.

1. *Event case 1.*
   Table 6.2 describes the detail of the first event case. From our intuition, there should be only one unsatisfied event, *BZ04 Inhoud*, after replaying the case. This is because the enablement of *BZ08 Indhoud* needs *BZ04 Intake* to be executed in prior. However*, BZ04 Intake* was already fired to execute the first *BZ08 Inhoud*. Therefore, the second event *BZ08 Inhoud* becomes unsatisfied and the *case task ratio fitness* is expected to be 0.929. Obviously, the replaying result (Figure 6.9) matches our intuition in which the second *BZ08 Inhoud* is actually the only one unsatisfied event and the *case task ratio fitness* is 0.929.

Case ID	Event trace(14 events)	Expected executing sequence	Expected f_c^{rat}
04301ESWM183483	*Case start - BZ02 Verdelen - BZ04 Intake - BZ08 Inhoud - BZ09 Secretaris - BZ10 Agenderen - BZ08 Inhoud - BZ09 Secretaris - BZ10 Agenderen - BZ12 Hoorzitting - BZ14 Voorstel - BZ16 Wacht Besluit - BZ18 Termijn Beroep - BZ28 Administratie*	*Case start - BZ02 Verdelen - BZ04 Intake - BZ08 Inhoud - BZ09 Secretaris - BZ10 Agenderen -* ***BZ08 Inhoud*** *- BZ09 Secretaris - BZ10 Agenderen - BZ12 Hoorzitting - BZ14 Voorstel - BZ16 Wacht Besluit - BZ18 Termijn Beroep - BZ28 Administratie*	*0.929*

Table 6.2: Description of first case together with expected executing sequence and case task ratio fitness



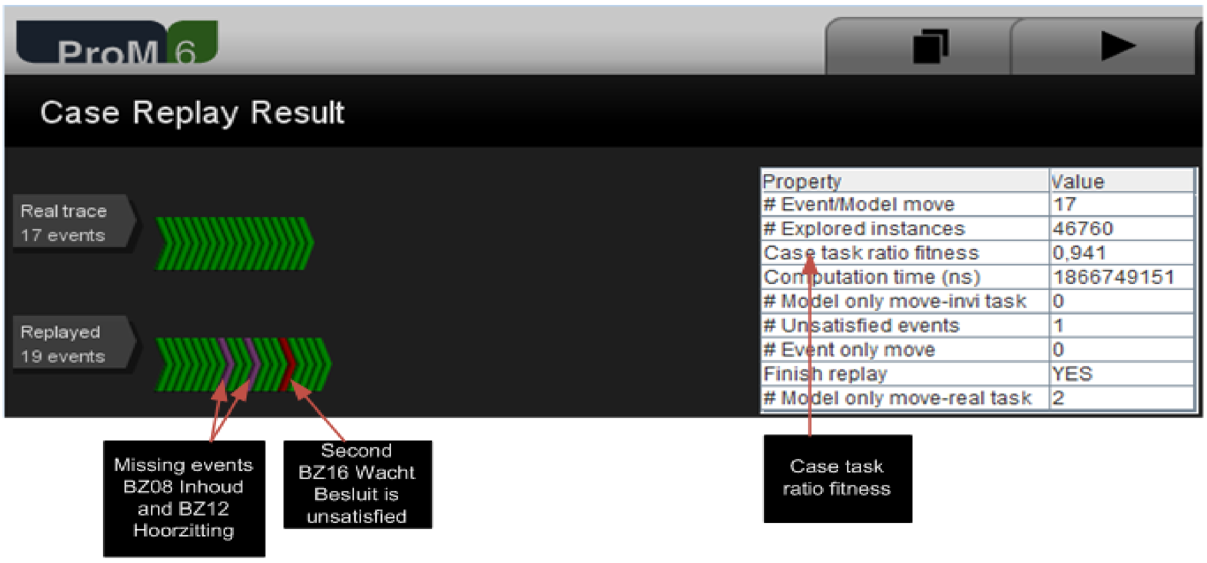
Property	Value
# Event/Model move	14
# Explored instances	326
Case task ratio fitness	0,929
Computation time (ns)	74885598
# Model only move-invi task	0
# Unsatisfied events	1
# Event only move	0
Finish replay	YES
# Model only move-real task	0

