Generating realistic logs using a Colored Petri Net simulator

Verberk, Tom M.

Award date:
2022

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Process Analytics

Generating realistic logs using a Colored Petri Net simulator

Master Thesis

Tom Verberk

Supervisors:
prof. dr. ir. Boudewijn van Dongen
ir. Mitchel Brunings
Assessment committee member: dr. ir. Tom Verhoeff

23-08-2022

Abstract

In the process mining field, several techniques have been developed during the last years for the discovery of Petri Net from event logs. One way to test a process discovery technique is to generate an event log by simulating a Petri Net, and then verify that the Petri Net discovered from such a log matches the original one. For this reason, a easy to use tool for generating event logs from Petri Nets becomes vital for the evaluation of process discovery techniques. In this thesis, we present a program which is able to do this, the extension built in CPN-IDE is able to generate logs conforming to the XES Standard from the simulation of a Petri Net. The tool will have the option to add lifecycle-transitions, time as well as resources to the log. An evaluation of the implemented tool is presented in which the generated logs are checked for correctness, this evaluation shows the correctness of the logs.

Contents

Chapter 1

Introduction

Process mining is a research and engineering discipline allowing for the analysis of business processes starting from event logs. The Extensible Event Stream [4] (in the remainder of this thesis called XES) is a well-known and accepted IEEE standard in the process analytics world for generating, storing, exchanging, and analyzing event logs. In this standard, each event in the process corresponds to an event in the log. The events are ordered by case and can be seen as the activities that belong to a specific case. Event logs may store additional information about events such as timestamps, resources, and lifecycle-transitions.

One of the branches of the process mining discipline is the discovery of process models from event logs. The main idea of this branch is to extract the knowledge from a log and generate a model from it. There are several real life logs publicly available, this includes the Business Process Intelligence Challenge (BPIC) logs [2]. However, there are only a limited number of logs and the logs might not have certain attributes that you want the log to have. For this reason, a common approach for testing discovery algorithms is with the use of synthetic logs. Synthetic logs allow the individuals to create a log that has the characteristics that the researchers want to use in their research.

1.1 State of the Art

Several synthetic log generators are already available in the literature. These tools are all based on different modelling languages [6][7][13][10]. Generating a log from the simulation of Colored Petri Nets (CPNs) allows researchers to create specific logs (as the created logs are a one-to-one representation of the simulation). It also allows researchers to immediately check if the model discovered in the discovery algorithm corresponds to the initial model, as the log is generated from this initial model.

To be able to better test specific discovery techniques, tools that allow for an straightforward way of generating a log from the simulation of a Petri Net are needed, and to my knowledge such a tool does not yet exist in the literature. To close this gap, this thesis presents a tool for log generation based on the simulation of Petri Nets. This extension will be built upon the already existing CPN simulator CPN-IDE [3], which will be detailed in detail in 3.2.1. The extension will allow the user to generate a log from the transitions fired during the simulation of a CPN.

1.2 Research Questions

The goal of this thesis is to create a tool with which we are able to generate logs from the simulation of a CPN. The simulation of the CPN will be done in the tool CPN-IDE [3]. I will assume that this program is able to simulate a CPN correctly.

The generated logs will be conforming to the XES Standard. The events in the generated logs can, next to the transition name, contain three kinds of attributes

1. A lifecycle-transition attribute, which shows the lifecycle of the event
2. A timestamp attribute, which shows the timestamp of the event
3. Possible multiple resource attributes, which show other resources used while executing the event

The thesis will not consider noise or adding other attributes to the log.

We built the application on the already existing structure of CPN-IDE. We Step-by-step expand the application with every sub-problem solved. While expanding the program we will have a program which is able to generate a log with more and more features until the tool eventually is able to generate logs with the above mentioned attributes.

First, the problem of generating a simple event log from CPNs is solved. Next, the solution for the first sub-problem is used to solve the problem of generating a simple event log from a Petri Net (PN).

After the first two sub-research questions are solved the modules used to solve the first two sub-research question are used as the basis to add the final three attributes to the log;
The first attribute wanted in the log was the lifecycle-transition attribute of the log.
The second attribute wanted in the log was the timestamp attribute of the

log.
The last attributes wanted in the log were resource attributes.

To verify if the generated logs are correct we will first check if the logs conform to the XES Standard, we will do this by importing the logs into ProM with the `"Import Conforming Log from IEEE XES Log"` import function, this will immediately check if the generated logs are conforming the XES Standard. We will also check the generated logs for correctness by replaying the generated logs on the original model.

When the logs are conforming to the XES Standard and the logs are able to be replayed on the model. We will be able to generate event logs which are correct.

1.3 Approach

We answered the research questions by adding modules to the CPN-IDE application, this application is owned by the process analytics cluster of the Department of Mathematics and Computer Science at Eindhoven Universitiy of Technology. The fact that the code was owned by the university meant that I was able to directly change the source code of the application.

The first research question was answered by adding a `recording` module, a `caseId` module and a `generate log` module. The recording module was used to record the bindings of the fired transition, the caseId module was used to determine which events belong to which trace, and the generate log module was used to generate a log from the attributes found in the previous two modules.

The second research question was answered by transforming the PN into a CPN when certain requirements of the model were satisfied. Two modules were added to help the user easily transform a PN into a CPN. The `Petri Net Transformation` module was added to easily change the static elements of the model. The `Token Generator` module was added to quickly change the dynamic elements of the model.

The third research question concerned adding lifecycle-transitions to the log. Lifecycle-transition attribute were not an attribute of CPN-IDE (in CPN-IDE there is no way to assign a lifecycle transition attribute to a transition), therefore I was free to implement how I wanted this to work. In order to get a good judgement of the possibilities I brainstormed with some professors in the process analytics cluster which solution would be the best for me to implement. In the end I added two ways in which lifecycle-transitions can be added to the log. The first way is when the lifecycle attribute is in the transition name. The second way is that the program automagically gives each event a lifecycle attribute.

The fourth research question concerned adding timestamps to the log. CPN-IDE already had a time function built in. The time in CPN-IDE is either an integer or a real number. I used the value of this time, and transformed this time into a real time in which the user has the option to determine how much time time an increase of one integer means to the time in real life.

The last research question concerned adding resources to the log. While the XES Standard does have a resource attribute, after careful consideration I decided not to use this attribute. Reason for this being that the resource attribute in the XES-log can only have one element, while in CPNs it is possible to have multiple tokens with the same color consumed in the firing of the same transition. We do not want to lose the information about the number of resources consumed in the transition, therefore we decided to not use the built-in resource attribute of the XES-log, but put resources in another attribute, which would be conforming to the XES Standard.

For each of the above research questions models were constructed that allows us to test all the functionality of the added modules.

1.4 Findings

To verify whether the generated logs are correct we will firstly check whether the generated event logs adhere to the XES Standard. We will do this by importing the event logs into ProM with the `"Import Conforming Log from IEEE XES Log"` import function. This import function will immediately check if the generated logs are conforming to the XES Standard, since logs that are not conforming to the XES Standard will not be able to be imported using this import function. We will also check the generated logs for correctness by replaying the generated logs on the original model. If the logs can be replayed on the original model in full, the attributes in the log for each event will be correct. Lastly for some features of the log (such as time), we will manually check if the attribute shown in the log is according to our the settings.

For each sub-research question above we specifically constructed some models that would cover all the added features of the log. Since these models covered all the features added to the log, when the logs generated from these models are correct, the modules added to the log will work as expected.

All logs shown in the evaluation were tested according to the above described method. One by one I imported each log manually into ProM, then I replayed the log on the model, either manually or with a tool, and lastly

I manually checked certain attributes. All logs passed the tests, were conforming to the XES Standard, and were able to be replayed over the initial model.

Since we picked the models used in the evaluation in such a way that it would cover all the functionality of the program, and all the generated logs are conforming to the XES Standard, and can indeed by fully replayed on the model we conclude that the program behaves as expected, and we are able to generate correct event logs from the simulation of a CPN

Chapter 2

Background

This chapter introduces Petri Nets in Section 2.1, Colored Petri Nets in Section 2.2, Timed Colored Petri Nets in Section 2.3, simple event logs in Section 2.4 and event logs in Section 2.5. A good understanding of these topics is necessary to understand the methodological and technical parts of the thesis. This chapter also provides a comprehensive discussion of all prior work in the area on this subject in Section 2.6

2.1 Petri Nets

A PN is a bipartite graph containing three types of objects. These objects are **transitions**, **places**, **arcs** and **tokens**. When displaying a PN as a figure, places are portrayed as circles, transitions are portrayed as rectangles and arcs are portrayed as arrows. Each place may hold a natural number of tokens (including 0), pictured by small solid dots. These dots are used to represent elements of a process model, for example a case, an employee or a machine.
Transitions in the model can fire. When a transition t fires, the firing consumes tokens from places in the **preset** of t, and produces tokens in the places in the**postset** of t.

In this section the fundamentals of Petri nets are explained. This includes the definition, basic terminology and transition firing rules. The basic concepts given in this section are used throughout this thesis.

We define a Petri Net as $PN = (P, T, W, M_0)$ in which:

- P : is a finite set of places

- T : is a finite set of transitions, $P \cap T = \emptyset$
- $W : A_i \cup A_o \rightarrow \mathbb{N}$ is a multiset of arcs. In which $A_i = (P \times T) \rightarrow \mathbb{N}$, is an input function that defines directed edges from transitions to places, and $A_o = (T \times P) \rightarrow \mathbb{N}$, is an output function that defines directed edges from places to transitions. The output of these function is the weight of the arc.
- $M_0 : P \rightarrow \mathbb{N}$ is the initial marking

A marking is an assignment of tokens to places of a PN. The number and positions of tokens may change during the execution of a Petri net. The function $M(p) \rightarrow \mathbb{N}$ is used to define how many tokens place p has.

The arcs between places and transitions are shown in two functions: A_o and A_i. For each possible combination between a place p and a transition t there is a natural number namely $A_o(t, p)$ which shows the weight of the arc from place p to transition t. This value can be 0 when no arc is present, or a positive number when the arc is present.
The function A_i displays the same characteristics, with the difference being that the function A_i displays the weight of the arc from a transition t to a place p.
All the places that have a positive integer for a specific transition t in A_i are called the **preset** of t. All the places that have a positive integer for a transition t in A_o are called the **postset** of t.
In the graphical representation of a PN arcs are shown as arrows going from places to transitions and vice versa.

Actions in the PN are controlled by tokens and arcs. A PN executes actions by firing transitions, before a transition can fire a transition first has to be enabled. The `Enabling Rule` and the `Firing Rule` describe the characteristics of tokens in input and output places, which determine whether a transition can fire.

Enabling Rule: A transition t is enabled if and only if for each input place p of t, p contains at least the number of tokens equal to the weight of the arc going from p to t. Or more formally: $\forall p \in P, M(p) \geq A_i(p, t)$. Note that when there are no edges $A_i(p, t)$ will be zero, and note that $M(p)$ cannot be smaller than zero.

Firing Rule: When a transition t is enabled, transition t can fire and if and only if transition t fires, then from all places p in the preset of t the number of tokens equal to the weight of the edge going from p to t are consumed. More formally: for a place p, $A_i(p, t)$ tokens will be consumed (deleted). Also tokens are produced (created) in all the places p in the postset of t. So for each place p, there will be $A_o(t, p)$ new tokens produced in p.

Silent transitions Silent transitions are transitions in a PN that are simply not observed in real life and therefore don't appear in the log. These transitions often serve to transfer tokens from one place to another place. Visually these silent transitions are often displayed as a black bar or as a transition with a black background.

2.2 Colored Petri Nets

Colored Petri Nets [11] are an extension of regular Petri Nets. This extension makes it possible for Petri Nets to carry data. This extension gives a Petri Net a greater expression power.

Data is added to the Petri Nets by means of colors and color sets, color sets can be seen as datatypes, and colors can be seen as values of these datatypes. Adding colors and color sets to PN allows the following things in a CPN.

- Each token now has a color. This can for example be an `Integer` value, a `String` value or any other value of a specific datatype.
- Each place now has a color set, this means that a place can only contain tokens of a certain color set. We define a function Θ that maps each place to a color set. For example, if a place p has the color set `Integer`, then only tokens with an `Integer` color can reside in the place.
- The function $M(p)$ now gives the colors of all the tokens in p instead of the number of tokens.
- Color sets can also be combinations of previously determined color sets. For example, say we have already established the color set `Machine`, and we have already established the color set `User`. We can now establish a new color set called `Machine-User`. Which will first have an element of the color set `Machine`, and second an element with color set User. A possible token of color `Machine-User` will be `(Forklift4, User001)`.
- We can now declare variables of a specific color set. For example a variable `x` defines a value of color set `Integer`.
- Arcs are labelled with expressions that show how tokens are processed when a transition fires. Expressions are part of a language Λ.
  the basics of this language are colors, for example `1`, and variables, for

example `x`. This language can be extended to include a combination of variables. For all arcs the expression on the arc should be of the same color set as the place of the arc. When the arc is an outgoing arc (coming from a transition to a place), all variables on the arc should be declared either in one of the incoming arcs. Specifically, the language of the arcs is constructed as follows:

1. We start the language with colors and variables. These are the same colors and variables as described above.
2. We can combine colors and variables, for example given the example above from the color set `Machine-User`, say we have defined a variable `m` of color set `Machine` and a variable `u` of color set `User`. Then there are two ways to place the color set `Machine-User` in the expression. The first way is to create a new variable of the color set `Machine-User`, and place this variable on the arc. The second way is to use the already existing variables, and combine them in the expression, this would look the following: *(m,u)*.
3. We can add positive integers to the arc expressions declared in 1 and 2. A positive integer indicates that an amount of tokens with the same value are requested by the arc. For example `2‘x` indicates that two tokens with the same color are needed.
4. If multiple tokens are needed that are not the same color we can add them with a `++`. Say that we need two machines, and we have defined color *"m1"* and color *"m2"* of color set `Machine`, then we can denote this on the arc by placing *1‘m1 ++ 1‘m2* to indicate we need two machines. Note that the color set of the variables should be the same, as they are getting the tokens from the same place, and all the tokens of that place will have the same color set.

- In a binding a variable is assigned one color. For example say we have a transition with two incoming arcs, one from p_1 and one from p_2, both arcs have variable x placed on it, and the color set of this variable is the same as the color set of the places p_1 and p_2. Then the transition is only enabled if there is a token $to_1 \in M(p_1)$ which has the same color as a token $to_2 \in M(p_2)$. When the transition fires, the tokens with the color used in x will be consumed from p_1 and p_2, and if there is an outgoing arc with x on it then a token with value x will be created in this place.

When a transition in a Colored Petri Net fires, each variable on an incoming and outgoing arc is assigned a value. The list of key-value pairings used in each firing of the transition is called the **binding** of the fired transition

With data on the tokens, the transitions in the model can have multiple attributes such as users, document number etc. This will help greatly in generating realistic logs.

2.3 Timed Colored Petri Nets

Timed Colored Petri Nets (TCPN) [5] are an extension of Colored Petri Nets. This extension makes it possible for tokens in the CPN to carry time next to data. TCPNs will be explained with the basics of CPNs in mind.

As the name suggests in TCPNs time is included. This is done by creating a global time variable and by giving each token a time value, this value is the moment in time. A TCPN works according to the following rules.

- There exists a global time variable. This variable starts at 0.0, and increases when there are no enabled transitions. This variable increases to the time when the next transitions will be enabled.

- Tokens have a time associated to it, this is a real number, schematically this is done by adding `@x` where x is the time the token is available again.

- Each transition now has a time associated to it. This is the time at which the transition will be enabled. This time is the same time as the highest time of any of the tokens consumed in the firing of the transition. Only when the global time is equal to or lower than the transition time the transition will be enabled.

- A transition can take time. This attribute of the transition is for example `@+x` where x is a real number. Besides taking a predetermined amount of time. A transition can also take a non-deterministic amount of time. For example the `exponential(x)` function can be used to generate values from exponential distributions.

- An outgoing edge can also take time. This value is placed directly behind the expression on the arc, and can also contain a function.

- When a token is produced the time of that token will be the time of the transition plus the time the transition takes plus the time the arc takes.

2.4 Simple event log

Assume the set of all process activities Σ is given. An event is the occurrence of an activity. A trace t is a (possible empty) sequence of events, for example $< a, b >$ is a trace. In $< a, b >$, first activity a occurs then activity b occurs. A simple event log L is a finite non-empty set of traces in which the event only contains an event-name. For example $[< a, b >, < b, a >]$ denotes a simple event log consisting of two traces $< a, b >$ and $< b, a >$.

To be able to construct a simple event logs from data we need at least the following three attributes for each event.

- The first attribute is the `activityName`, this is the name of the activity the event refers to, this is used to differentiate between the different activities.

- The second attribute is the `traceId` of the event. The traceId is used to determine which trace the event is part of.

- The third attribute is the `position` of the event. The position attribute indicates which position the event has in the list of all events. For example, if we have only have two traces $< a, b >$ and $< c, d >$, and the events happen in order $acdb$ then event a has position 1 in the list, event c has position 2 in the list, event d has position 3 in the list and event b has position 4 in the list. Once the events are sorted based on traceId, we can find the order of both traces, and see that the two traces are $< a, b >$ and $< c, d >$. In this thesis we will portray the position attribute of an event in the eventId attribute. This means that the event that happened first has the eventId e_1, the event that happened second e_2 etcetera.

These attributes are used to create simple event logs.

Lists of events are often portrayed as `Event Tables`. For example say we have the events of table 2.1, and we want to create a log from this as shown above. These events are ordered based on position because this is the order the events entered the system.

We first order the events based on traceId, retaining the order by position.

Table 2.2 shows that there are three events with traceId 1, these are a, c and b. Table 2.2 also shows that a happens before c and c happens before b. Thus using this information we know that the trace $< a, c, b >$ happened. When using the same technique for the other traceIds the simple event log

eventId	activityName	traceId
e_1	a	1
e_2	b	2
e_3	a	3
e_4	a	4
e_5	b	4
e_6	c	1
e_7	a	2
e_8	b	3
e_9	b	1
e_{10}	c	3
e_{11}	d	4

Table 2.1: A possible list of events sorted on eventId

eventId	activityName	traceId
e_1	a	1
e_6	c	1
e_9	b	1
e_2	b	2
e_7	a	2
e_2	a	3
e_8	b	3
e_{10}	c	3
e_4	a	4
e_5	b	4
e_{11}	d	4

Table 2.2: A possible list of events sorted first by traceId then by eventId traceId

$[< a, c, b >, < b, a >, < a, b, c >, < a, b, d >]$ can be extracted from this table.

This technique allows the creation of a simple event log from any set of events, as long as it has these three attributes.

2.5 Event Logs

Events are also allowed to have more attributes than the three mentioned above, those attributes include but are not limited to a lifecycle-transition attribute, a timestamp attribute and one or multiple resource attributes. When we want to include these attributes in the log we are extending our simple event logs into an event log.

Lifecycle transition attribute

One of the possible attributes of an event is the lifecycle-transition attribute. The lifecycle-transition attribute refers to the lifecycle of activities. In most situations activities take time, for example the activity cook will not start and end at the same time. The lifecycle-transition attribute is there to indicate the lifecycle of the activity cook, one event can have the lifecycle attribute start, and one event can have the lifecycle attribute complete. This way the activityName simply stays cook, but the lifecycle-transition attributes indicates the lifecycle of the cook transition. In the end the lifecycle attribute allows the activityName to be easier. In this thesis we assume the lifecycle-transition model from the official XES website [4], this model is shown in figure 2.1

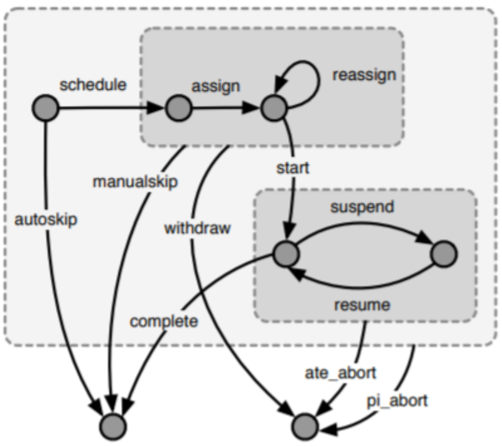


Figure 2.1: The lifecycle model

Timestamp attribute

Another possible attribute of an event is the Timestamp attribute. The timestamp attributes gives the date and time of the event when it happened in real life.