Figure 6.9: Result of replaying first case

2. <u>*Event case 2*</u>.
   The detail of second case is described in Table 6.3. We expect the executing sequence after replaying the case to contain only one unsatisfied event, *BZ16 Wacht Besluit*. Furthermore, we expect two missing events (*BZ08 Inhoud* and *BZ12 Hoorzitting*) are identified in the executing sequence**.** The expected *case task ratio fitness* should be 0.942**.** Figure 6.10 shows result of replaying the second event case. We observe that result fits our intuition in which the *case task ratio fitness* has value of 0.941 and two missing events (*BZ08 Inhoud* and *BZ12 Hoorzitting)* are identified.

Case ID	**Event trace(17 events)**	**Expected executing sequence**	**Expected** f_c^{rat}
04349ESWM205386	*Case start-BZ02 Verdelen-BZ04 Intake-BZ02 Verdelen-BZ04 Intake-BZ02 Verdelen-BZ04 Intake-BZ09 Secretaris-BZ10 Agenderen-BZ14 Voorstel-BZ16 Wacht Besluit-BZ18 Termijn Beroep-BZ16 Wacht Besluit-BZ18 Termijn Beroep-BZ20 Beh. Beroep-BZ18 Termijn Beroep-BZ28 Administratie*	*Case start-BZ02 Verdelen-BZ04 Intake-BZ02 Verdelen-BZ04 Intake-BZ02 Verdelen-BZ04 Intake-**BZ08 Inhoud**-BZ09 Secretaris-BZ10 Agenderen-**BZ12 Hoorzitting** -BZ14 Voorstel-BZ16 Wacht Besluit-BZ18 Termijn Beroep-**BZ16 Wacht Besluit**-BZ18 Termijn Beroep-BZ20 Beh. Beroep-BZ18 Termijn Beroep-BZ28 Administratie*	*0.942*

Table 6.3: Description of second case together with expected executing sequence and case task ratio fitness

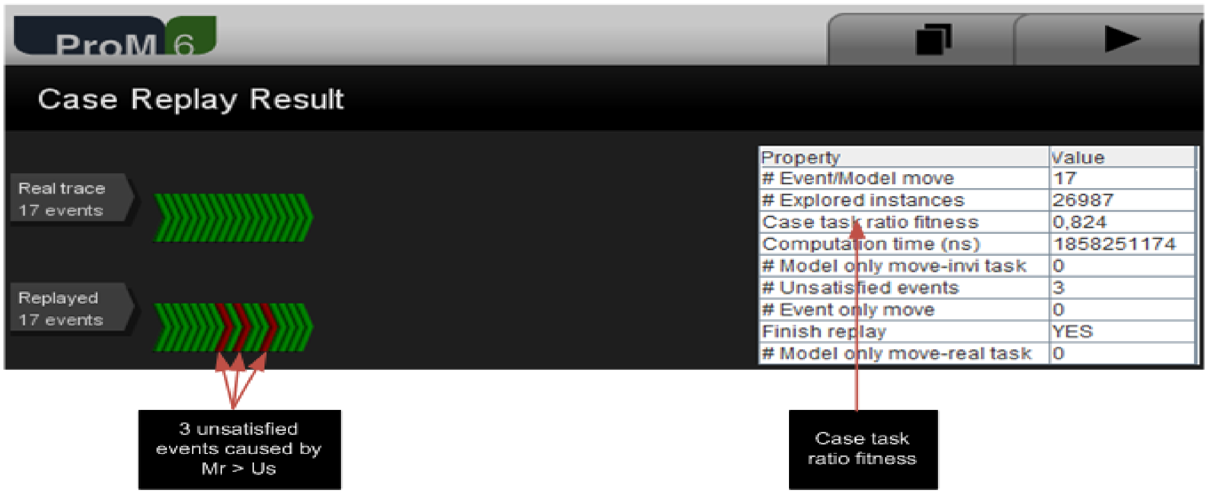

ProM 6
Case Replay Result
Real trace
17 events
Replayed
19 events
Missing events BZ08 Inhoud and BZ12 Hoorzitting
Second BZ16 Wacht Besluit is unsatisfied
Case task ratio fitness