Resource attributes

More possible attributes are resource attributes. Resource attributes can have any name, but specify which resources are used in an action. A possible resource can be an user. This resource attribute specifies which user performed the action.

Event log with all attributes

When these three attributes are added to events we can create for example the event table of Table 2.3. All the events in Table 2.3 have activity a as activityName, however activity a only happens two times. Event e_1 and e_3 record the start of activity a, and event e_2 and e_4 record the end of activity a. Thus activity a is executed two times. Each event also has a timestamp, and each event also has an user. Event e_1 and e_2 are done by `Jaap`, event e_3 is done by `Geert` and event e_4 is done by `Henk`.

eventId	activityName	traceId	timestamp	lifecycle	user
e_1	a	1	01/02/2013 15:40	Start	Jaap
e_2	a	1	01/02/2013 17:20	Suspend	Jaap
e_3	a	1	02/02/2013 10:00	Resume	Geert
e_4	a	1	02/02/2013 14:00	Complete	Geert

Table 2.3: A possible list of events including extra attributes excluding eventId

2.6 Related work

The work presented in this thesis is related to previous work that has been done in the field of process analytics. The logs currently used in the BPI challenge [2] are logs generated from real life data. These are datasets that portray the events as they happen in the real-life world the best. However we do not have a constant supply of these datasets. These datasets often take months to generate (as they are from real life systems). The literature has tried to solve this shortage by creating synthetic datasets. All approaches in the literature create a log based on different model languages. None of the approaches in the literature generate a log based on the simulations of Petri Nets. The approach used in Burattin and Alessandro [6] is based on generation of process descriptions via a stochastic context-free grammar whose definition is based on well-known process patterns. The approach used by Di Ciccio et al. [7] generates logs based on declarative models, while the approach of Skydanienko [13] uses an approach based on multi-perspective declarative models. Leiva et al [9] generates logs based on unplugged processes and Alexey [10] generates logs based on BPMN models.

Noise generation

There has also been a lot of research in the area of noise generation. Shugurov et al. [12] build a plug-in in ProM to create logs with noise from a PN. Jonghyeon et al. [8] used existing event logs to generate new event logs with noise. While this thesis will not go into detail about adding noise to the log, adding noise to logs is an important part of log generation, and should therefore be mentioned.

Chapter 3

Problem Exposition

This chapter introduces the problem context in more detail than Section 1.2 in Section 3.1, this chapter will also explain the research topic in relation with the background discussed in Chapter 2, it will explain in detail how the background is relevant for the problem context. Next this chapter will explain the setting in which the research question is answered, then this chapter will explain how the research problem is tackled, namely by subdividing the main problem into several sub-research questions, each of these will briefly be touched upon in this chapter.

3.1 Context Understanding

In Section 2.3 Timed Colored Petri Nets (TCPNs), and their functionality were explained. One of the functionalities of a TCPN is that a TCPN can be simulated. During the simulation of a TCPN transitions fire. The transitions that fire during the execution of a TCPN are recordable. The goal of this thesis is to devise a method to record the firing of these transitions along with the color of the token used in the firing of the transition, and to generate a log from these recordings.

3.2 Data Understanding

In this thesis the method to generate logs from TCPNs will be explained. The method is based on an implementation of the functionality in CPN-IDE. This tool will be further explained in Chapter 3.2.1. In this tool an extension is created that allows the user to record the fired transitions of a TCPN, how TCPNs are displayed in CPN-IDE is explained in Chapter 3.2.2. From this recording a log conforming to the XES standard will be exported to a file. The XES Standard will be explained in Chapter 3.2.3.

3.2.1 CPN-IDE

The application in which the extension is built is CPN-IDE[3]. CPN-IDE is a modelling and simulation tool for TCPNs. The underlying structure of CPN-IDE is divided into two parts: the front end and the back end. The front end of CPN-IDE handles all things related to user interaction, user visualisation and modelling. The user interaction part of the front end includes pressing a certain transition to fire, choosing a specific binding of a transition to fire and selecting a number of transitions the model should fire. The user visualisation part of the front end includes the showing of the TCPN, the showing of the current marking of the simulation as well as the showing which transitions are enabled for the user to fire. The modelling part of the front end includes all functionalities in creating a model; creating new places, creating transitions, creating arcs, and adding labels and attributes to all these objects. The back end of CPN-IDE consists of the `SimulatorController`, the `PetriNetContainer` and the `Simulator`, and handles the checking for correctness of the models and simulating the models. The `Simulator` component of the back end was code not accessible by me. I had to work with the already existing functions to gather information from the `Simulator`. This only allowed me to perform certain actions with the `Simulator`.

A diagram of how the different parts of CPN-IDE interact with each other in the simulation state is given in Figure 3.1. In this diagram arrows rectangle represent the different parts of the application, and the arrows represent communication between the different parts. In this diagram we identify four components. `FE` is the front end component. `BE` is the back end component. `SC` is the `SimulatorController` component. This component is responsible for the back end side of the communication with the front end. `PNC` is the `PetriNetContainer` component. This component is responsible for communicating with the `Simulator`. `Sim` is the `Simulator`. Arrows represent communication between the components. A black arrow corresponds to either the firing of one transition or the firing of a number of transitions. A blue arrow corresponds to the firing of one transition. Blue arrows from the `Simulation` do not have any attributes, they simply tell the `PNC` that the `Simulator` is done firing the transition.

In practice this looks as follows: first the entire model is created in the front end. Then when the user wants to simulate the model the front end sends the model to the back end. The back end checks if this model is valid,

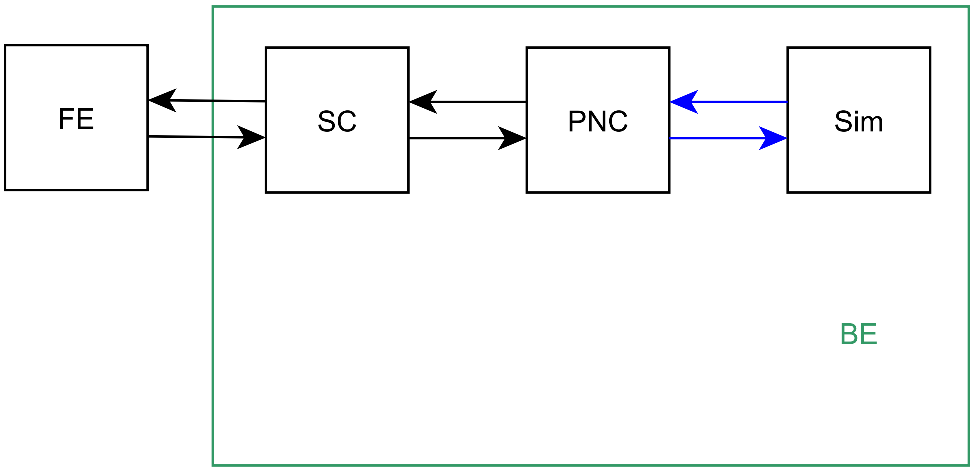
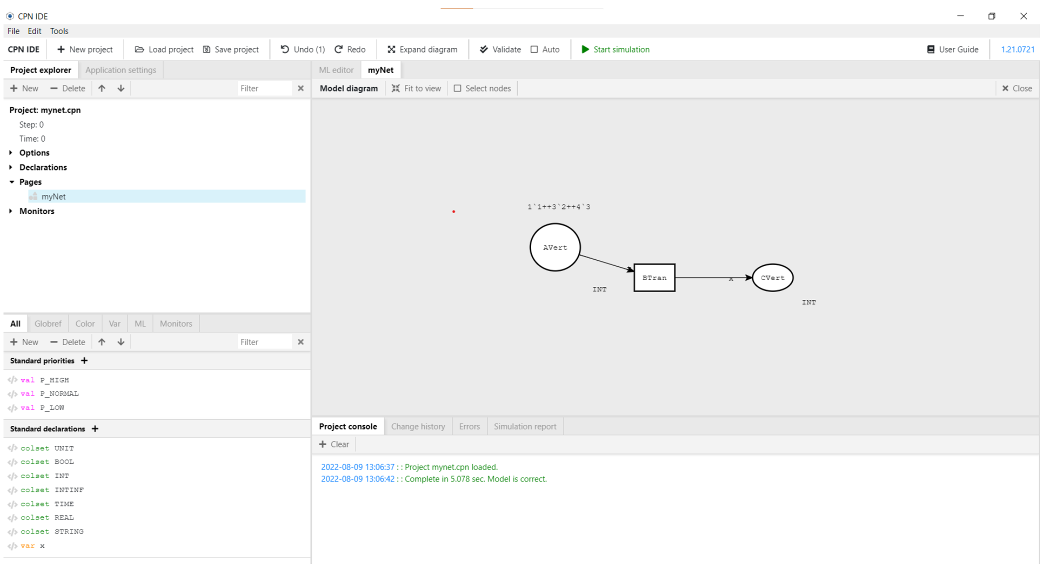


Figure 3.1: The components of the original CPN-IDE application

and if so it starts a simulation. The enabled transitions are then sent from the back end to the front end, and displayed in the front end. Next, the user has the ability to either fire a specific transition or select a number, this number indicates how many transitions should fire. This information is sent from the front end to the back end. Then the back end sends the transitions to fire to the simulator one at a time. The simulator fires the transition(s), and the new marking with the new enabled transitions are sent to the front end.

The CPN-IDE program has two states. When opening the program the program is in the `modelling state`. In the `modelling state` the user is able to save, open and edit models. When pressing the `start simulation` switch the user will enter the `simulation state`. In the `simulation state` the user is not able to do the above mentioned actions, however the user is able to fire transitions and simulate the model. When pressing the `stop simulation` switch the program will enter the `modelling state` of the program again. Figure 3.2 shows a screenshot of the `modelling state` of the application. Figure 3.3 shows a picture of the `simulation state` of the program.

An important feature of the `modelling state` of the application as shown in 3.2 is the `start simulation` button in the top bar. Pressing this button changes the state of the simulation from the `modelling state` of the program to the `simulation state`. Another important feature is the `model diagram` panel shown in the right middle of the screen, this diagram shows the model (which is a Petri Net or Timed Petri Net), and allows the user to change the it. Last important feature is the properties and declarations panel in the lower left. In this panel color set and variables are declared.



Project: mynet.cpn
Step: 0
Time: 0
Options
Declarations
Pages
myNet
Monitors

Standard priorities

```
val P_HIGH
val P_NORMAL
val P_LOW
```

Standard declarations

```
colset UNIT
colset BOOL
colset INT
colset INTINF
colset TIME
colset REAL
colset STRING
var x
```

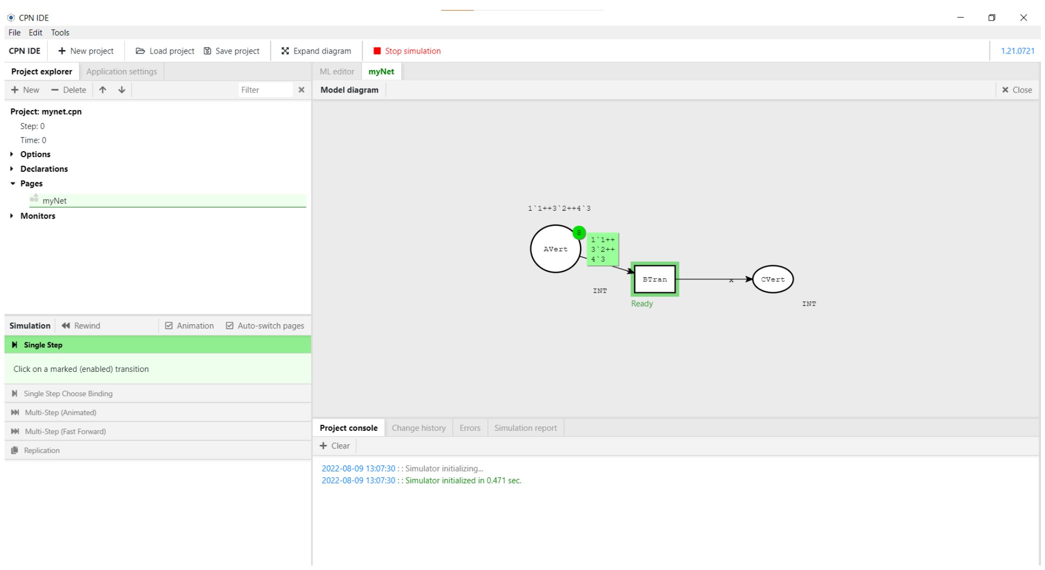
```
2022-08-09 13:06:37 : : Project mynet.cpn loaded.
2022-08-09 13:06:42 : : Complete in 5.078 sec. Model is correct.
```

Figure 3.2: The modelling state of CPN-IDE

An important feature of the `simulation state` of the application as shown in 3.3 is the `stop simulation` button in the top bar. Pressing this button changes the state of the program from the `simulation state` of the program to the `modelling state`. Another important feature is the `model diagram` panel, this panel shows the model and the current marking of the simulation. Also enabled transitions are encircled with a green line. The simulation panel, shown in the bottom left of the screen. Shows which particular simulation actions can be taken. This includes firing a single transition (as selected in the screenshot), or firing multiple transitions. While the `Single step` action is selected. The user is able to select a transition in the `model diagram` panel, if this transition is enabled then this transition will fire.

3.2.2 Petri Nets, Colored Petri Nets and Timed Colored Petri Nets

In the thesis the PN, CPNs and TCPNs will be shown as they are shown in CPN-IDE. Besides the properties mentioned in Section 2.3 this also entails that tokens are displayed slightly differently. Instead of each place having a possible infinite amount of tokens, the tokens are textually represented, more specifically tokens are represented by a pair in which one element is an integer greater than 0, and the other element is the color of the token. For



Project: mynet.cpn
Step: 0
Time: 0
Options
Declarations
Pages
myNet
Monitors

Click on a marked (enabled) transition

```
2022-08-09 13:07:30 : : Simulator initializing...
2022-08-09 13:07:30 : : Simulator initialized in 0.471 sec.
```

Figure 3.3: The simulation state of CPN-IDE

example: say we have two tokens with color 1, we can represent this textually as 2’1. When there are multiple tokens with different values the tokens are separated by `"++"`.

A model in CPN-IDE with the above mentioned display is shown in Figure 3.4. In this example there is one token with value 1 situated in P1.

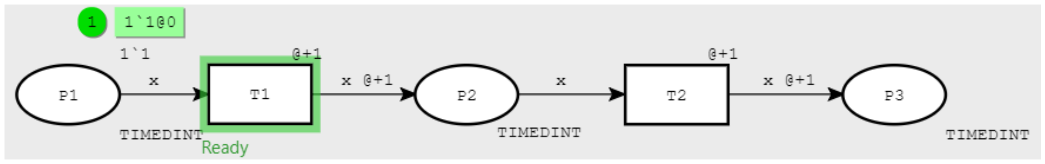


Figure 3.4: An example of a TCPN in CPN-IDE

all places contain four attributes and all transitions contain three attributes. In CPN-IDE the user is able to freely move these attributes all over the model diagram, for the sake of simplicity we will assume that the different attributes stay at the same place for all nets shown in this thesis.

A place contains four attributes. The name of the place is the first attribute, this attribute is placed in the middle of the place, and is also known as the `place name`. The second attribute is the `color set` of the place, this attribute is placed in the bottom right just outside the place. The third and fourth attribute are the `initial marking` and the `current marking`. The initial marking is displayed in the top right of the transition. The current marking is displayed in green boxes, also at the top right of the place.

A transition contains three attributes. The name of the transition is the first attribute, this attribute is placed in the middle of the place and is also known as the `transition name`. The second attribute is the `time` a transition takes, this attribute is placed in the top right of the transition. The third attribute is the `silent transition` attribute. This is a boolean attribute that shows if the transition is a silent transition or not, silent transitions are displayed with a black background and not silent transitions with a white background.

For an arc the only place where information can be stored is on top of the arc, however this inscription can contain multiple elements. The first elements is an expression as discussed in Section 2.2. The second element is the time an edge takes, this number is places after a `\@+"`.

The data and models studied in this thesis are data and models that are specifically picked and created. The data and models are picked and created in such a way that the data and models will cover all parts of the implemented features.

3.2.3 Logs conforming to the XES Standard

In this thesis I want to create logs from simulating TCPNs. We want these logs to be universally usable and conforming to a specific standard. The XES Standard was chosen to be this standard. This standard was chosen as it is the language officially published by the IEEE as the standard for Achieving Interoperability in Event Logs and Event Streams. [4].

A log conforming to the XES Standard looks something like the log shown in Listing 3.1

Listing 3.1: XES example log

```
1  <log>
2      <trace>
3          <string key="concept:name" value="1"/>
4          <event>
5              <string key="concept:name" value="T1"/>
6          </event>
7          <event>
8              <string key="concept:name" value="T2"/>
9          </event>
10     </trace>
11 </log>
```

A XES log has a hierarchical order. A log contains log attributes and traces, a trace contains trace attributes and events, and an event contains event attributes. Each line of a XES log has two possible uses. Either it states the beginning or end of an object, or it gives an attribute of an object. For example lets take the event object of Listing 3.1 on lines 7 till 9. Line 7 shows the start of the object, and line 9 shows the end of the object. Line 8 shows an attribute of the object. Attributes of the event are shown following the following template

```
<TYPE key=KEY value=VALUE/"
```

in which all words in capital letters should be filled in correctly by someone. Lets look at line 7 again. The key of the object of line 7 is `"concept:name"` the type of the object is `string` and the value of the object is `"T2"`

The `concept:name` attribute of an object in a XES log is used to denote the ID of the object. When the `concept:name` attribute is the object of a trace the `concept:name` attribute denotes the traceId. When the `concept:name` attribute is an attribute of an event the `concept:name` attribute gives the activityName of an event.

If we try to simplify the log shown in Listing 3.1 we see that the log has one trace. This trace has traceId 1 and has two events. One event has transition name T1 and one event has transition name T2. Thus the log is listing 3.1 is a XES log of the log $[< T1, T2 >]$ in which the trace has traceId 1.

3.3 Detailed Research Questions

The main problem of the thesis can be subdivided into two research problems. The first research problem will solve the question of generating logs conforming to the XES standard based on simple event logs which are generated from CPN or PN. This research problem will be solved in Chapter 4. This problem will be solved by first generating simple event logs based on the execution of a CPN in Chapter 4.1, then having this solution as a base method Chapter 4.2 will explain how this method can be used to solve the problem for PNs. The second research problem will be explained in Chapter 5. This research focusses on generating realistic logs conforming to the XES standard from TCPN. The realism is added to the logs by adding lifecycle-transitions in Chapter 5.1, by adding time to the log in Chapter 5.2, and by adding resources to the log in Chapter 5.3. This division between research problems was chosen because the first research problem focusses on generating a log, and the second research problem focusses on expanding the log generated.

In both the research problems the models will be presented as CPN-IDE models, and the resulting logs will be given in a `.xes` file.

Chapter 4

Generating simple event logs

This chapter explains how a simple event log conforming to the XES Standard can be created from the simulation of a Colored Petri Net (CPN) and a Petri Net (PN). As explained in Section 2.5, when we have a list of events, and each event has an activityName and a traceId, and we can determine the position, then we can construct a simple event log from this list of events. This chapter explains the modules created in the application that ensure that events can be recorded from a CPN and PN, and that each event has the before mentioned attributes. After explaining this the chapter will also explain how the simple event log is transformed to a log adhering to the XES Standard.

The remainder of this chapter is structured as follows. In Section 4.1 we will create a simple event log conforming to the XES Standard from any Colored Petri Net. Then in Section 4.2 we will use the methods previously discovered to create a simple event log conforming to the XES Standard for some PNs.

Both research questions will be answered in the same way, first the implemented methods will be explained, then the thesis will explain how the implemented modules are relevant for solving the sub-problem.

4.1 Generating a simple event log from a Colored Petri Net

4.1.1 Chapter description

The first sub-research question revolves around generating a simple event log from a Colored Petri Net (CPN). To illustrate this take the model shown in Figure 4.1

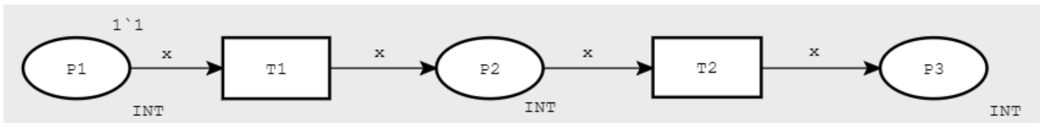


Figure 4.1: A simple CPN

If we try to execute this model, we first check which transitions are enabled. The only transition that is enabled is T1, so we fire T1. The token currently in P1 will be removed, and a new token will be created in P2. Then we check again which transitions are enabled. The only transition that is enabled is T2, so we fire T2. The token currently in P2 will be removed, and a new token will be created in P3. We can generate a log by hand from this model, this would be $[< T1, T2 >]$. Our goal is that the application is also able to do this.

In section 4.1.2 the first module which records transitions in CPN-IDE will be explained. In section 4.1.3 the second module which adds a global caseId variable is explained. In section 4.1.4 the third module which creates the log from the recorded activities is explained. Lastly in Section 4.1.5 all the modules are combined to generate a log from the simulation of a CPN.

4.1.2 Recording fired transitions

The first module added is the recording module, this module allows the user to record the firing of transitions. More specifically, when the program is in the simulation state, the module adds a `record events` switch. When this switch is pressed, the application will enter the recording state and will record the transitions that fire, when the switch is not activated, the program will be in the simulation state and the transitions that fire will not be recorded. The recordings will stay saved while the user is in the recording state or in the simulation state. Only when exiting the simulation state to the modelling state of the program or pressing the `delete recordings` button the recordings will be deleted. this way the original functionality of the simulation remains and no additional computational power is needed to run the original program. When the program is recording transitions and a transition fires, the transition and all its attributes together with the binding of the fired transition will be recorded. This is done by catching the binding of the executed transition and recording this binding. The program records the transition the moment it fires in a Queue, this means the transition that will be recorded first will be the transition that has fired first, thus preserving the order of the simulation. It is important to note that the program remembers all recorded events while in the same simulation, only by pressing the `clear recordings` button or restarting a simulation will the recordings be removed. This module also has a submodule that ignores irrelevant fired transitions.

Ignore irrelevant fired transitions

The submodule of the recording module is the silent transitions module. This module is placed within the record activities module and makes sure the program only records transitions that are not silent transitions. As discussed in the previous paragraph, the transition and all its attributes are recorded. One of those attributes is the background of the transition. The value of this attribute is used to determine whether the transition is a silent transition or a normal transition, as silent transitions have a black background. When trying to record a transition, the transition is checked against a list of silent transitions, when it is part of this list, the transition is ignored. This simulates real life, as silent transitions are also not recorded in real life. The list of silent transitions is constructed when the model is sent to the simulator, the recording module catches the places (and its backgrounds) and creates a list of silent transitions.

4.1.3 Create caseId variable

The second module added is the caseId module. This module gives the user the possibility to fill in a caseId variable, which serves as a traceId for the events. More specifically, as discussed in Chapter 2.2, when creating a Colored Petri Net the user defines a set of colors and a set of variables, in this module the user selects one of the variables to be the caseId variable. The user is able to change the caseId variable at any point during the recording process. Only when generating the actual log is the caseId variable used.

4.1.4 Generating the log

The third module added is the generate log module. This module generates a simple event log from the recorded activities. The simple event log can be generated at any point in time. The module uses the OpenXES Library [4], this library has, next to many other classes, a log, trace and event class. We will be using these classes to create the log in our implementation. The generate log module gets the queue of recorded activities point, then the module will loop over the queue, starting from the first activity. Then for each transition the program will check whether the caseId variable is part of the binding, if this is the case, the program will use the value for the caseId variable as the traceId and thus have a value for the traceId for the event. When the caseId variable is not part of the binding-variables the program will ignore the event, as it does not have a traceId and thus is not part of a trace and thus is not interesting for the log. After checking the traceId of an event, the module will add the event to the trace it belongs to, which is the trace with the same traceId. When no such trace exists a new trace is created. When all events are placed in their corresponding trace a ".xes" file will be created and the list of traces will be parsed into this file using the parse function of the OpenXES library. During this parsing the transition name will become the activityName in the log. In the original program the program already saved a simulation report to the files of the user. I used this functionality to slightly change the folder and place the generated log in the folder with the following path `path/CPN-IDE/logout/CPN-sessionID`. In this path the CPN-sessionID is automatically generated by CPN-IDE when starting the simulation.

4.1.5 Combining the modules

All the modules discussed in the previous section are part of the `Recording` component. How the `Recording` component interacts with the original components is shown in Figure 4.2. This diagram is an expansion on the original diagram shown in 3.1. Compared to the original diagram there are a couple of key changes. First the `Recording` component and the `Disk` component are added. The `Recording` component contains all modules discussed before as well as components that we discuss in later chapters. The `Disk` component is the hard drive of the user that installed the program.
Also a lot of arrows were added. The blue arrows correspond to transitions being fired in the `Simulation state`, these arrows are always in a specific order; first arrow 1 happens which sends the transition that should be fired to the `Simulator`, then arrow 2 tells the `PNC` that the `Simulator` is done

firing the step. Next, arrow 3 requests the current state of the simulation, the current state of the simulator includes the last binding fired. Next, arrow 4 sends the current state of the simulation back to the PNC. Then the `Recording` component is called with as argument the current state of the simulation. Lastly, arrow 6 tells the `PNC` that the `Recording` component is done. Only when all six arrow steps have been taken can we continue with firing a new transition using arrow 1. The red arrows correspond to log configuration options sent to the recording module, these include the caseId variable, the name of the log and many more. The pink arrows correspond to the location of the generated log. The log is always saved in a specific folder that cannot be changed by the user, the location of this folder is sent to the front end to be displayed to the user. The light blue arrow corresponds to the log being saved on the disk.

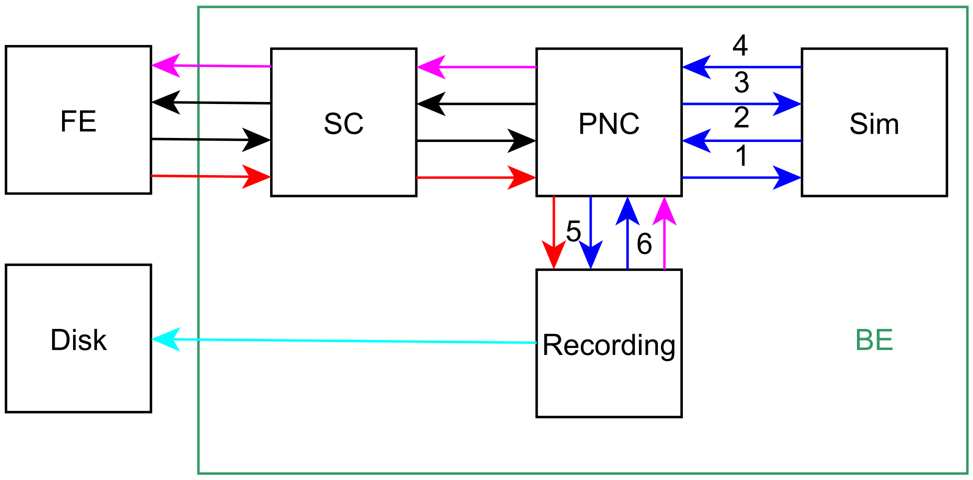


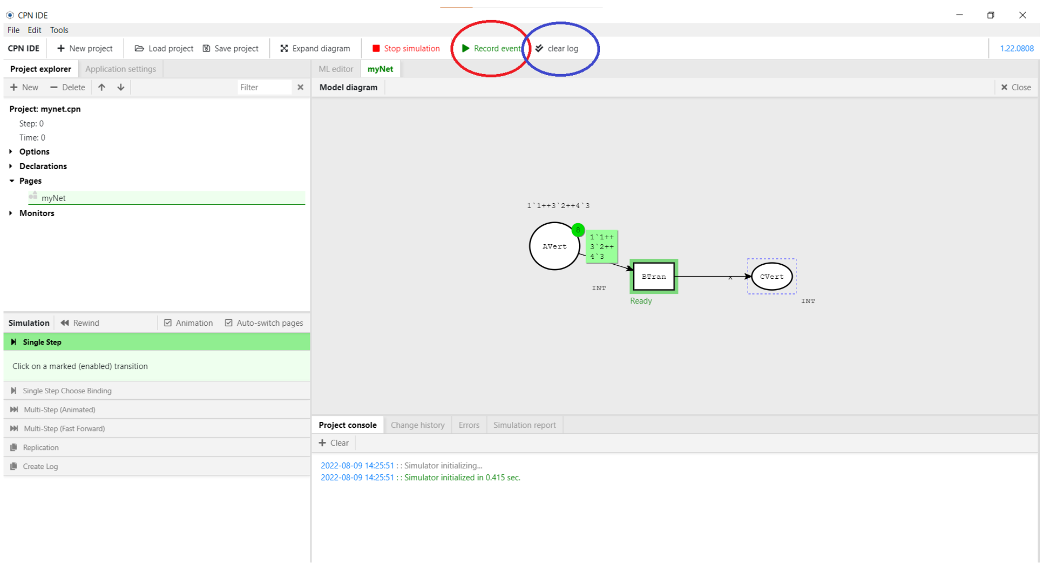
Figure 4.2: The components of the extended CPN-IDE application

4.1.6 Result

This section shows the result after implementing the modules described above. This section will do this by first showing the GUI after implementing the modules described in the sections of this chapter. Next this section will give a walk-through of how a log can be generated from a CPN in CPN-IDE. Lastly this section will generate some logs from some models, these models will be picked in such a way that all functionality of the feature is included.

Updated GUI

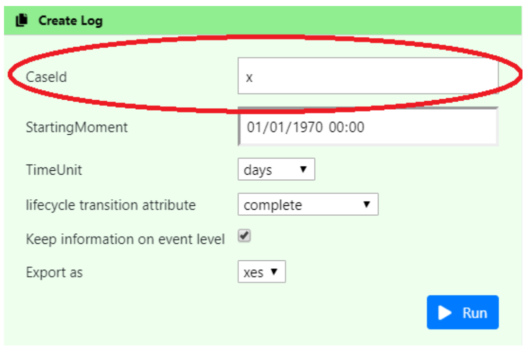
The implementation of the added modules is shown in Figure 4.3



```
2022-08-09 14:25:51 : : Simulator initializing...
2022-08-09 14:25:51 : : Simulator initialized in 0.415 sec.
```

Figure 4.3: The updated simulation state of the program

Most notable are the additions to the top bar. A switch named `record events` (shown with a red circle) was added and a `clear log` (shown with a blue circle) button was added. Pressing the record events will move the program from the simulation state to the recording state. The `clear log` button will clear all the current recorded events. Also added to the simulation state of the program is the `create Log` tab in the simulation panel on the bottom left of the screen. The folded out `create log` tab is shown in Figure 4.4



CaseId	x
StartingMoment	01/01/1970 00:00
TimeUnit	days
lifecycle transition attribute	complete
Keep information on event level	✓
Export as	xes

Figure 4.4: The create log panel

In this tab the user can select the `caseId` as discussed in 4.1.3, and run the generate log module as discussed in 4.1.4 (this option is shown with a red circle). The other functionality of the create log tab is discussed in later chapters as they do not add functionality discussed in this chapter.

Walk-through

Now this thesis will give a walk-through how to create a log from a simple CPN. The model chosen for this example is shown in figure 4.5, during this walk-through arrows will guide you to the buttons to press.

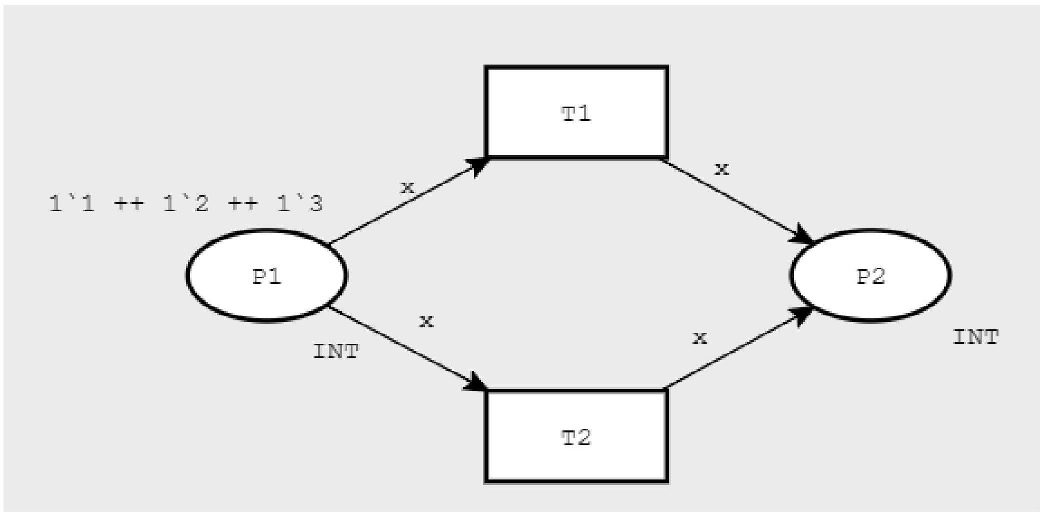
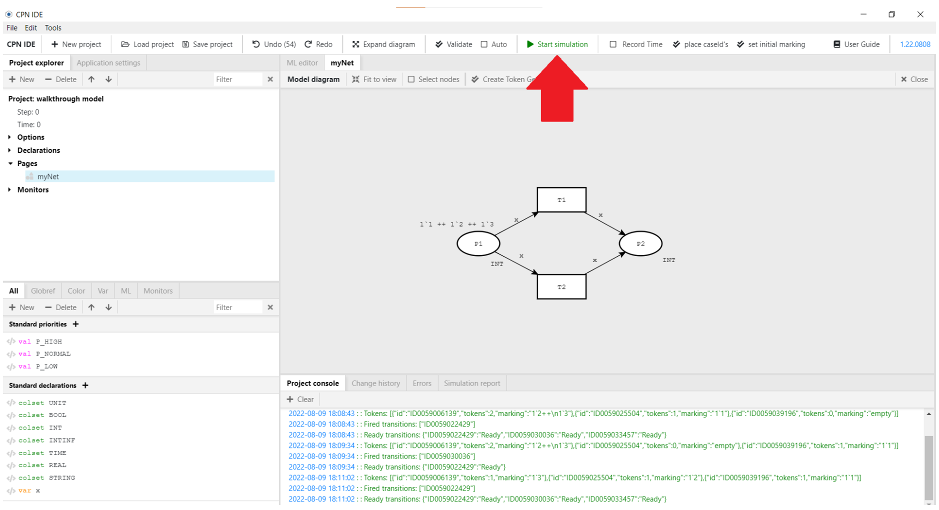


Figure 4.5: The model

We start this walk-through in the `modelling state` of the program in Figure 4.6. Here it is possible to change the model conforming to what we want. We have chosen and created the model of Figure 4.5.

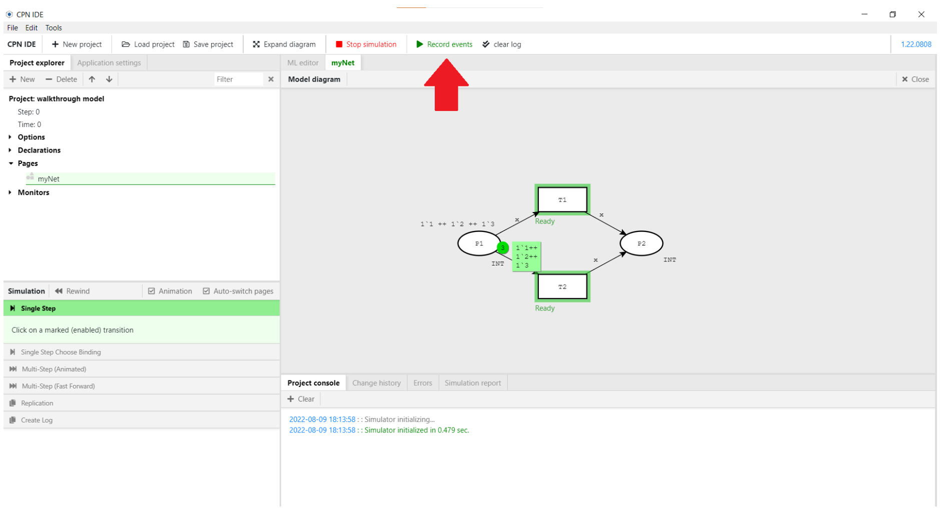


```
val P_HIGH
val P_NORMAL
val P_LOW
colset UNIT
colset BOOL
colset INT
colset INTINF
colset TIME
colset REAL
colset STRING
var x
```

```
2022-08-09 18:08:43 : : Tokens: [{"id":"ID0059006139","tokens":2,"marking":"1`2++\n1`3"},{"id":"ID0059025504","tokens":1,"marking":"1`1"},{"id":"ID0059039196","tokens":0,"marking":"empty"}]
2022-08-09 18:08:43 : : Fired transitions: ["ID0059022429"]
2022-08-09 18:08:43 : : Ready transitions: {"ID0059022429":"Ready","ID0059030036":"Ready","ID0059033457":"Ready"}
2022-08-09 18:09:34 : : Tokens: [{"id":"ID0059006139","tokens":2,"marking":"1`2++\n1`3"},{"id":"ID0059025504","tokens":0,"marking":"empty"},{"id":"ID0059039196","tokens":1,"marking":"1`1"}]
2022-08-09 18:09:34 : : Fired transitions: ["ID0059030036"]
2022-08-09 18:09:34 : : Ready transitions: {"ID0059022429":"Ready"}
2022-08-09 18:11:02 : : Tokens: [{"id":"ID0059006139","tokens":1,"marking":"1`3"},{"id":"ID0059025504","tokens":1,"marking":"1`2"},{"id":"ID0059039196","tokens":1,"marking":"1`1"}]
2022-08-09 18:11:02 : : Fired transitions: ["ID0059022429"]
2022-08-09 18:11:02 : : Ready transitions: {"ID0059022429":"Ready","ID0059030036":"Ready","ID0059033457":"Ready"}
```