Property	Value
# Event/Model move	17
# Explored instances	46760
Case task ratio fitness	0,941
Computation time (ns)	1866749151
# Model only move-invi task	0
# Unsatisfied events	1
# Event only move	0
Finish replay	YES
# Model only move-real task	2

Figure 6.10: Result of replaying second case

In order to identify the missing events during replay, cost of executing a real task must be smaller than cost of an unsatisfied event. If this condition does not hold, some events will become unsatisfied while they are supposed to be satisfied in reality. We have shown that with the cost parameters given in Table 6.1, missing events were captured during relaying the second case.

We now increase the cost of executing a real task such that it is bigger than cost of an unsatisfied event (e.g. *Mr*=301 and *Us*=300).The executing sequence after replaying the second case is then expected to have three unsatisfied events, which are *BZ09 Secretaris, BZ14 Voorstel* and *BZ16 Wacth Besluiy.* And the *case task ratio fitness* value is expected to be 0.824. Indeed, the replay result (Figure 6.110) shows that there exist three unsatisfied evens and the *case task ratio* is 0.824 as expected.



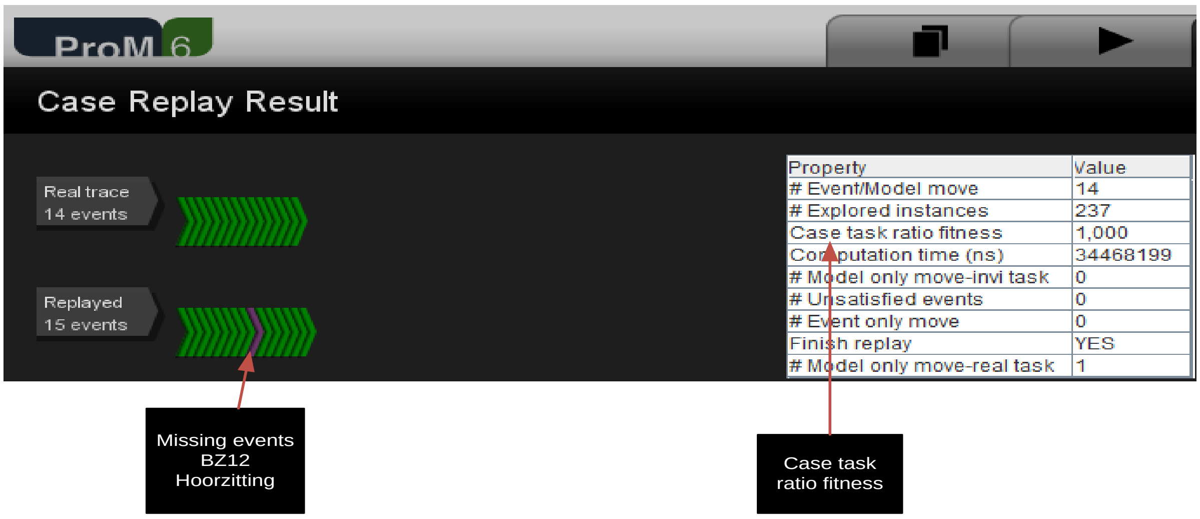
Property	Value
# Event/Model move	17
# Explored instances	26987
Case task ratio fitness	0,824
Computation time (ns)	1858251174
# Model only move-invi task	0
# Unsatisfied events	3
# Event only move	0
Finish replay	YES
# Model only move-real task	0

Figure 6.11: Result of replaying the second case with cost Mr is bigger than cost Us

3. <u>*Event case 3.*</u>
   The detail of the third case is shown in Table 6.3. From our intuition, we expect that the missing event *BZ12 Hoorzitting* will be captured and there is no unsatisfied event. The expected *case task ratio fitness* is 1. The replay result depicted in Figure 6.12 proves that our assumption is true. The missing event *BZ12 Hoorzitting* is actually executed and the *case task ratio fitness* has value of 1 as expected.