Figure 4.6: The modelling state

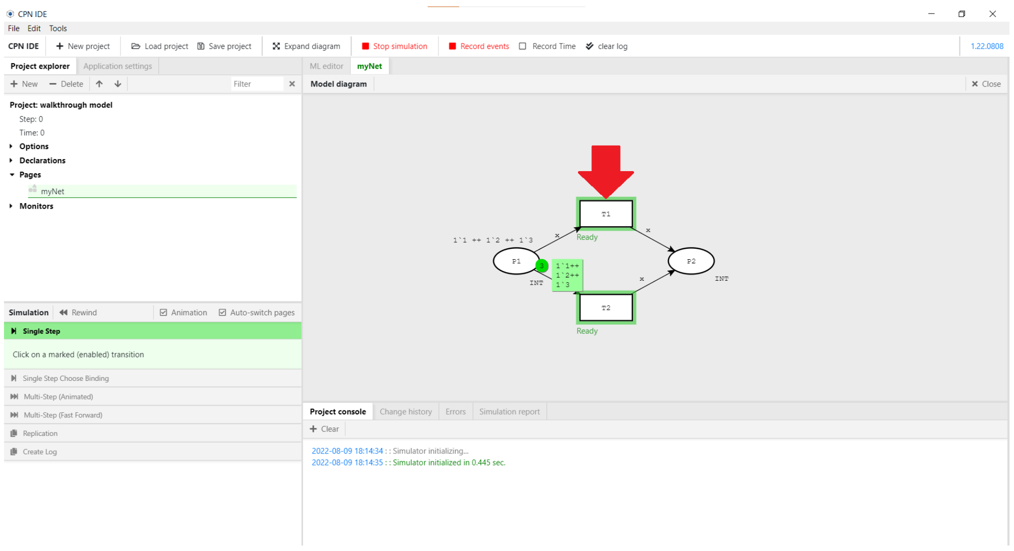
After pressing the `Start simulation` button in Figure 4.6, we are now in the `simulation state` of the model shown in Figure 4.7.



```
2022-08-09 18:13:58 : : Simulator initializing...
2022-08-09 18:13:58 : : Simulator initialized in 0.479 sec.
```

Figure 4.7: The simulation state

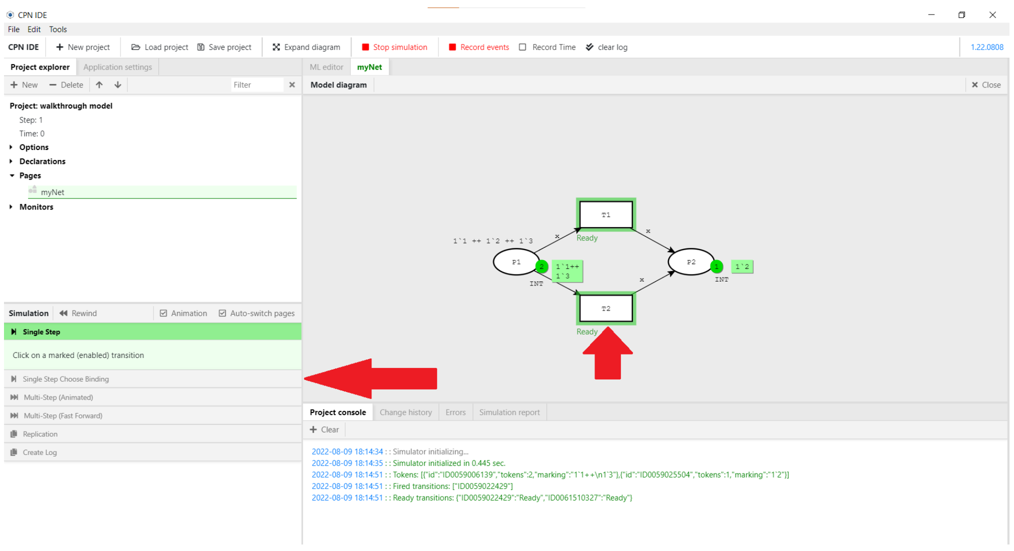
After pressing the `start recording` button in Figure 4.7 we are now in the recording state, shown in Figure 4.8.



```
2022-08-09 18:14:34 : : Simulator initializing...
2022-08-09 18:14:35 : : Simulator initialized in 0.445 sec.
```

Figure 4.8: The recording state

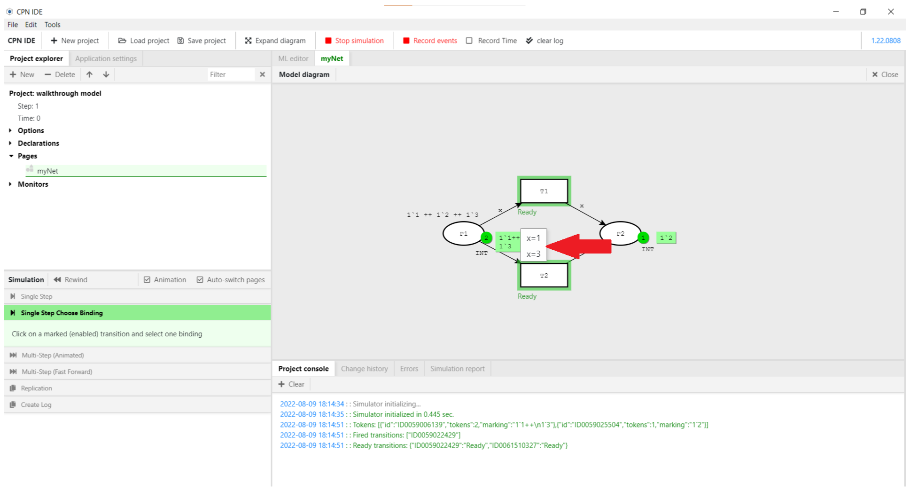
In this recording state we are able to press either T1 or T2 to fire that transition. We choose to fire T1 and thus press the T1 transition.



```
2022-08-09 18:14:34 : : Simulator initializing...
2022-08-09 18:14:35 : : Simulator initialized in 0.445 sec.
2022-08-09 18:14:51 : : Tokens: [{"id":"ID0059006139","tokens":2,"marking":"1`1++\n1`3"},{"id":"ID0059025504","tokens":1,"marking":"1`2"}]
2022-08-09 18:14:51 : : Fired transitions: ["ID0059022429"]
2022-08-09 18:14:51 : : Ready transitions: {"ID0059022429":"Ready","ID0061510327":"Ready"}
```

Figure 4.9: The marking state after firing T1 with the token with color 2

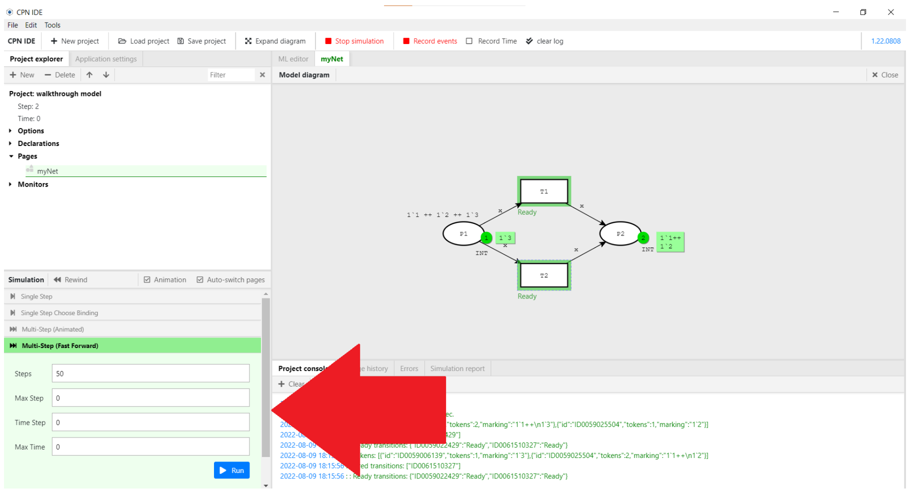
We pressed the transition T1, which fired, as color for `x` the program has selected the color 2. We now want to fire a transition with a specific binding. Therefore we first select the tab `Single step choose binding` and press the `T2` transition.



```
2022-08-09 18:14:34 : : Simulator initializing...
2022-08-09 18:14:35 : : Simulator initialized in 0.445 sec.
2022-08-09 18:14:51 : : Tokens: [{"id":"ID0059006139","tokens":2,"marking":"1`1++\n1`3"},{"id":"ID0059025504","tokens":1,"marking":"1`2"}]
2022-08-09 18:14:51 : : Fired transitions: ["ID0059022429"]
2022-08-09 18:14:51 : : Ready transitions: {"ID0059022429":"Ready","ID0061510327":"Ready"}
```

Figure 4.10: The program displays which color should have the value x

We have pressed the T2 transition, the program asks us to bind the variable x to a color. We choose the color of `x` to be 1, this is shown in Figure 4.10

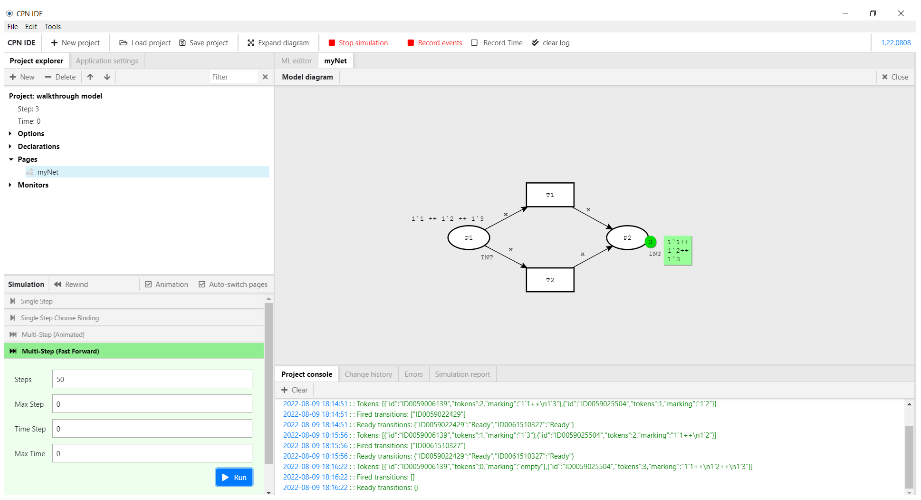


Steps	50
Max Step	0
Time Step	0
Max Time	0

```
ec.
202                                        "tokens":2,"marking":"1`1++\n1`3"},{"id":"ID0059025504","tokens":1,"marking":"1`2"}]
2022-0                                     429"]
2022-08-09                ady transitions: {"ID0059022429":"Ready","ID0061510327":"Ready"}
2022-08-09 18:1           kens: [{"id":"ID0059006139","tokens":1,"marking":"1`3"},{"id":"ID0059025504","tokens":2,"marking":"1`1++\n1`2"}]
2022-08-09 18:15:56       ed transitions: ["ID0061510327"]
2022-08-09 18:15:56 : : Ready transitions: {"ID0059022429":"Ready","ID0061510327":"Ready"}
```

Figure 4.11: The program after firing T2 with color x and pressing the Multi-step fast forward tab

After firing the transition $T2$ with color x. We now want to make the last step randomly. For this we select the `multi-step (fast forward)` tab. This tab allows us to make a number of random moves. We select 50 in this walk-through. This screen is shown in Figure 4.11.



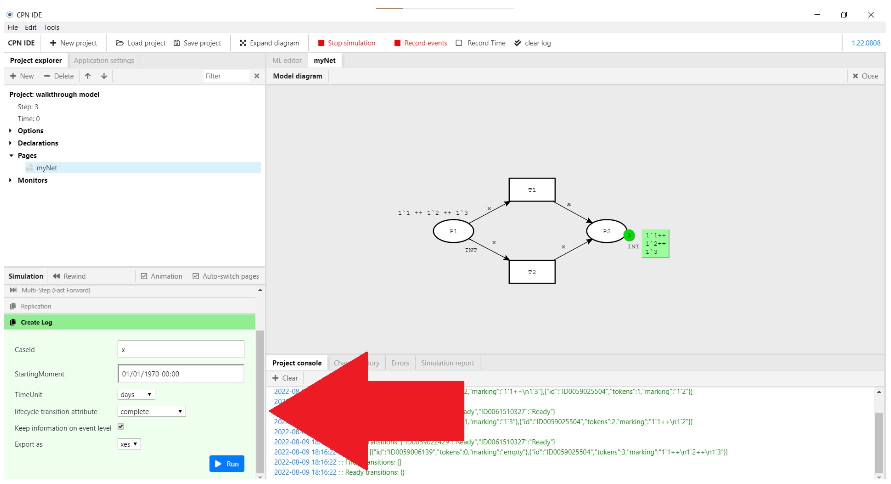
Project: walkthrough model
Step: 3
Time: 0
Options
Declarations
Pages
myNet
Monitors

Steps: 50
Max Step: 0
Time Step: 0
Max Time: 0

```
2022-08-09 18:14:51 : : Tokens: [{"id":"ID0059006139","tokens":2,"marking":"1`1++\n1`3"},{"id":"ID0059025504","tokens":1,"marking":"1`2"}]
2022-08-09 18:14:51 : : Fired transitions: ["ID0059022429"]
2022-08-09 18:14:51 : : Ready transitions: {"ID0059022429":"Ready","ID0061510327":"Ready"}
2022-08-09 18:15:56 : : Tokens: [{"id":"ID0059006139","tokens":1,"marking":"1`3"},{"id":"ID0059025504","tokens":2,"marking":"1`1++\n1`2"}]
2022-08-09 18:15:56 : : Fired transitions: ["ID0061510327"]
2022-08-09 18:15:56 : : Ready transitions: {"ID0059022429":"Ready","ID0061510327":"Ready"}
2022-08-09 18:16:22 : : Tokens: [{"id":"ID0059006139","tokens":0,"marking":"empty"},{"id":"ID0059025504","tokens":3,"marking":"1`1++\n1`2++\n1`3"}]
2022-08-09 18:16:22 : : Fired transitions: []
2022-08-09 18:16:22 : : Ready transitions: {}
```

Figure 4.12: After running the multi-step fast forward

After all the transitions are fired we are done simulating the model as shown in Figure 4.12. We now want to extract a log from this model. To do this we first press the create log tab in the simulation panel.



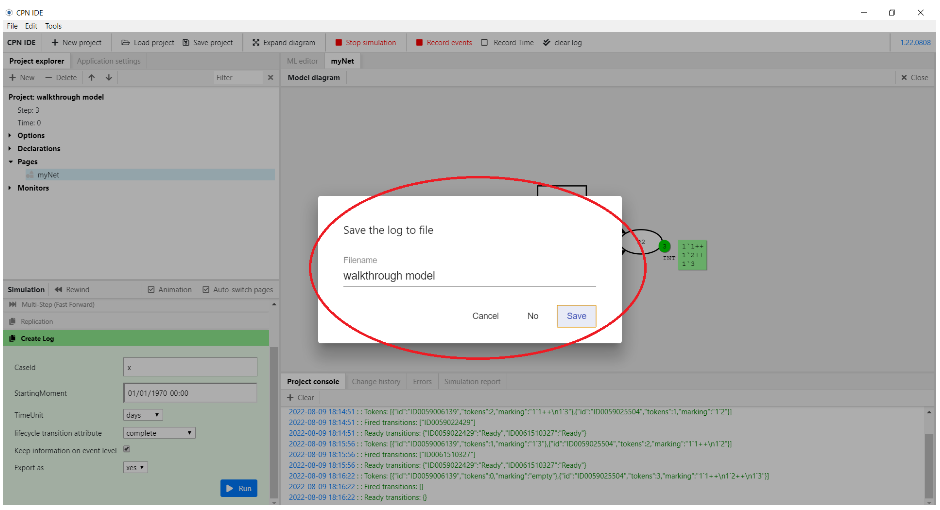
Project: walkthrough model
Step: 3
Time: 0
Options
Declarations
Pages
myNet
Monitors

CaseId: x
StartingMoment: 01/01/1970 00:00
TimeUnit: days
lifecycle transition attribute: complete
Keep information on event level: ✓
Export as: xes

```
2022-08-09 18:16:22 : : Ready transitions: {}
```

Figure 4.13: After opening the create log tab

Next we check the setting for generating the log shown in Figure 4.13, only the caseId setting is important at this point. This value is correct as **x** is the only variable used in the CPN. Then we press the run button to generate a log.

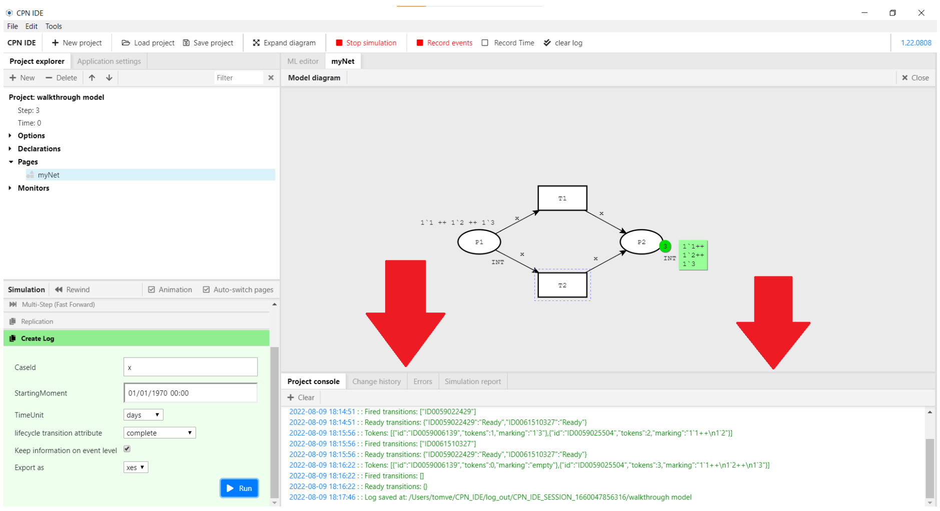


CaseId: x
StartingMoment: 01/01/1970 00:00
TimeUnit: days
lifecycle transition attribute: complete
Keep information on event level: ✓
Export as: xes

```
2022-08-09 18:14:51 : : Tokens: [{"id":"ID0059006139","tokens":2,"marking":"1`1++\n1`3"},{"id":"ID0059025504","tokens":1,"marking":"1`2"}]
2022-08-09 18:14:51 : : Fired transitions: ["ID0059022429"]
2022-08-09 18:14:51 : : Ready transitions: {"ID0059022429":"Ready","ID0061510327":"Ready"}
2022-08-09 18:15:56 : : Tokens: [{"id":"ID0059006139","tokens":1,"marking":"1`3"},{"id":"ID0059025504","tokens":2,"marking":"1`1++\n1`2"}]
2022-08-09 18:15:56 : : Fired transitions: ["ID0061510327"]
2022-08-09 18:15:56 : : Ready transitions: {"ID0059022429":"Ready","ID0061510327":"Ready"}
2022-08-09 18:16:22 : : Tokens: [{"id":"ID0059006139","tokens":0,"marking":"empty"},{"id":"ID0059025504","tokens":3,"marking":"1`1++\n1`2++\n1`3"}]
2022-08-09 18:16:22 : : Fired transitions: []
2022-08-09 18:16:22 : : Ready transitions: {}
```

Figure 4.14: After pressing run on the create log tab

Before generating the log the program first ask us under what name we want to save the generated log as shown in Figure 4.14. The standard value for this is the projectName but the user is able to change the name of the log file. After pressing save the log is generated.



CaseId: x
StartingMoment: 01/01/1970 00:00
TimeUnit: days
lifecycle transition attribute: complete
Keep information on event level: ✓
Export as: xes

```
2022-08-09 18:14:51 : : Fired transitions: ["ID0059022429"]
2022-08-09 18:14:51 : : Ready transitions: {"ID0059022429":"Ready","ID0061510327":"Ready"}
2022-08-09 18:15:56 : : Tokens: [{"id":"ID0059006139","tokens":1,"marking":"1`3"},{"id":"ID0059025504","tokens":2,"marking":"1`1++\n1`2"}]
2022-08-09 18:15:56 : : Fired transitions: ["ID0061510327"]
2022-08-09 18:15:56 : : Ready transitions: {"ID0059022429":"Ready","ID0061510327":"Ready"}
2022-08-09 18:16:22 : : Tokens: [{"id":"ID0059006139","tokens":0,"marking":"empty"},{"id":"ID0059025504","tokens":3,"marking":"1`1++\n1`2++\n1`3"}]
2022-08-09 18:16:22 : : Fired transitions: []
2022-08-09 18:16:22 : : Ready transitions: {}
2022-08-09 18:17:46 : : Log saved at: /Users/tomve/CPN_IDE/log_out/CPN_IDE_SESSION_1660047856316/walkthrough model
```

Figure 4.15: After saving the log

Lastly the program will show in the project console where the generated log is saved as shown in Figure 4.15 This is in an automatic generated folder in which also the simulation reports are shown.