Case ID	Event trace(14 events)	Expected executing sequence	Expected f_c^{rat}
05011ESWM217434	*Case start-BZ02 Verdelen-BZ04 Intake-BZ02 Verdelen-BZ04 Intake-BZ08 Inhoud-BZ09 Secretaris-BZ10 Agenderen-BZ14 Voorstel-BZ16 Wacht Besluit-BZ18 Termijn Beroep-BZ20 Beh. Beroep-BZ18 Termijn Beroep-BZ28 Administratie*	*Case start-BZ02 Verdelen-BZ04 Intake-BZ02 Verdelen-BZ04 Intake-BZ08 Inhoud-BZ09 Secretaris-BZ10 Agenderen-* ***BZ12 Hoorzitting*** *-BZ14 Voorstel-BZ16 Wacht Besluit-BZ18 Termijn Beroep-BZ20 Beh. Beroep-BZ18 Termijn Beroep-BZ28 Administratie*	*1*

Table 6.4: Description of third case together with expected executing sequence and case task ratio fitness



Property	Value
# Event/Model move	14
# Explored instances	237
Case task ratio fitness	1,000
Computation time (ns)	34468199
# Model only move-invi task	0
# Unsatisfied events	0
# Event only move	0
Finish replay	YES
# Model only move-real task	1

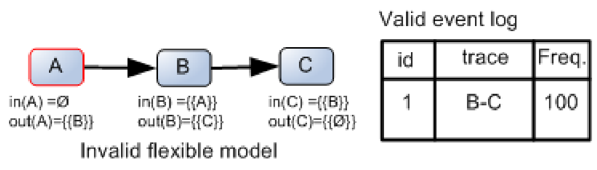
Figure 6.12: Result of replaying third case

6.4 Conclusion

In this chapter, we define some constraints for configuring cost parameters to obtain an optimal task ratio fitness value. We also illustrate a case study in which a real log is replayed in a mined process model. In this case study, we set up the cost parameters according to the defined constraints. We obtain results of replaying that quite match our intuitions. However, there are still several existing problems which can only be solved by performing further analysis.

Firstly, we stated that the cost of unsatisfied event must be bigger than *n* times of the cost of moving on an invisible task. The value of *n* is defined as the maximum number of invisible tasks executed consecutively to enable firing a certain task in an arbitrary flexible model. Detecting the value of *n* is a problem of interest. It is desirable to construct an optimal algorithm which is able to identify the value of *n*. However, this is difficult and takes lots of efforts.

Secondly, there is a serious problem related to the constraints involving cost of executing a real task (*Minv*) and cost of an unsatisfied event (*U*s). Consider an example in which we have an invalid flexible model and a valid event log as shown in Figure 6.13. This invalid model does not conform perfectly to reality such as the execution of task *A* does not actually happen in reality, whereas the valid event log is recorded from reality and contains one event trace *B-C* with frequency value of 100.



id	trace	Freq.
1	B-C	100

Figure 6.13: Valid event log is replayed in an invalid flexible model

We replay this event log in the flexible model. According to the constraint (i.e. *Minv* < *Us*), the resulting executing sequence is *A-B-C* with *task ratio fitness* value of *1.* This result seems not to be correct since the model is assumed to be invalid (i.e. execution of A does not occur in reality) Therefore, we might need to extend the current *case task ratio fitness* such that it involves the number of "*move on model only with real task"* steps during replay.

Finally, a unique activity in a process model may be logged unexpectedly more than one in event logs. Thus, the step "*move on log only*" during replay will remove the redundant events occurring in the log. As we observe that the step "*move on log only*" is mainly decided by the cost of removing an event. Therefore, further analysis on the relation between the cost of removing an event and other cost parameters needs to be taken to capture redundant events during replay.

Chapter 7

Conclusions

In this thesis, we investigated an approach to make log replay applicable on a variety number of process modeling languages. The occurred problem is the limitation of log replay techniques which work only on some specific process modeling languages. Instead of developing a new log replay for each existing process modeling languages, in Chapter 4, we proposed a unification approach that abstracts away the need of having specific process modeling languages. The idea of this approach is to convert any process models, regardless of their modeling language, into a generic process format and deploy a replay technique for that generic process format. We chose the canonical process format (CPF) as a general process format and extended existing conversions from various process modeling language to CPF such that they preserve necessary information needed for replay log analysis.

After tackling the dependency on specific process modeling languages problem, the next problem we tackled is the problem of replaying logs on the selected generic process format. In Chapter 5, we proposed a log replay technique on CPF by extending the A*-based log replay technique that currently works for Flexible models. This solution requires a transformation from CPF to an equivalent flexible model. Therefore, we also proposed a mapping rule for converting from CPF to flexible model in advance in Chapter 4.

Finally, in Chapter 6, we took an analysis on cost parameters required in the extended flexible model replay technique. We came up with several constraints between cost parameters to obtain the optimal replay result (i.e. optimal task ratio fitness value). Furthermore, we performed a case study in which a real life log was replayed in a mined process model. In this case study, we used the extended flexible model replay technique and set up the cost parameters based on the defined constraints. We obtained the replay result which matched well our pre-defined intuition.

With the proposed approach, we believe that log replay is applicable in any existing process modeling languages as long as they can be expressed in an equivalent canonical process format.

7.1 Limitations and future works

There are several existing issues which should be noticed. The first issue is related to conversion from Yawl to CPF in which multiple-instances characteristic cannot be preserved in CPF. This is because canonical process format is not able to capture multiple-instances characteristic. Moreover, the current flexible model and flexible replay technique were not invented to deal with multiple-instances. As we chose to enable the replay on CPF indirectly via flexible model, loosing information involving multiple-instances potentially leads to an unexpected replay result. Thus, it is essential to extend CPF, flexible mode and flexible model-based replay technique such that multiple-instances characteristic can be identified.

The second issue is concerning the cost parameters required in the flexible model replay technique. We have already mentioned in Section 6.4 it is necessary to construct an algorithm to indentify the longest number of invisible tasks executed consecutively in an arbitrary flexible model. Moreover, it is also essential to perform further analysis on the cost parameters such that redundant events in event logs can be identified.

Finally, in this thesis, we only introduced mapping rules for converting to CPF of four process modeling languages such as Petri net, Yawl, EPC and Flexible model. Thus, it is desirable to design mapping rules of other process modeling languages (e.g. BPMN, Protos, Staffware or WS-BPEL).

References

1. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. Distributed and Parallel Databases, 14(1):5-51, 3002.

2. W. Reisig and G. Rozenberg, editors. Lectures on Petri Nets : Basic Models, volume 1491of Lecture Notes in Computer Sceince. Springer-Verlag, Berlin, 1998.34.

3. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245-275, 2005.

4. W.M.P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639{650, 1999.

5. C.W. Gunther. *Process Mining in Flexible Environments*. PhD thesis, Department of Technology Management, Technical University Eindhoven, 2009.

6. Marcello La Rosa, Hajo A. Reijers, Wil M.P. van der Aalst, Remco M. Dijkmanb, Jan Mendling, Marlon Dumas, Luciano Garcıa-Banuelos. *APROMORE: An Advanced Process Model Repository.*

7. Adriansyah, B.F. van Dongen, and W.M.P van der Aalst. Towards robust conformance checking. In *(To appear) Proceedings of the 6th International Workshop on Business Process Intelligence (BPI 2010)*, 2010.

8. W.M.P. van der Aalst, B.F. van Dongen, C. G¨unther, A. Rozinat, H.M.W. Verbeek, and A.J.M.M. Weijters. ProM: The Process Mining Toolkit, volume 489 of CEUR-WS.org. Ulm, Germany, 2009.

9. B.F. van Dongen and W.M.P. van der Aalst. A Meta Model for Process Mining Data. In Conference on Advanced Information System Engineering, volume 161, Porto, Portugal, 2005.

10. J.C.A.M. Buijs. *Mapping Data Sources to XES in a Generic Way*. Master thesis, 2010.

11. Process mining main website. http://www.processmining.org/.

12. A.Rozinat. *Process Mining: Conformance and Extension*. PhD thesis. Pages 64-134. Eindhoven University of Technology, Eindhoven, 2010.