When opening this file in a text editor such as notepad. The file will show the log shown in Listing 4.1

Listing 4.1: A log generated from the model shown in the walk-through

```
1   <?xml version="1.0" encoding="UTF-8" ?>
2   <!-- This file has been generated with the OpenXES library. It conforms -->
3   <!-- to the XML serialization of the XES standard for log storage and -->
4   <!-- management. -->
5   <!-- XES standard version: 1.0 -->
6   <!-- OpenXES library version: 1.0RC7 -->
7   <!-- OpenXES is available from http://www.openxes.org/ -->
8   <log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7">
9           <trace>
10                  <event>
11                          <string key="lifecycle:transition" value="complete"/>
12                          <string key="concept:name" value="T2"/>
13                          <string key="traceId" value="1"/>
14                  </event>
15          </trace>
16          <trace>
17                  <event>
18                          <string key="lifecycle:transition" value="complete"/>
19                          <string key="concept:name" value="T1"/>
20                          <string key="traceId" value="2"/>
21                  </event>
22          </trace>
23          <trace>
24                  <event>
25                          <string key="lifecycle:transition" value="complete"/>
26                          <string key="concept:name" value="T2"/>
27                          <string key="traceId" value="3"/>
28                  </event>
29          </trace>
30  </log>
```

In the remainder of the thesis when discussing generated logs. I will generate the logs the same way I have generated the logs in the walk-through. I will randomly pick which transitions to fire when more than one transition is able to fire, I will keep doing this until there are no transition enabled anymore. In the remainder of the thesis when displaying logs I will also keep out lines 1 through 7 and shorten line 8 of the logs as they will be the same comments for all logs.

Generated logs

The models used in this section of the thesis are used to show that the modules work as expected, the models will show that when the caseId variable is part of the bindings of a fired transition that that event will be recorded and placed in the right trace. In all the model the variable `x` and the variable `y` are of the color set Integer. The caseId variable chosen for each log is `x`. The first four models show that when the caseId variable is part of an arc. No matter in what way that the program records the firing of this transition correctly. The fifth model shows that when the caseId variable changes the recorded log can change. The sixth model shows a model which has a silent transition. The logs created from these models are evaluated in Chapter 6

Simple CPN model

The first CPN model is shown in 4.16. This model is a basic model with one transition and two places. When simulating this model the log shown in Listing 4.2 is generated.

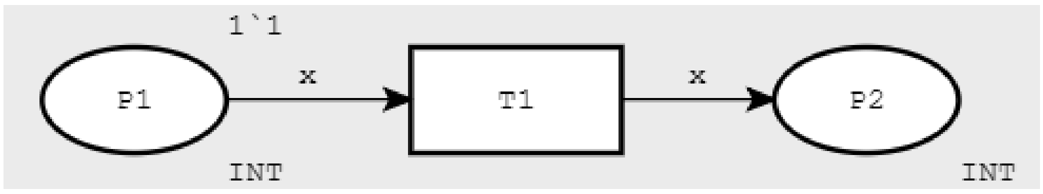


Figure 4.16: Simple CPN model

Listing 4.2: A log generated from the model of Figure 4.16

```
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7">
        <trace>
                <event>
                        <string key="lifecycle:transition" value="complete"/>
                        <string key="concept:name" value="T1"/>
                        <string key="traceId" value="1"/>
                </event>
        </trace>
</log>
```

CPN model with two tokens of the same color needed

The second CPN model is shown in figure 4.17. In this model two tokens with the same color are needed to fire the transition. When simulating this model the same log is generated as when generating a log for the model shown in Figure 4.16.

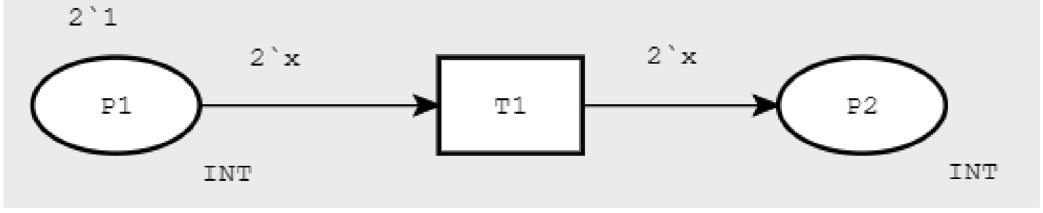


Figure 4.17: CPN model with two tokens of the same color needed

CPN model in which caseId variable is part of an expression

The third model is shown in figure 4.18. In this model `x` is part of an expression on an arc. When simulating the model the same log is generated as when generating a log for the model shown in Figure 4.16 and Figure 4.17.

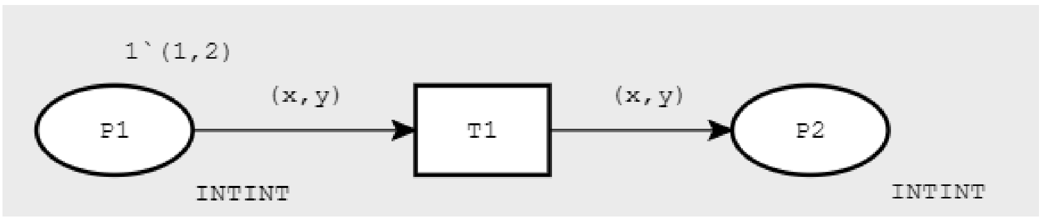


Figure 4.18: CPN model in which caseId variable is part of an expression

CPN model with transition needing (not necessary similar) two tokens

The fourth model is shown in figure 4.19. In for this transition two tokens of color int are requested. One of these colors will be the variable X. Since the value of the caseId variable can be either 1 or 2. This model was simulated ten times. Four times the same log as for the above models was generated. Six times the log shown below was generated.

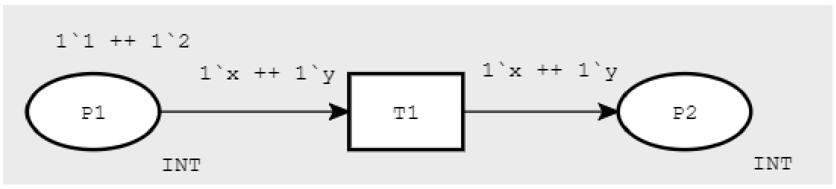


Figure 4.19: CPN model with transition needing (not necessary similar) two tokens

Listing 4.3: XES example log

```
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7">
    <trace>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="2"/>
        </event>
    </trace>
</log>
```

CPN model which can have two different caseId variables

The fifth model is shown in Figure 4.20. This model shows that different logs are generated when selecting a different caseId variable. This model shows a transition `T1` in which both variable `x` and variable `y` are used. Then it shows a transition `T2` in which only variable `x` is used and a transition T3 in which only variable `y` is used. When simulating this model and generating a log with caseId `x` the log shown in Listing 4.4 is generated.

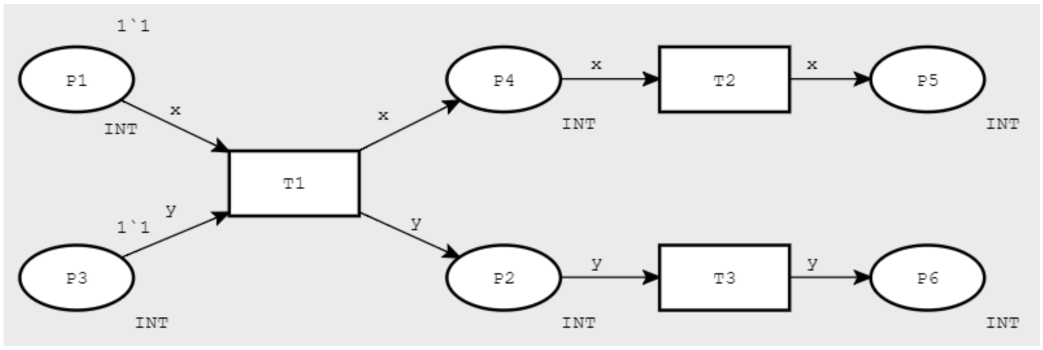


Figure 4.20: CPN model which can have two different caseId variables

Listing 4.4: The log generated from the model shown in Figure 4.20 and `x` as caseId variable

```
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7">
    <trace>
        <event>
            <string key="traceId" value="1"/>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
        </event>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T2"/>
            <string key="traceId" value="1"/>
        </event>
    </trace>
</log>
```

When changing the caseId variable to be `y` and generating a log a different log is produced by the application. The log shown in Listing 4.5 and is produced when selecting `y` as caseId variable.

Listing 4.5: The log generated from the model shown in Figure 4.20 and `x` as caseId variable

```
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7">
    <trace>
        <event>
            <string key="traceId" value="1"/>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
        </event>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T3"/>
            <string key="traceId" value="1"/>
        </event>
    </trace>
</log>
```

CPN model with silent transition

The sixth model is shown in Figure 4.21. This model contains a silent transition, which will not appear in the log. The following log is produced when generating a log from model 4.21.

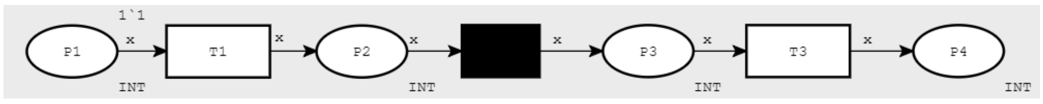


Figure 4.21: CPN model with silent transition

Listing 4.6: A Log generated from the model in Figure 4.21

```
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7">
    <trace>
        <event>
            <string key="traceId" value="1"/>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
        </event>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T3"/>
            <string key="traceId" value="1"/>
        </event>
    </trace>
</log>
```

4.2 Generating a simple event log from a Petri Net

Section 4.1 discussed the first sub-problem which was generating a simple event log from a Colored Petri Net (CPN). In this section we will use the fact that we can create a log from a CPN, to create a log from a Petri Net (PN). To do this we transform the PN into a CPN in such a way that the flow of the Net remains the same and that we can construct a log from the transformed Net.

This section begins by briefly introducing the specific problem we are dealing with in Section 4.2.1. Then Section 4.2.2 explains which modules were created and how they help to solve the problem. Then Section 4.2.3 will discuss in detail the requirements of a PN on which our transformation is successful and lastly Section 4.2.4 will show the result of the features in CPN-IDE.

4.2.1 Problem description

Chapter 4.1 dealt with generating logs from simulations of a CPN by picking a specific variable and making that variable the traceId of the event. A PN does not have variables thus this solution will not work for a PN. Thus we are looking for a solution to give each firing of a transition a traceId. My proposed solution is rebuilding the PN into a CPN. In the newly transformed CPN each place has the color set `TIMEDINT`, this color set is the color set `INT`, but timed. Each arc has the same variable and each token has an unique color. This unique color will be the traceId. When doing this each transition will have its own caseId again thus we are able to use our solution from Chapter 4.1 to generate a log.

The color set `TIMEDINT` was chosen instead of the color set `INT` as we would like the user to have the ability to easily add time to the model, and the only difference between the `TIMEDINT` color set and the `INT` color set is that in the `TIMEDINT` color set the tokens can have a time, this will be further discussed in Chapter 5.2.

The above solution gives each place the `TIMEDINT` color and each token a different value. While this is a solution for generating a log from a PN, this change can influence the flow of the model. Explanation why, requirements for the PN and an example of a model that does not work is given in Section 4.2.3

4.2.2 Added Modules

This section will explain the two modules added to the application; the Petri Net Transformation module and the Token Controller module. These modules are both created for user simplicity they don't add anything to the functioning of the recording modules explained in Chapter 4, they simply make the life of the user easier by transforming multiple components of the model at the same time. This section will also explain how these modules help in creating a CPN suitable for generating a log.

Petri Net Transformation module

The Petri Net Transformation module allows the user to easily change the places and arcs of the Petri Net. This module is built in the front end of the application and is only accessible when in the modelling state of the program. When pressing the `place caseId's` button this module will give a prompt, in this prompt the user is requested to give in a caseId variable. After filling in this caseId variable the program will do the following four things.

1. First the program will create a new color set `TIMEDINT` which is an integer which has a time associated to it (The time part is not important for this part of the thesis but is useful when the PN has time attributes)
2. Second the program will create a new variable, the name of this variable will be the same as the name filled in in the box and as color set the `TIMEDINT` color.
3. thirdly the program will add the `TIMEDINT` color set to all places.
4. fourthly the program will add the new variable to all arcs.

When these changes are applied to the model all the stationary parts of the PN (the PN excluding tokens) will be transformed into a CPN.

Token generator module

The token generator module allows the user to easily change the tokens of the Petri Net. This module is also built in the front end of the application and is only accessible when in the modelling state of the program. When pressing the `set initial marking` button the module will give a prompt. In this prompt the user is requested to give a place of the model and the number of tokens the user wants in this place. After filling in the place and the number of tokens. The program will create new tokens in the chosen place, giving the first token the value 1 and increasing the value with each token generated. For example if the number of tokens requested is ten, tokens with value 1 till 10 will be generated in that place. When after this a new place is given with the same numbers, the tokens 1 till 10 will be generated in that place as well.

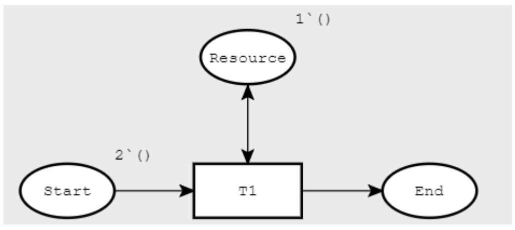
Combining the modules

The Petri Net transformation module allows the transformation of everything but the tokens of a PN, the Token generator module allows the transformation of the tokens of the PN. Combining this the user is able to easily transform the entire PN to a CPN.

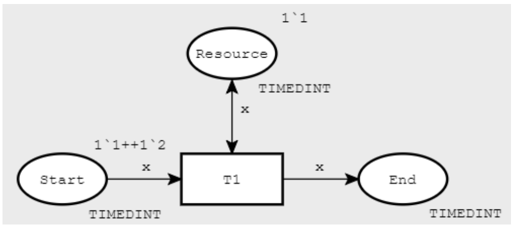
4.2.3 Limitations

While these modules transform the PN to a CPN implementing the solution this way has a couple of limitations. In Section 2.2 it was explained that when a transition has two or more incoming arcs with the same variable on the arc the transition can only be enabled when there are tokens in both places with the same value. Now that each token has a value assigned to it, there will be transitions that were possible in the PN and are no longer possible in the CPN.

The models shown in Figure 4.22 gives an example of a transformation that gives a the CPN created a different flow. The PN is shown in Figure 4.22a, and the CPN is shown in Figure 4.22b. In the PN transition T1, can fire twice, once for both tokens in start. However the CPN can only fire once. Only the token with value `1` in Start can fire. Since the token with color `2` is different from the color `1` in resource while the inscription of both arcs is `x`.



(a) PN version



(b) CPN version

Figure 4.22: An example of a model transformation from a PN to a CPN

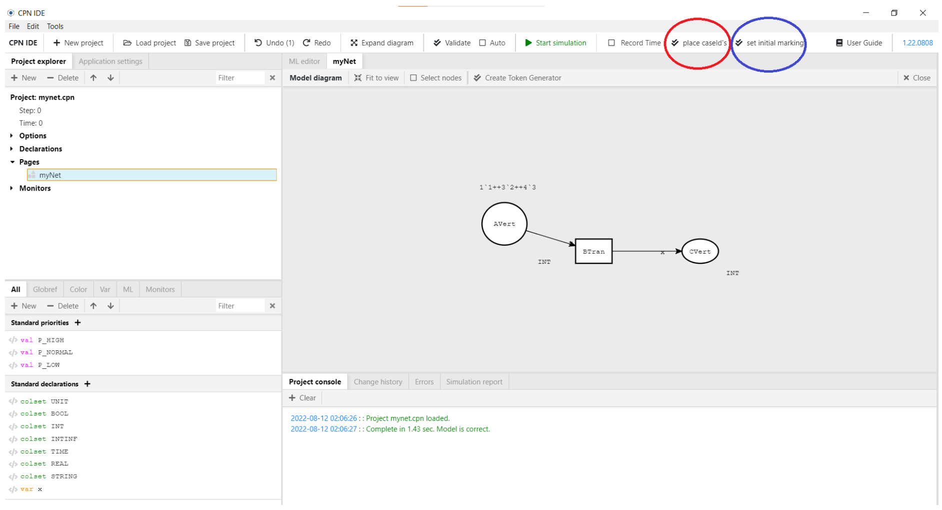
The problem with the created CPN in Figure 4.22b is that the value `x` is not able to bind with both color `2` and color 1. This problem is because in the original PN tokens are all the same, thus there are no restriction on the firing of transitions, while tokens in the transformed PN can only be used in transitions with other tokens that have the same ID. This means that if a CPN has to have the same flow as the original PN, tokens that were able to fire a transition in the original PN should be able to fire a transition in the new PN. This means that a PN can correctly be transformed to a CPN if it follows one of the following two requirements. The first requirement is that the tokens should be all of the same ID (when this is the case the tokens will all be able to fire transitions with each other). When each place of the original PN only has one token in each place, the program will give each token the same ID. The second requirement is that a token should be able to fire all the same transition when it is the only token in the initial marking of the PN (this way we ensure that tokens were not communicating). The first requirement can be easily checked by the user, for the second requirement the user will need some insight in the model. If either of the previous two statements holds the CPN will have the same flow as the original PN and thus be a correct transformation.

4.2.4 Result

This section shows the modules discussed in the previous sections in practice. This section will start with the changes in the GUI and the actions the user can do. Then a small walk-through of how a PN can easily be changed into a PN with the newly added modules is shown. Lastly this section will give some examples of PNs that can be transformed into CPNs and how the transformed PN will look.

Updated GUI

The updated GUI is shown in figure 4.23. The notable changes are encircled in the top bar, there are two new buttons. The first new button is the `place caseId's button` shown with a red circle. Pressing this button will give a pop-up in which the user is able to easily place caseId's on all arcs and color sets on all places as discussed in Section 4.2.2. The second new button is the `set initial marking button` shown with a blue circle. Pressing this button will give a pop-up in which the user is able to easily create unique tokens in a place as discussed in Section 4.2.2.



```
val P_HIGH
val P_NORMAL
val P_LOW
```

```
colset UNIT
colset BOOL
colset INT
colset INTINF
colset TIME
colset REAL
colset STRING
var x
```

```
2022-08-12 02:06:26 : : Project mynet.cpn loaded.
2022-08-12 02:06:27 : : Complete in 1.43 sec. Model is correct.
```

Figure 4.23: The GUI with the PN modules added

Walk-through

This thesis will give a walk-through on how to transform a PN into a CPN. The model chosen for this example is shown in Figure 4.24. During the walk-through red arrows will highlight the part of the program to focus on.