13. A. Adriansyah, B.F. van Dongen, and W.M.P. van der Aalst. *Reliable Log Replay for Conformance Checking*, 2010.

14. P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths in Graphs. *IEEE Trans. Syst. Sci. and Cybernetics*, SSC-4(2):100{107, 1968.

Appendix A

Meta-model of canonical process format

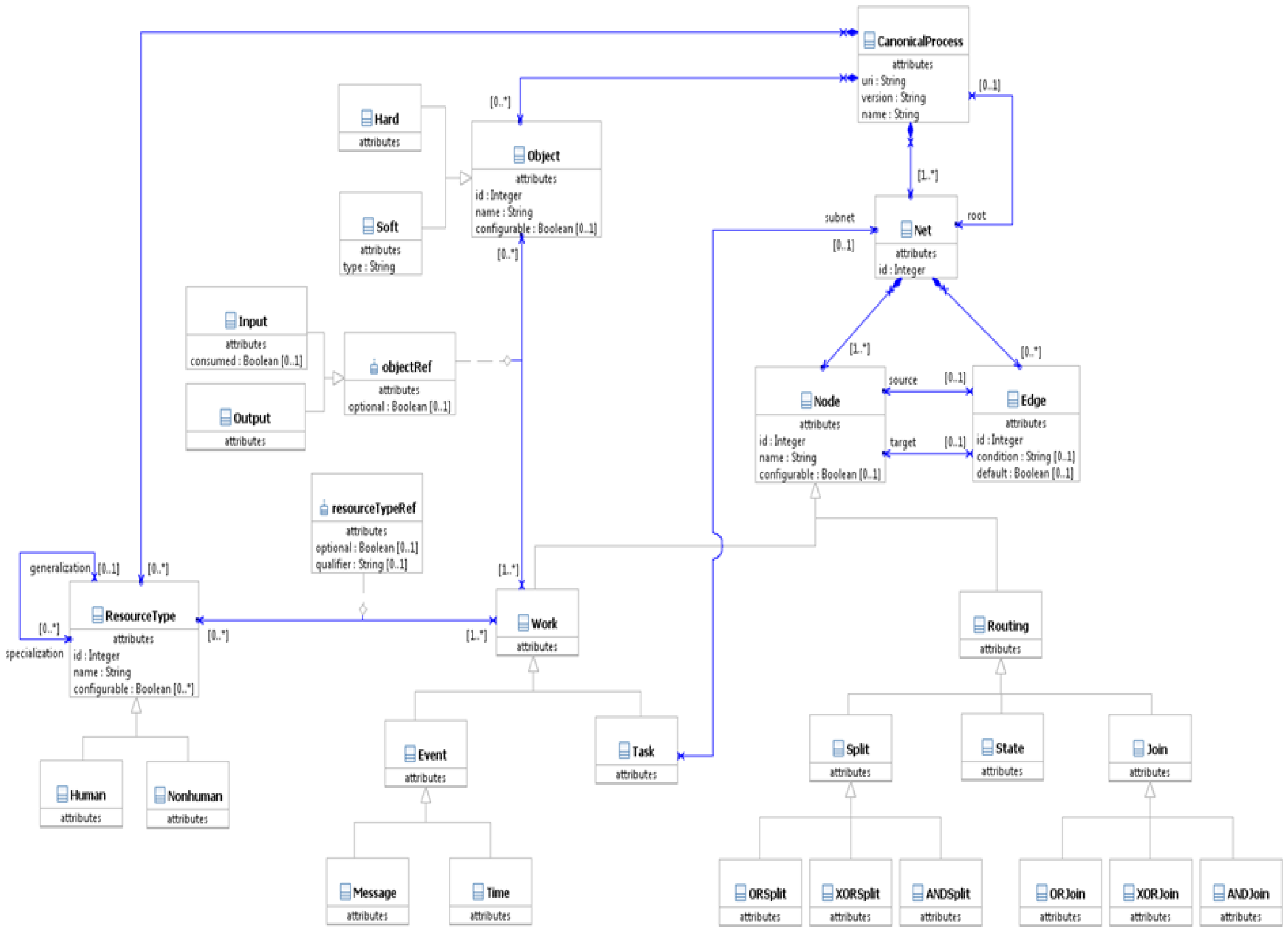


Figure A.1: Meta-model of Canonical Process Format

Appendix B

Tables of mapping rules of Petri net, Flexible model, Yawl and EPC

Table B.1: Mapping from Petri net elements to Canonical elements

Petri net elements	**Canonical elements**
Net	*Net*
Arc	*Edge*
Transition	*Task*
Sees - input place, *Sese - output place,* *Sese - normal place.*	*Event*
	Message Event
	Timer Event
Transition's AND-split	*ANDSplit*
	ORSplit
	XORSplit
Transition's AND-join	*ANDJoin*
	ORJoin
	XORJoin
Non-sese (input/output) Place	*State*

Table B.2: Mapping from Flexible elements to Canonical elements

Flexible elements	**Canonical elements**
Net	*Net*
Arc	*Edge*
Task(including invisible task)	*Task*
	Event
	Message Event
	Timer Event
Task's AND-split	*ANDSplit*
Task's OR-split	*ORSplit*
Task's XOR-split	*XORSplit*
Task's AND-join	*ANDJoin*
Task's OR-join	*ORJoin*
Task's XOR-join	*XORJoin*
	State

Table B.3: Mapping from Yawl elements to Canonical elements

Yawl elements	**Canonical elements**
Net	*Net*
Flow	*Edge*
Task (Atomic)	*Task*
Task(Composite)	*Sub-net*
Sese - input condition, *Sese - output condition,* *Sese - normal condition*	*Event*
	Message Event
	Timer Event
Task's AND-split	*ANDSplit*
Task's OR-split	*ORSplit*
Task's XOR-split	*XORSplit*
Task's AND-join	*ANDJoin*
Task's OR-join	*ORJoin*
Task's XOR-join	*XORJoin*
Non-sese (input/output) Condition	*State*

Table B.4: Mapping from EPC elements to Canonical elements

EPC elements	**Canonical elements**
Net	*Net*
Arc, Events subsequent to (X)OR-split connector	*Edge*
Function	*Task*
Event not subsequent to (X)OR-split connector	*Event*
	Message Event
	Timer Event
AND-split connector	*ANDSplit*
OR-split connector	*ORSplit*
XOR-split connector	*XORSplit*
AND-join connector	*ANDJoin*
OR-join connector	*ORJoin*
XOR-join connector	*XORJoin*
	State

Table B.5: Mapping from Canonical elements to elements of other process models

Canonical elements	**Petri net**	**Flexible elements**	**Yawl**	**EPC**
Net	*Net*	*Net*	*Net*	*Net*
Edge	*Arc*	*Arc*	*Flow*	*Arc*
Task	*Transition*	*Task*	*Task*	*Function*
Event	*Place*		*Condition*	*Event*
Message Event				
Timer Event				
ANDSplit	*Task's AND-split*	*Task's AND-split*	*Task's AND-split*	*AND-split connector*
ORSplit		*Task's OR-split*	*Task's OR-split*	*OR-split connector*
XORSplit		*Task's XOR-split*	*Task's XOR-split*	*XOR-split connector*
ANDJoin	*Task's AND-join*	*Task's AND-join*	*Task's AND-join*	*AND-join connector*
ORJoin		*Task's OR-join*	*Task's OR-join*	*OR-join connector*
XORJoin		*Task's XOR-join*	*Task's XOR-join*	*XOR-join connector*
State	*Place*		*Condition*	