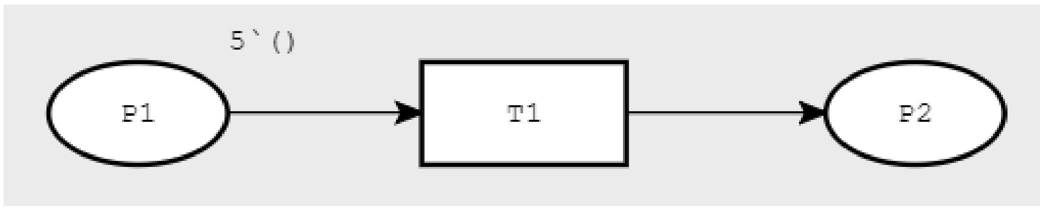
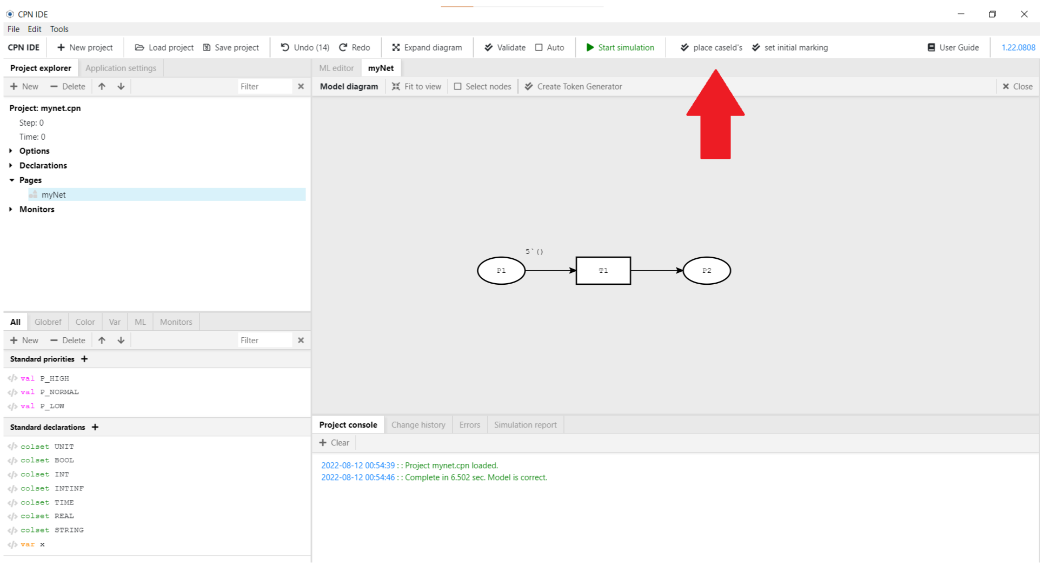


Figure 4.24: PN model of walk-through

We start the walk-through in the modelling state of the program in Figure 4.25. In this state we press the `place caseId's button`.



```
val P_HIGH
val P_NORMAL
val P_LOW
```

```
colset UNIT
colset BOOL
colset INT
colset INTINF
colset TIME
colset REAL
colset STRING
var x
```

```
2022-08-12 00:54:39 : : Project mynet.cpn loaded.
2022-08-12 00:54:46 : : Complete in 6.502 sec. Model is correct.
```

Figure 4.25: Initial modelling state

After pressing the `place caseId's button`, the program will open a pop-up screen as shown in Figure 4.26. In this pop-up screen the user is able to choose a caseId value, this value will be placed on all arcs.

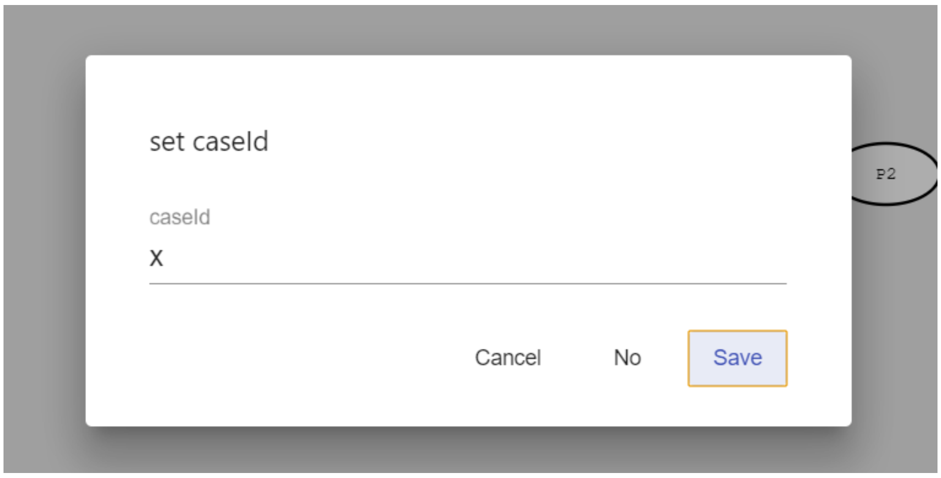
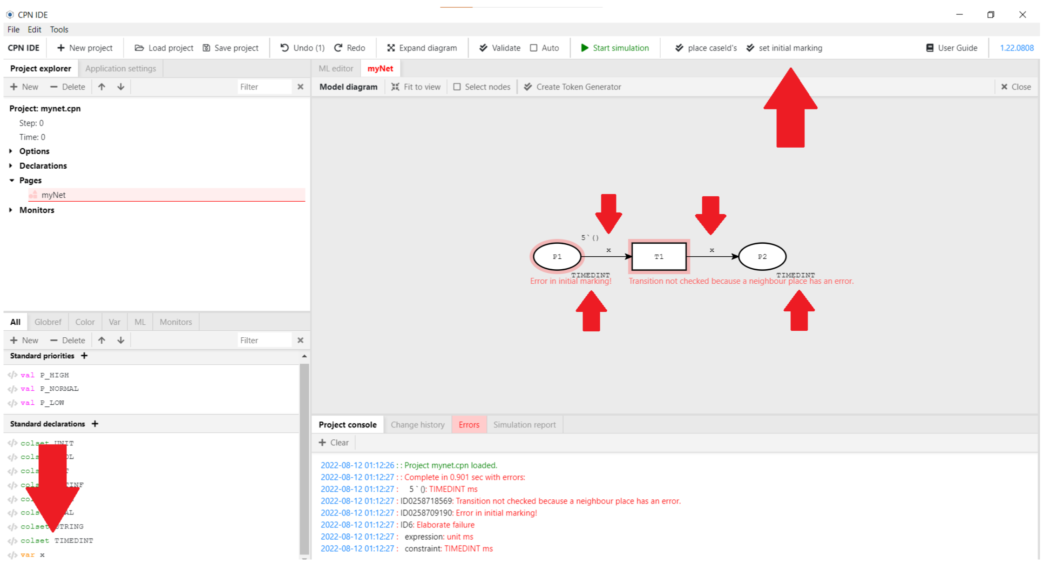


Figure 4.26: `place caseId` pop-up screen

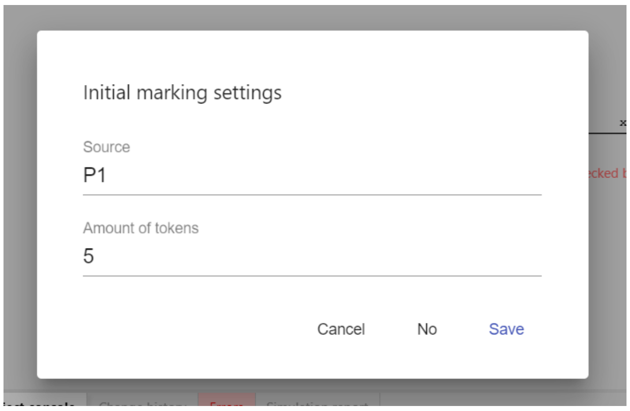
After pressing save in the previous screen the program will return to the original screen as shown in Figure 4.27. The program will have created the TIMEDINT color set and a variable of the color set TIMEDINT, this variable is the variable filled in for the caseId value (in our example this is x). Also the colorset of all places has changed and the arcs now have the chosen variable as expression on it. The model now gives an error because the original marking is no longer correct. We will change this by pressing the `set initial marking` button.



```
2022-08-12 01:12:26 : : Project mynet.cpn loaded.
2022-08-12 01:12:27 : : Complete in 0.901 sec with errors:
2022-08-12 01:12:27 :     5 ` (): TIMEDINT ms
2022-08-12 01:12:27 : ID0258718569: Transition not checked because a neighbour place has an error.
2022-08-12 01:12:27 : ID0258709190: Error in initial marking!
2022-08-12 01:12:27 : ID6: Elaborate failure
2022-08-12 01:12:27 :   expression: unit ms
2022-08-12 01:12:27 :   constraint: TIMEDINT ms
```

Figure 4.27: Modelling state after applying `place caseId` changes

After pressing the `set initial marking button` the program will open a pop-up screen as shown in Figure 4.28. In this pop-up the user is able to select a place and choose how many tokens we want in that place. As we had five tokens in $P1$ in the initial model we select these values.



Initial marking settings

Source
P1

Amount of tokens
5

Figure 4.28: The `set initial marking` pop-up menu after filling in the values

After pressing save the model will have created five tokens with unique colors (1 through 5) in P1. We have now transformed a PN into a CPN. The newly constructed PN is shown in Figure 4.29

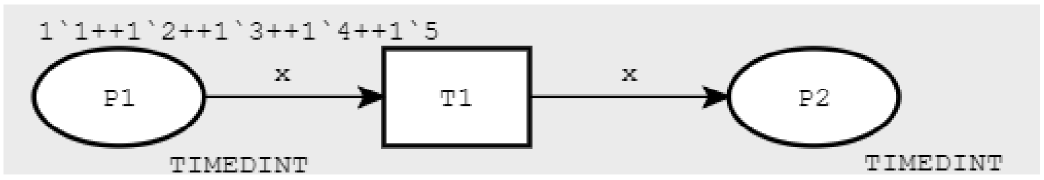


Figure 4.29: The model after all changes have been applied

Transforming PN into CPNs

The models used in this section of the thesis are used to show that the flow of the model does not change when keeping the established requirements of a PN in mind when transforming the PN to a CPN. For both requirements models will be given that follow these requirements. We argue that since it works on these models, it will work on all models. We argue this because, while there can be many more transitions in models, these models will have the same foundation as the models shown in this section, namely the firing of a transition. Given that both the examples shown in this section and the more elaborate models both use the transitions and the same firing rules we argue that the transformation will work for all models that adhere to the requirements.

PN model with one token in each place

The first model is shown in Figure 4.32. This model is a simple model that has two tokens in P1. However the tokens do not interact and the execution does not change when the token is on their own. First a T1 fires with one token from P1. Then T1 fires again with the other token from T1. The tokens themself do not interact with any of the other tokens. When transforming the model to a CPN the model is transformed to the model shown in 4.33. When simulating this model the log shown in Listing 4.8 is generated.

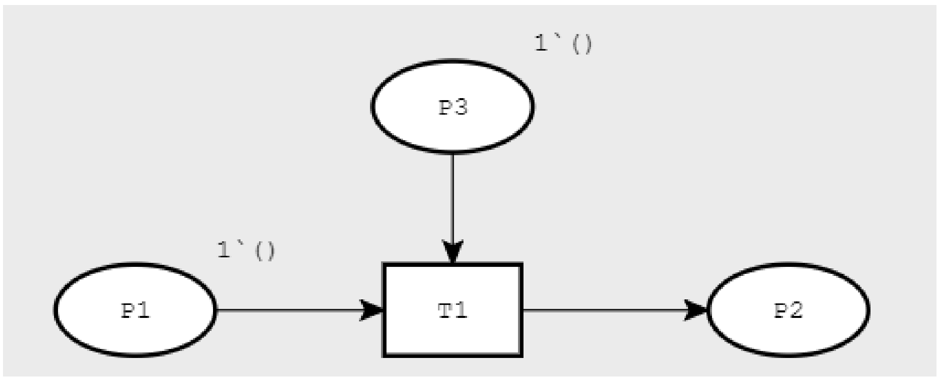


Figure 4.30: PN with one token in each place

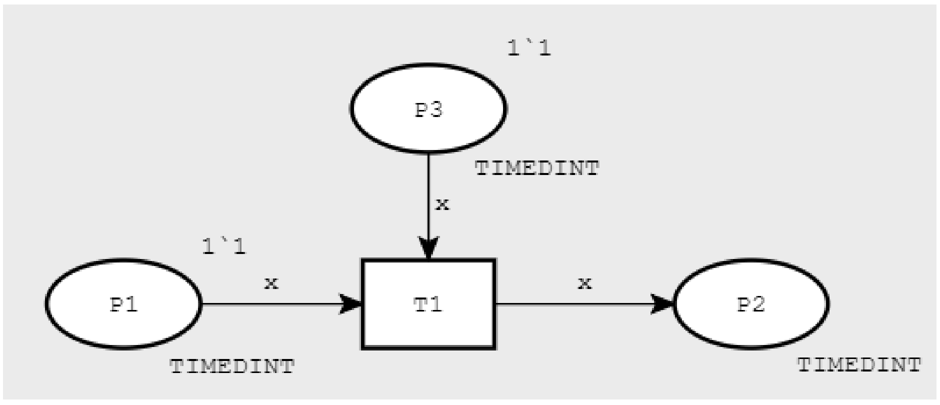


Figure 4.31: Transformed PN with one token in each place

Listing 4.7: XES example log

```
<log>
        <trace>
                <event>
                        <string key="lifecycle:transition" value="complete"/>
                        <string key="concept:name" value="T1"/>
                        <string key="traceId" value="1"/>
                </event>
        </trace>
</log>
```

PN model with tokens that don't interact

The second model is shown in Figure 4.30. This model is a simple model that has one token in each place. When transforming the model to a CPN the model is transformed to the model shown in 4.31. Since both tokens are of the same color transition T1 is still able to fire. When simulating this model the log is shown in Listing 4.7.

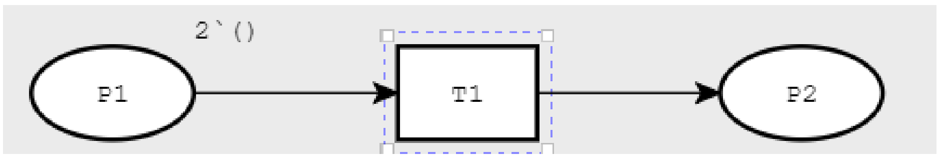


Figure 4.32: PN with multiple token in one place

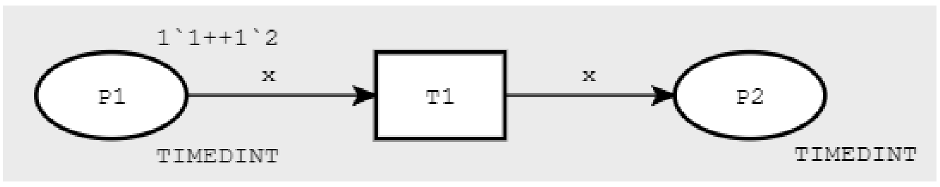


Figure 4.33: Transformed PN with multiple token in one place

Listing 4.8: XES example log

```
<log>
    <trace>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="1"/>
        </event>
    </trace>
    <trace>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="2"/>
        </event>
    </trace>
</log>
```

4.2.5 Creating a log from a PN

Chapter 4.1 explained how a log can be created from a CPN, Chapter 4.2 explained how a PN can be transformed into a CPN if it has some specific properties. If we apply the modules created in Chapter 4 to a CPN created from the modules in chapter 4.2 then we are able to generate a log from a PN.

Chapter 5

Generating realistic logs

In Chapter 4 the problem of generating a simple event log conforming to the XES Standard from a Colored Petri Net was solved. The solution to that problem will be the basis of the second research problem. This research problem deals with extending the basic log, and creates a more realistic log which can be found in everyday systems. This chapter will make logs more realistic by adding three features to the log. The first feature which will be added is a lifecycle-transitions attribute, this module will be explained in Section 5.1. The second feature which will be added is a timestamp attribute. The Time module will be explained in Section 5.2. The third feature that will be added is the resource feature, the resource module will be explained in Section 5.3.

5.1 Adding lifecycle-transitions

In Chapter 4 a simple log was generated from any CPN. This result will be the basis for the feature added in this section. The feature added in this section will be the lifecycle-transition attribute (LTA) for events. This attribute tells something about the lifecycle of the event as explained in Section 2.5. Is the event portraying the beginning of an activity, the end of the activity or something in between.

This section will first explain the module added to the application, then it will explain how the module is used and what kind of logs we are now able to generate.

5.1.1 Modules

The lifecycle module allows the user to add lifecycle-transitions attributes (LTAs) to the log in two ways. One way is the automatic way in which the program itself adds the LTA to the event and will be discussed in Section 5.1.1, the second way the program allows the user to add LTAs to events is the manual way in which the user is allowed to select for each transition which lifecycle it has, this part of the module will be explained in Section 5.1.1. Then the result of this implementation will be shown in Section 5.1.2. Reason for this option is because I want users to have the ability to construct any log they want. By implementing lifecycle-transitions this way the user is always able to pick the lifecycle-transitions they want by manually adding them. Also I gave the option to automatically add the most common lifecycle-transition if the user simply wants a lifecycle-transition, but the user want to have the same lifecycle-transition everywhere.

Manually adding lifecycle-transitions

The lifecycle module also allows the user to manually enter the LTA for each transition. For the LTA value the option `in transition name` is selected. Initially the program will not output any LTA for each event when this option is chosen. When picking this option the user should identify for each transition what the intended LTA is and add that LTA to the transition name. For example say we have a transition `cook` and we want to give this transition the LTA `resume`. We now change the transition name to be `cook + resume`. This way the application knows the activityName is `cook` and the LTA is `resume`. Instead of the LTA `resume` the user is able to choose any LTA as given in figure 5.1.

Automaticity adding lifecycle-transitions

The lifecycle module allows the user to enter a preset LTA value. This value will then be added to all events as the LTA. There are three possible options for this value. The first option is the `start` LTA, this value resembles the start of an activity. The second option is the `complete` LTA, this value resembles the end of an activity. The final option is a combination of both the `start` LTA and the `complete` LTA. In this combination there are two events created from each transition that fires, one of the events has the `start` LTA and the second event has the `complete` LTA. This implementation allows the user to pick a common LTA for each event.

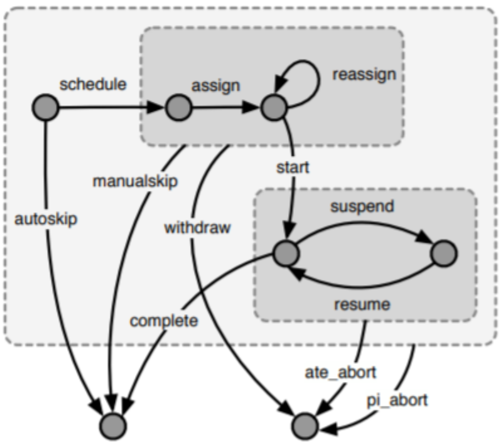


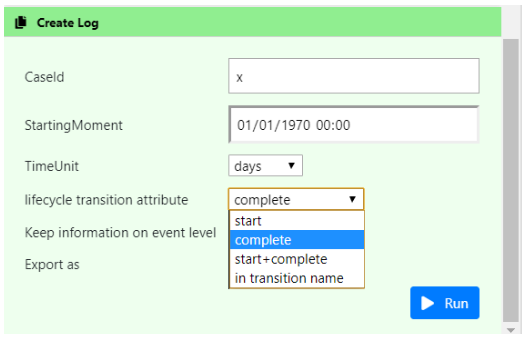
Figure 5.1: The lifecycle model

5.1.2 Result

This section shows the result of the implementation of the modules discussed in the previous sections. To do this this section will start with the changes to the GUI and the actions the user is able to do. Then this section will give some examples of CPN with lifecycle-transition attributes.

Updated GUI

The updated `GUI` is shown in Figure 5.2. The only notable change is in the `create log` panel. In this panel the option to select a lifecycle-transition attribute is added, this is done by the use of a dropdown menu. The possible options are `complete`, `start & complete`, `start` and `in transition name`.



Create Log

CaseId	x
StartingMoment	01/01/1970 00:00
TimeUnit	days
lifecycle transition attribute	complete
Keep information on event level	
Export as	

Figure 5.2: Screenshot of the create log panel

Models

The models used in this section of the thesis are used to show that the all the modules discussed in the previous section work as they should. This includes all options for th lifecycle-transition attribute as well as lifecycle-transition attributes in the name of the transition. For all combinations of options and transition Name a model will be shown.