Table B.6: A list of different types of Yawl task

Atomic Task	MI [5]Atomic Task	Composite Task	MI Composite Task
Tasks don't have routing:	***Tasks don't have routing:***	***Tasks don't have routing:***	***Tasks don't have routing:***
Atomic task	*MI Atomic task*	*Composite task*	*MI Composite task*
Tasks have only join pattern:	***Tasks have only join pattern:***	***Tasks have only join pattern:***	***Tasks have only join pattern:***
ANDJoin-Atomic task	*ANDJoin- MI Atomic task*	*ANDJoin-Composite task*	*ANDJoin- MI Composite task*
XORJoin-Atomic task	*XORJoin-MI Atomic task*	*XORJoin-Composite task*	*XORJoin-MI Composite task*
ORJoin-Atomic task	*ORJoin-MI Atomic task*	*ORJoin-Composite task*	*ORJoin-MI Composite task*
Tasks have only split pattern:	***Tasks have split pattern:***	***Tasks have split pattern:***	***Tasks have split pattern:***
ANDSplit-Atomic task	*ANDSplit-MI Atomic task*	*ANDSplit-Composite task*	*ANDSplit-MI Composite task*
XORSplit-Atomic task	*XORSplit-MI Atomic task*	*XORSplit-Composite task*	*XORSplit-MI Composite task*
ORSplit-Atomic task	*ORSplit-MI Atomic task*	*ORSplit-Composite task*	*ORSplit-MI Composite task*
Tasks have join-split pattern:	***Tasks have join-split pattern:***	***Tasks have join-split pattern:***	***Tasks have join-split pattern:***
ANDJoin-ANDSplit Atomic task	*ANDJoin-ANDSplit MI Atomic task*	*ANDJoin-ANDSplit Composite task*	*ANDJoin-ANDSplit MI Composite task*
ANDJoin-XORSplit Atomic task	*ANDJoin-XORSplit MI Atomic task*	*ANDJoin-XORSplit Composite task*	*ANDJoin-XORSplit MI Composite task*
ANDJoin-ORSplit Atomic task	*ANDJoin-ORSplit MI Atomic task*	*ANDJoin-ORSplit Composite task*	*ANDJoin-ORSplit MI Composite task*
XORJoin-ANDSplit Atomic task	*XORJoin-ANDSplit MI Atomic task*	*XORJoin-ANDSplit Composite task*	*XORJoin-ANDSplit MI Composite task*
XORJoin-XORSplit Atomic task	*XORJoin-XORSplit MI Atomic task*	*XORJoin-XORSplit Composite task*	*XORJoin-XORSplit MI Composite task*
XORJoin-ORSplit Atomic task	*XORJoin-ORSplit MI Atomic task*	*XORJoin-ORSplit Composite task*	*XORJoin-ORSplit MI Composite task*
ORJoin-ANDSplit Atomic task	*ORJoin-ANDSplit MI Atomic task*	*ORJoin-ANDSplit Composite task*	*ORJoin-ANDSplit MI Composite task*
ORJoin-XORSplit Atomic task	*ORJoin-XORSplit MI Atomic task*	*ORJoin-XORSplit Composite task*	*ORJoin-XORSplit MI Composite task*
ORJoin-ORsplit Atomic task	*ORJoin-ORsplit MI Atomic task*	*ORJoin-ORsplit Composite task*	*ORJoin-ORsplit MI Composite task*

[5] MI: Multiple instances

Appendix C

Mined process model for case study

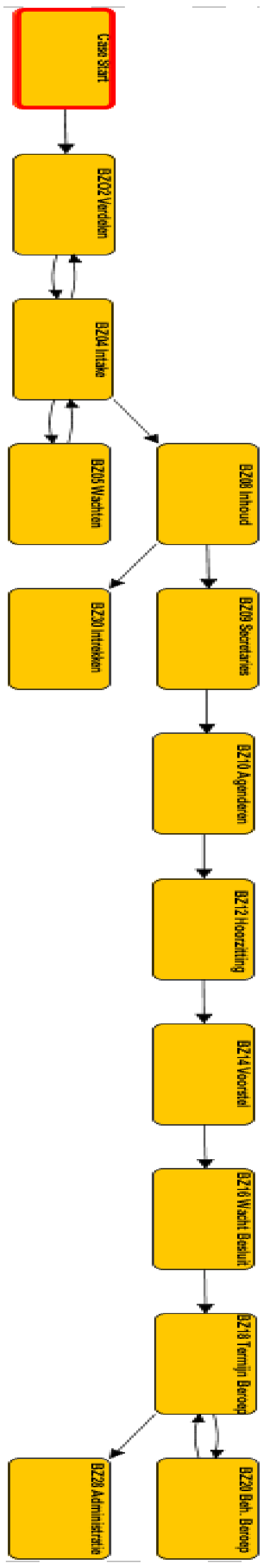


Figure C.1: Flexible model of the complaint handling procedure