CPN with manually added LTA

The second LTA model is shown in Figure 5.3. This model is a model with two transitions which have the LTA in the transition name. We will use this model to check whether the manual LTA module works as expected. We will also be setting the LTA option to complete to show that when an automatic option is selected the transition name does not matter. When the `complete` LTA option is chosen the program will generate the module shown in 5.1. When the `in transition name` option is chose the program will generate the module shown in 5.2

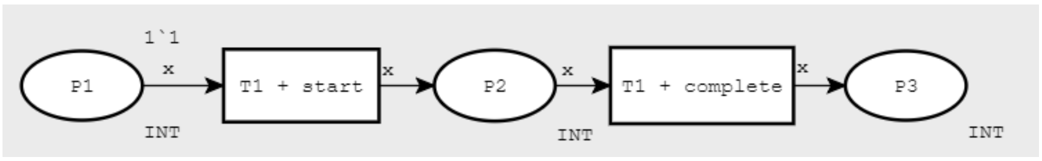


Figure 5.3: CPN with manually added LTA

Listing 5.1: Log generated from the model shown in 5.3 and LTA `complete`

```
<log>
    <trace>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1 + start"/>
            <string key="traceId" value="1"/>
        </event>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1 + complete"/>
            <string key="traceId" value="1"/>
        </event>
    </trace>
</log>
```

Listing 5.2: Log generated from the model shown in 5.3 and LTA `in transition name`

```
<log>
    <trace>
        <event>
            <string key="lifecycle:transition" value="start"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="1"/>
        </event>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="1"/>
        </event>
    </trace>
</log>
```

CPN model without particular LTA attributes in the transition name

The first LTA model is shown in Figure 5.4. This model is the basic model with one transition and two places. This model is used to show that the automatic selected lifecycle-transition module works. We will use this model to generate a log with the lifecycle-transition being `complete`, `start` and `start and complete`. When simulating this model with those lifecycle-transitions attribute the following three logs are generated.

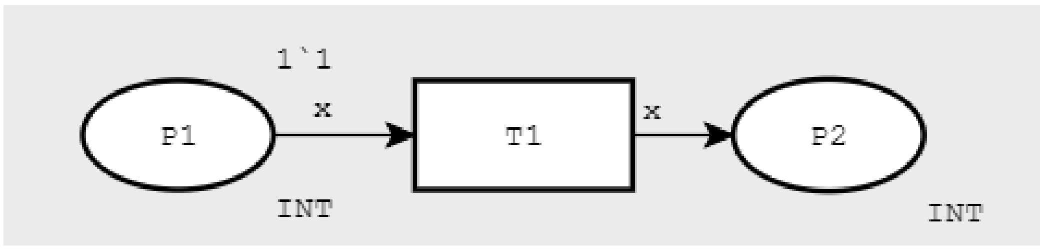


Figure 5.4: CPN model without LTA attributes

Listing 5.3: Log generated from the model of figure 5.4 with LTA `complete`

```
<log>
    <trace>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="1"/>
        </event>
    </trace>
</log>
```

Listing 5.4: Log generated from the model of figure 5.4 with LTA `start`

```
<log>
    <trace>
        <event>
            <string key="lifecycle:transition" value="start"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="1"/>
        </event>
    </trace>
</log>
```

Listing 5.5: Log generated from the model of figure 5.4 with LTA `start + complete`

```
<log>
    <trace>
        <event>
            <string key="lifecycle:transition" value="start"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="1"/>
        </event>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="2"/>
        </event>
    </trace>
</log>
```

CPN model with incorrect LTA attribute

The third LTA model is shown in Figure 5.5. This model is a model with one transition. This model is used to show that when selected `in transition name` for the lifecycle attribute and the string after the `+` does not contain a valid lifecycle-transition that the model will not identify this string as a lifecycle-transition and simply add it to the activityName. The log generated from this model is shown in Listing 5.6

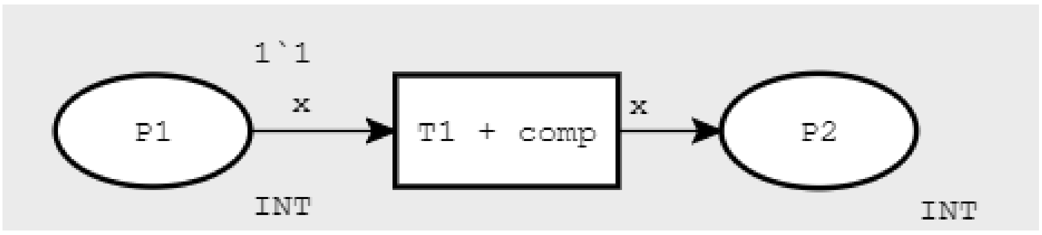


Figure 5.5: CPN model with incorrect LTA attribute

Listing 5.6: Log generated from the model of figure 5.5 with LTA `in transition name`

```
<log>
    <trace>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1 + comp"/>
            <string key="traceId" value="1"/>
        </event>
    </trace>
</log>
```

5.2 Adding Time

Chapter 5.1 added lifecycle-transitions to the log. We will use the functionality after the added modules in Chapter 5.1 as starting point for the added functionality in this Chapter. We take this as starting point as we want to use the lifecycle attribute when determining the time of a transition.

The feature added to the log in this chapter will be time of events. This attribute tells something about the date and time the event happened in real life. The basis for this will be the already built in time function of CPN-IDE which is built upon the functionality of Timed Colored Petri Nets (TCPNs) as explained in Section d2.3. This chapter will explain the module built to record the already existing time in CPN-IDE and the module built to transformed this time into a real life time. In these module we will be building on the logs generated in 5.1. This includes that we need to associate a time to an event with lifecycle-transition attribute (LTA) `start`, which should be the start time of the event and LTA `complete`, which should be the end time of the event.

This section begins with explaining certain design decision made to determine the time of events. Then this section will explain the modules added to CPN-IDE; the TimeLog module and the TokenController module. These modules will be explained in Section 5.2.1. Then Section 5.2.2 will explain how these modules work together to generate the correct time. Lastly the result of the implemented modules will be shown in 5.2.2

The design decision was made to make the time dependent on the lifecycle-transition attribute, when the lifecycle-transition attribute is `In transition name` or `start` then the fire time of the transition will be the time of the transition. When the lifecycle-transition attribute is `complete`, we don't want the time at the beginning of the transition. We want the time at the end of the transition. To find this time we looked at all the tokens produced from the firing of that transition and pick the end time of the transition as the earliest time of any of the produced tokens. I confirmed this design decision with my supervisors and we believe that this implementation gives us the most expressing power while remaining easy to configure when generating a log.

5.2.1 Modules

Two modules are added to the application to be able to add time to the logs. The first module is the TimeLog module, this module is created to transform the time from the simulation to a real life time and add this time to the log. The second module is the TokenController module, this module is created to construct an end time of the firing of a transition in the simulation.

Time log module

The time log module is built to transform the simulator time into a real life time and add this time to the log. The module uses the simulator time and two user inputs; the `StartingMoment` and the `TimeUnit`. The StartingMoment input is used to set a begin date and time for the log, for example `01/01/1970 00:00`. The date and time chosen in this input are equal to time 0.0 in the simulation. The TimeUnit input is used to determine how time in the simulation relates to time in the real world. Time in the simulation is done in real numbers. The time log module allows the user to select how much time one time in the simulator takes. The user can pick from the following options; years, months, weeks, days, hours or even minutes. These two user inputs allows the user to add time to the log any way the user would like without having to change the original model (as the TimeUnit input can be adjusted to suit the model).

TokenController module

The TokenController module is built to get the end time of fired transitions. Section 5.1.1 added the possibility to automatically add lifecycle-transitions to the log, this includes the option to both generate an event with the LTA `start` and with the LTA `complete` from the same firing of a transition. However due to the fact that CPN-IDE allows transitions to take a non-deterministic amount of time, the end-time of activities is not always trivial. To solve this problem the TokenController module was created. The TokenController keeps track of the time and position of each token. The module does this by using the `ReturnTokensAndMarking` function of the simulator to request the current marking of the simulation after the firing of each transition in the simulation. From this marking the tokens and the times of these tokens are extracted. Then the tokens in the new marking are compared to the tokens in the old marking. All the already existing tokens are filtered out using this method leaving only the newly produced tokens. Then the lowest time of these tokens is calculated and send back to the recording module, which will use this time in the generation of the log. This module assumes that the earliest time of any newly produced token is the end time of the firing of the transition. Thus for this module to work there should be at least one outgoing arc for each transition that does not take time. Since this module uses a lot of processing power, the module can be turned on or of by using the `record time` switch.

5.2.2 Result

The TokenController module explained in Section 5.2.1 allows the program to generate the end time of fired transitions. The time log module explained in Section 5.2.1 allows the program to transform the simulation time into a realistic time. These two additions to the program allow the user to generate a realistic time for each event in the generated logs from 5.1 as we are able to generate a simulation time for the `Complete` lifecycle-transition attribute and we are able to transform the simulator time into a realistic time.

This section shows the result of the implementation discussed in the previous section. To do this this section will start with the changes to the GUI and the actions the user is now able to do. Then this section will give some examples of TCPNs in which the functionality of the added modules is evaluated.

Updated GUI

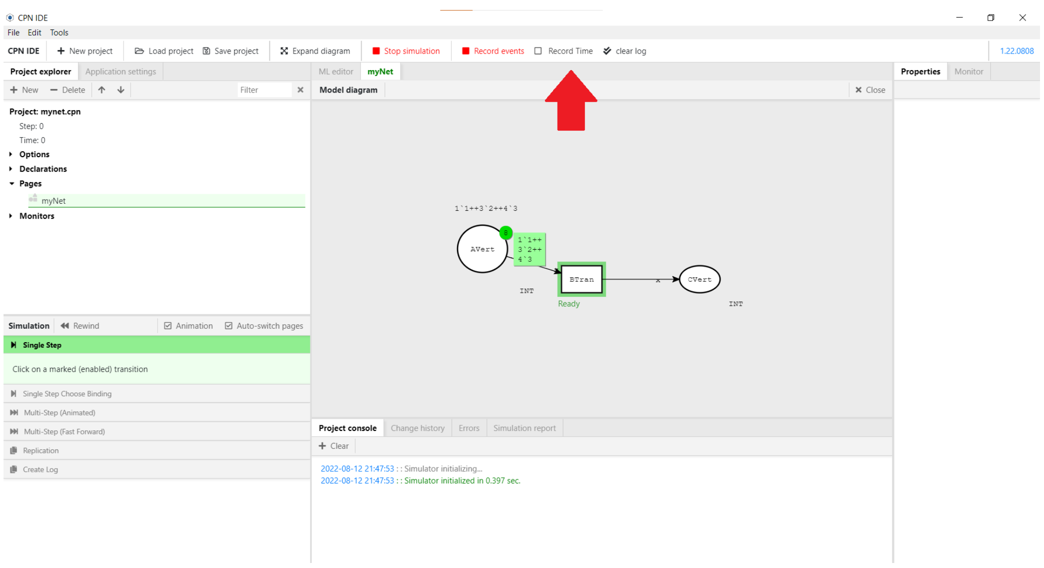
The GUI is updated in two ways with the module. The first update is in the top bar as shown in figure 5.6. The `Record Time` button is added to the top bar, when this button is pressed the log will now record time as well as bindings. This is a seperate option and not built in with the recording to save computational power when time is not of importance. The second update is in the create log tab and is shown in Figure 5.7. There are two field added a `StartingMoment` field, indicating the time in the real world equivalent to the simulator time 0 and a `TimeUnit` dropdown menu, the user can select in this dropdown menu what one timeunit in the simulator relates to; one minute, hour, day, week, month or year.

Models

The models used in this section of the thesis are used to show that the all the modules discussed in the previous section work as they should. This includes all options for the `TimeUnit` variable as well as different options for the `StartingMoment` dropdown menu as well as different options for the lifecycle-transition attribute. The models are picked in such a way that in the end all functionality of Time added to the logs is shown.

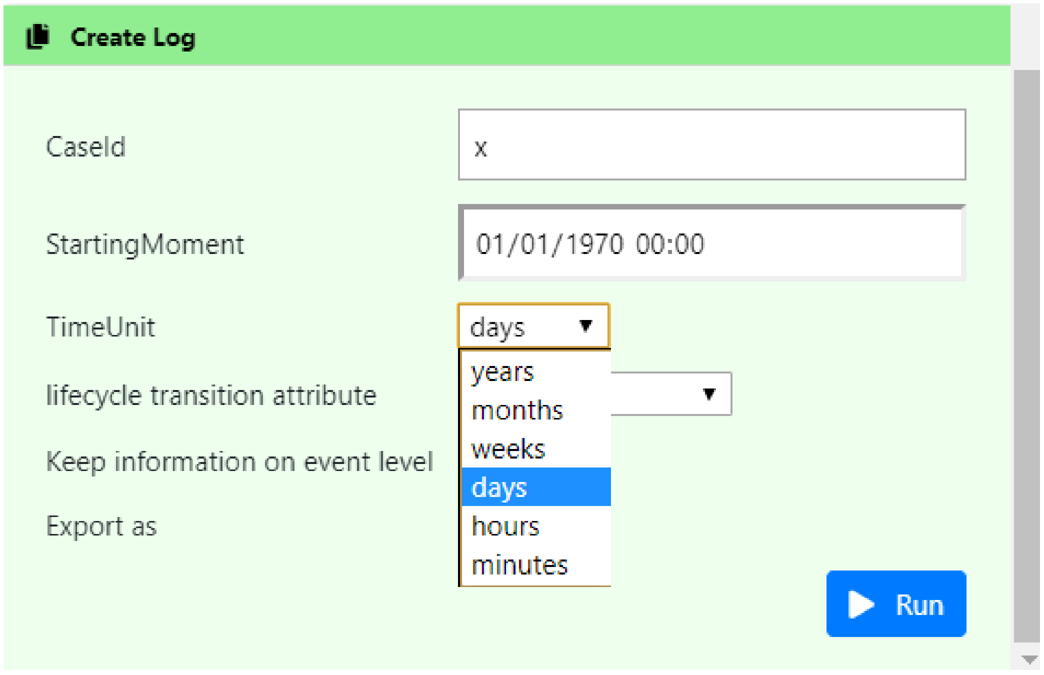
Generic time TCPN

The first time model is shown in Figure 5.8. I will use this model to show that time is recorded with the activity, I will show that the TimeUnit functionality works, I will show that the StartingMoment functionality works and I will



```
2022-08-12 21:47:53 : : Simulator initializing...
2022-08-12 21:47:53 : : Simulator initialized in 0.397 sec.
```

Figure 5.6: An updated GUI after the time module was added



CaseId	x
StartingMoment	01/01/1970 00:00
TimeUnit	days
lifecycle transition attribute	
Keep information on event level	
Export as	

Figure 5.7: The update create log panel

show that the complete LTA value has the timestamp of the end of the activity. I will first generate a log with as StartingMoment 01/01/1970 at 00:00, with TimeUnit minutes. Then I will change the timeunit value until I have gone through all values. These logs will only differ in the time attribute, thus I will only show this attribute. Then I will change the StartingMoment of the log to show that changing this variable also works.

The log generated from the model in Figure 5.8 is shown in Listing 5.7. Logs generated with other TimeUnit attribute values are shown in Listings 5.8, 5.9, 5.10, 5.11 and 5.12. Next part of a log with the startingMoment 02/02/2002 at 02:02 and TimeUnit days is shown in Listing 5.14. Lastly the log with LTA `start` and TimeUnit `minutes` will be shown in Listing 5.13. In the caption of the listing the thing that is different from the first listing will be displayed

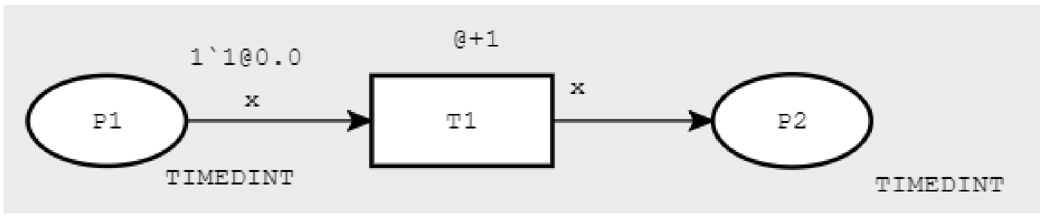


Figure 5.8: A model with time

Listing 5.7: Log from model 5.8 with SM 01/01/1970 00:00 and TU minutes and LTA complete

```
<log>
    <trace>
        <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="1"/>
            <date key="time:timestamp" value="1970-01-01T00:01:00.000+01:00"/>
        </event>
    </trace>
</log>
```

Listing 5.8: Line 7 from listing 5.7 but with TU hours

```
<date key="time:timestamp" value="1970-01-01T01:00:00.000+01:00"/>
```

Listing 5.9: Line 7 from listing 5.7 but with TU days

```
<date key="time:timestamp" value="1970-01-02T00:00:00.000+01:00"/>
```

Listing 5.10: Line 7 from listing 5.7 but with TU weeks

```
<date key="time:timestamp" value="1970-01-08T00:00:00.000+01:00"/>
```

Listing 5.11: Line 7 from listing 5.7 but with TU months

```
<date key="time:timestamp" value="1970-02-01T00:00:00.000+01:00"/>
```

Listing 5.12: Line 7 from listing 5.7 but with TU years

```
<date key="time:timestamp" value="1971-01-01T00:00:00.000+01:00"/>
```

Listing 5.13: Log shown in Listing 5.7 but with LTA start

```
<log>
    <trace>
        <event>
            <string key="lifecycle:transition" value="start"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="1"/>
            <date key="time:timestamp" value="1970-01-01T00:00:00.000+01:00"/>
        </event>
    </trace>
</log>
```

Listing 5.14: Line 7 from listing 5.13 but with SM 02/02/2002 02:02

```
<date key="time:timestamp" value="2002-02-02T02:02:00.000+01:00"/>
```

TCPN model with lifecycle-transitions in transition name

The second model is shown in Figure 5.9. I will use this model to show that when the lifecycle-transition attribute is in the name, then the time of that event will simply be the firing time of the transition. The log shown from generating a log with LTA `In transition name`, TimeUnit `minutes` and StartingMoment `01/01/1970 00:00` is shown in Listing 5.15.

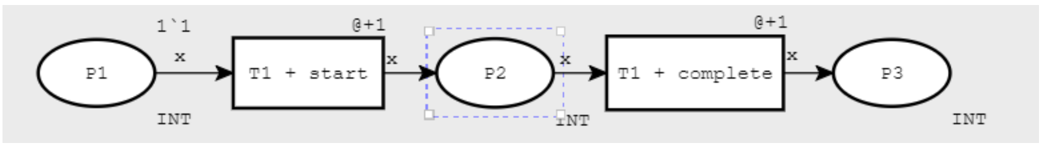


Figure 5.9: TCPN model with lifecycle-transitions in transition name

Listing 5.15: The log from the model shown in 5.9 and the above mentioned configuration settings

```
<log>
    <trace>
        <event>
            <string key="lifecycle:transition" value="start"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="1"/>
            <date key="time:timestamp" value="1970-01-01T00:00:00.000+01:00"/>
        </event>
                <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="1"/>
            <date key="time:timestamp" value="1970-01-01T00:01:00.000+01:00"/>
        </event>
    </trace>
</log>
```

TCPN model with non deterministic time and two output arcs and two input arcs

The third model is shown in Figure 5.10. This model is used to show that even when the time is non-deterministic we are still able to calculate the correct end-time from the log. To do this we will show the current marking after

firing transition T1. This model also shows that the time of the transition is the lowest of the tokens consumed when firing the transition.
As the time in this model is non deterministic we need to know the time in the simulation after transition T1 has fired. We need this time of the simulation to check if the time for the transition is correctly calculated. The current marking including times is shown in Figure 5.11, the time after firing T1 in the simulation is *0.391483647815*. The generated log from the simulation with LTA `start+complete`, TimeUnit `hours` and StartingMoment `01/01/1970 00:00` of this particular run of this model is shown in 5.16, note that this log is not reproducible as the time is non deterministic.

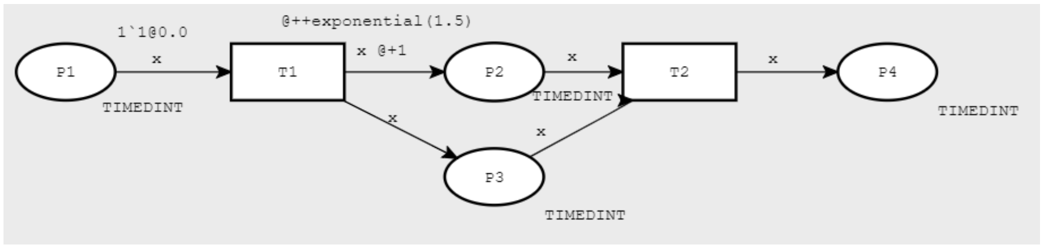


Figure 5.10: TCPN model with non deterministic time and two output arcs and two input arcs

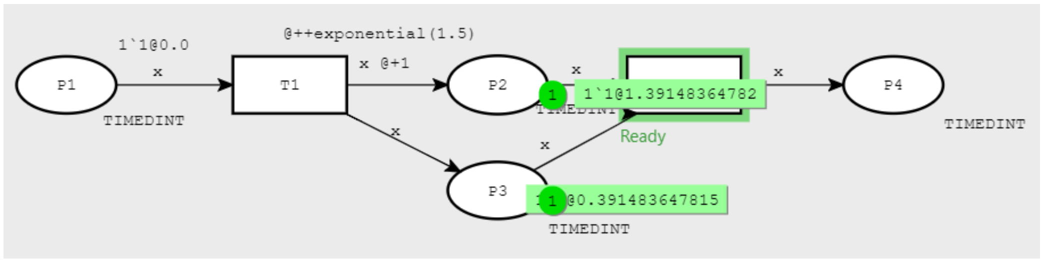


Figure 5.11: TCPN model with non deterministic time and two output arcs and two input arcs

Listing 5.16: A Log generated from TCPN model with non deterministic time and two output arcs and two input arcs

```
<log>
    <trace>
        <event>
            <string key="lifecycle:transition" value="start"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="1"/>
            <date key="time:timestamp" value="1970-01-01T00:00:00.000+01:00"/>
        </event>
                <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
            <string key="traceId" value="1"/>
            <date key="time:timestamp" value="1970-01-01T00:00:23.489+01:00"/>
        </event>
                <event>
            <string key="lifecycle:transition" value="start"/>
            <string key="concept:name" value="T2"/>
            <string key="traceId" value="1"/>
            <date key="time:timestamp" value="1970-01-01T00:01:23:489+01:00"/>
        </event>
                <event>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T2"/>
            <string key="traceId" value="1"/>
            <date key="time:timestamp" value="1970-01-01T00:01:23:489+01:00"/>
        </event>
    </trace>
</log>
```

If these modules all show a correct log we will have checked all the possibilities of added functionality to the log, thus we will have correctly added time to the log.

5.3 Adding resources

Chapter 5.1 added time to the generated log, Chapter 5.2 added time to the generated log. This chapter focusses on adding resources to the log. Resources are tokens used for the firing of the transition that are not the token identifying the caseId. The model in Figure 5.12 shows an example of a resource. In this model `x` is the caseId, to fire transition T1 a token of the color set `Machine` from place P3 is needed. We would like to record the token used for the firing of this transition and add it to the log. Adding the tokens needed for firing a transition shows us in this example which machine was used to do the event.

This Chapter will begin by explaining the design choices made for the resources attributes in Section 5.3. Then it will explain the two modules added in Section 5.3.1. Lastly this chapter will show the result of the implemented models in Section 5.3.2

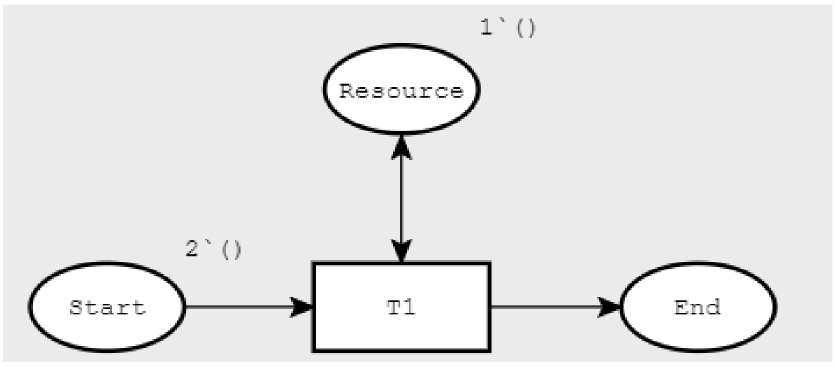


Figure 5.12: A model where a resource is needed to fire a transition

Design decisions

While designing how resources should be portrayed in the log, we should take the following things in mind. The first thing that should be taken into mind is that we want to adhere to the XES standard, the second thing taken in mind is that we want to clearly show which bindings are bound to which variable. The last thing to take in mind that is that we might have multiple tokens of the same color set. We want to differentiate between these colors and show all these colors separately in the log.

The main design for a resource in the log is encoded as such:

```
<string key="COLOR SET:VARIABLE" value="COLOR"/>
```

in which `COLOR SET`, `VARIABLE` and `COLOR` are all variables based on the binding. Sometimes multiple tokens with the same color are needed to fire the transition, for example when a arc has inscription "2‘x". When this is the case a list is created as shown in Listing 5.17.

Listing 5.17: An example of a list generated from an arc inscription

```
<list key="COLOR SET:VARIABLE">
    <string key="r0" value="COLOR"/>
    <string key="r1" value="COLOR"/>
</list>
```

Having this key for the event allows us to know which tokens are used on which arc for a variable.

5.3.1 Modules

To create the row shown in the example above we need three argument. We need the variable and the color of this variable as used in the firing of the transition and we also need the `COLOR SET` of this variable. We already can get the variable and the color of the variable as we are already recording these values as explained in 4.1.2. To be able to match the correct COLOR SET to the variable and to generate the improved log, the variable module was created and the create log module was extended.

Variable module

The variable module is responsible for creating a mapping between variables and color set. Such that each variable maps to a color set. This is done at the beginning of the simulation when loading in the model to the simulator.

Create log module

The create log module was extended in such a way that also resources that are part of the binding are part of the generated log. For each variable in the binding of the transition which is not the caseId variable, a row is generated in the log adhering to the standard shown in 5.3.

5.3.2 Results

The results of Chapter 5.1 already shows that when a variable is part of a binding in any form that the variable is saved. Thus in the result we will assume that the recording module records all the variables that are part of a binding. This section will focus on correctly displaying the resource values in the log. The models used in this section of the thesis are used to show that all the modules discussed in the previous section work as they should. This includes a singular resource added to the log or a list of resources added to the log.

Model with one resource

the model in Figure 5.13 shows a CPN in which one resource is needed to fire a transition. This is a token of color set Machine and has the variable `m`. The log generated when running this example is shown in 5.18

Listing 5.18: Log generated from the model shown in 5.13

```
<log>
    <trace>
        <event>
            <string key="traceId" value="1"/>
            <string key="MACHINE:m" value="m1"/>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
        </event>
    </trace>
</log>
```

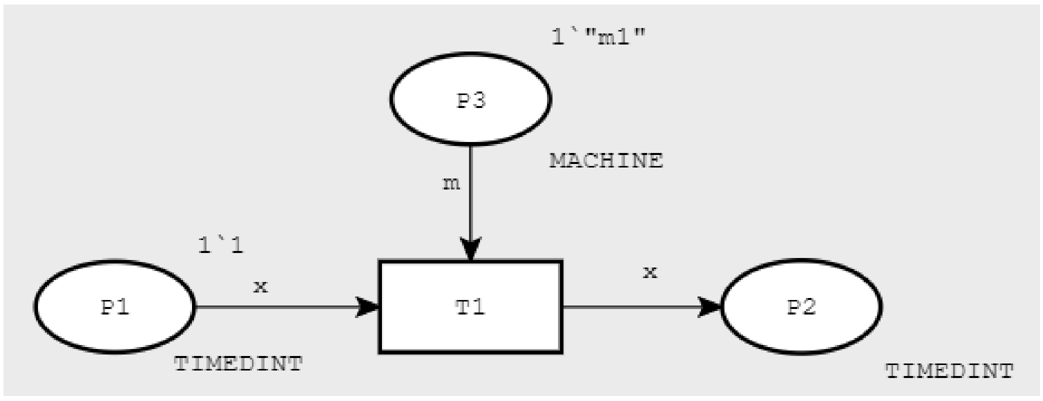


Figure 5.13: Model with one resource

Model with a list of resources

The model in Figure 5.14 shows a CPN in which two tokens of one resource are needed to fire a transition. The log generated when running this example is shown in 5.19

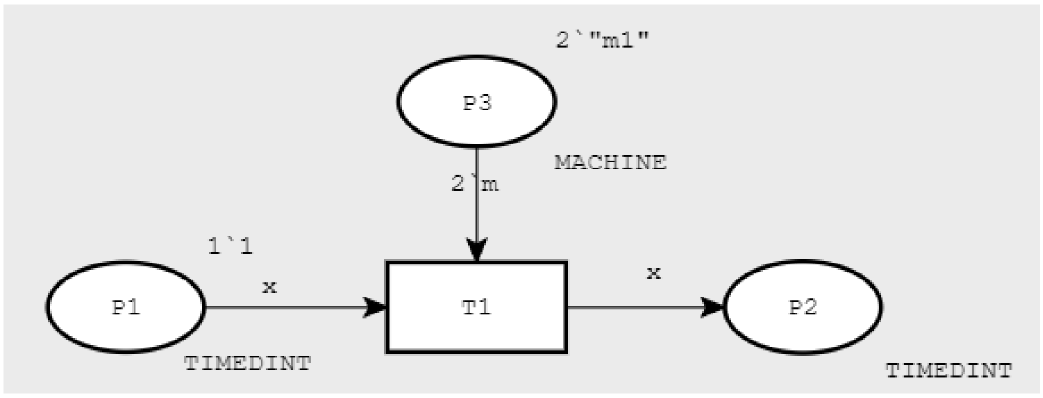


Figure 5.14: Model with a list of resources

Listing 5.19: Log generated from the model shown in 5.13

```
<log>
    <trace>
        <event>
            <string key="traceId" value="1"/>
            <list key="MACHINE:m">
                <string key="r0" value="m1"/>
                <string key="r1" value="m1"/>
            </list>
            <string key="lifecycle:transition" value="complete"/>
            <string key="concept:name" value="T1"/>
        </event>
    </trace>
</log>
```

Model with a resource in an expression

The model in Figure 5.14 shows a CPN in which the resource token is part of an expression, note that this token does not come from a place which has as color set the color set of the resource variable. The log generated from the model is the same as the log generated from model 5.13 and is shown in Listing 5.18

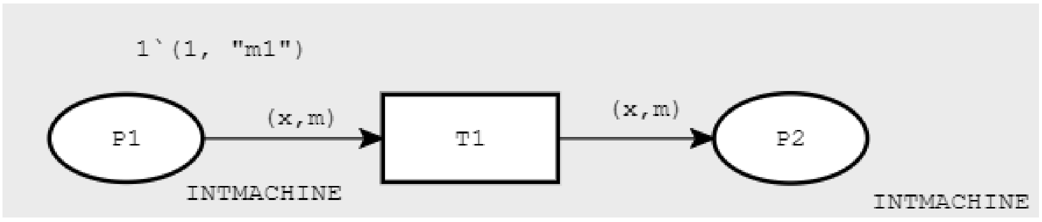


Figure 5.15: Model with a resource in an expression

Chapter 6

Evaluation

6.1 Objective

In this chapter, I want to evaluate the logs generated in this thesis. The logs generated in the chapters should be correct, meaning that you should be able to replay the log on the model and the logs should be adhering to the XES Standard.

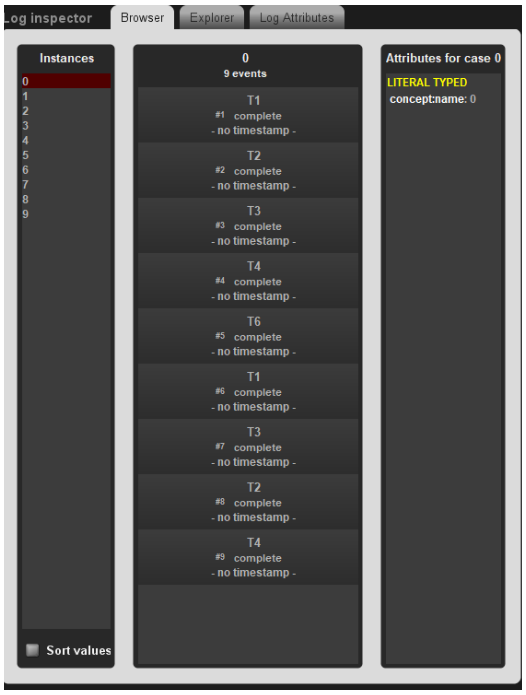
6.2 Execution tools

This section of the thesis shows the way the generated logs will be checked for correctness.

6.2.1 Execution for checking XES Standard and Correctness of Arguments

Since we are using the OpenXES library to create the event log and OpenXES is maintained by the same organisation that defines the XES Standard we will be assuming that this works as expected. However the log will be checked to see if all arguments of the log are linked in the correct way to their attributes. The log will be loaded into the log visualisation function of ProM. This will show for each event the attributes. We expect the values to link to these attributes.

An example of the log visualisation tool (LVT) is shown in Figure 6.1. The LVT has three important windows. The most left window is the log overview window. This window shows all the traces and the names of the traces. The log shown in Figure 6.1 has ten traces. The middle window is the trace overview window. In this window the events in the trace are shown. The trace selected in Figure 6.1 has nine events. For each event a quick summary is given in the trace overview. First the first event the activityName is T1, the lifecycle-transition argument is "complete" and there is no timestamp. The right window shows all the attributes for a selected trace or event. When evaluating the generated logs we will primarily look at the window which is important for the log to be evaluated, this can change for each log.



0
9 events

T1
#1 complete
- no timestamp -

T2
#2 complete
- no timestamp -

T3
#3 complete
- no timestamp -

T4
#4 complete
- no timestamp -

T6
#5 complete
- no timestamp -

T1
#6 complete
- no timestamp -

T3
#7 complete
- no timestamp -

T2
#8 complete
- no timestamp -

T4
#9 complete
- no timestamp -

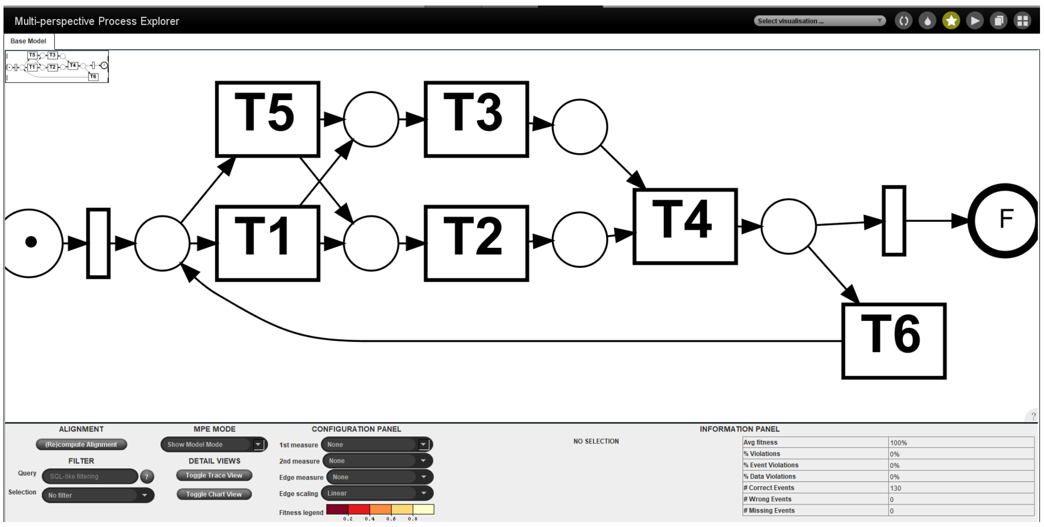
Attributes for case 0
LITERAL TYPED
concept:name: 0

Figure 6.1: An example of the log inspector tool in ProM

6.2.2 Replaying the log

In this section we will be making a distinction between Nets that can be replayed in a replayability tool and nets that cannot be replayed in a replayability tool. We will replay the logs generated from Petri Nets on Petri Nets in the Multi-Perspective Process Explorer (MPPE) of ProM [1]. We will use both the original PN and the generated log as input and try to replay the log on the net. When we have a fitness of 100% and no missed events we know that our log is correctly generated. The logs generated from CPNs, TCPNs and PNs, in which the initial marking has tokens in more than one place, cannot be replayed in a replayability tool, this is because the MPPE does not support Colored Petri Nets. For this an auxiliary PN will be created, this auxiliary PN will have one extra transition which will consume a token in a newly created start place and produce a token in all the places that have a token in the initial marking in the original PN. This way we can still test the PN and the log.

An example of the MPPE is shown in figure 6.2 important aspects are the time each transition takes (shown in the arcs) and the information on the information panel. This information panel shows the fitness as well as number of wrong events and number of correct events.



MPE MODE: Show Model Mode

Query: SQL-like filtering

Selection: No filter

1st measure: None

2nd measure: None

Edge measure: None

Edge scaling: Linear

Fitness legend: 🟥 🟥 🟧 🟨 🟨

Avg fitness	100%
% Violations	0%
% Event Violations	0%
% Data Violations	0%
# Correct Events	130
# Wrong Events	0
# Missing Events	0

Figure 6.2: An example of the MPPE tool in ProM

6.2.3 Checking correctness of lifecycle attributes & time attributes

The correctness of the lifecycle attribute and the time attribute will be checked by hand.

6.3 Setup & Results

This section shows the setup and the result for each of the sub-problems.

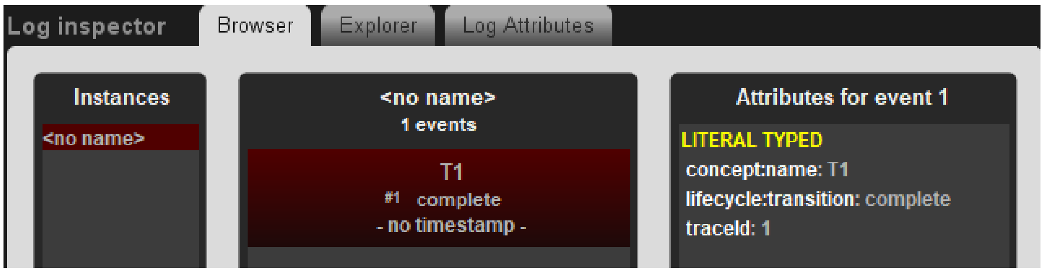
6.3.1 Setup & Result for section 4.1

Setup for Section 4.1

To check whether the program generates a correct log for a CPN I will conduct the following steps. First I will check for each log whether they adhere to the XES Standard. After this I will transform each log in the XES Standard to a log which has a better overview. Then I will try to replay the log on the model to show the correctness of the log. I will do this by hand as there are no auxiliary programs which can help me do this

CPN model from Figure 4.16, 4.17 and 4.18

In this section the log given in Listing 4.2 from model 4.16, 4.17 and 4.18 will be evaluated. First we will check if this log is conforming to the XES Standard. Thus we will upload the log into the log explorer in ProM. This is shown in Figure 6.3. This overview shows that there is one trace and this trace has one event which has activityName T1 and traceId 1.



LITERAL TYPED
concept:name: T1
lifecycle:transition: complete
traceId: 1

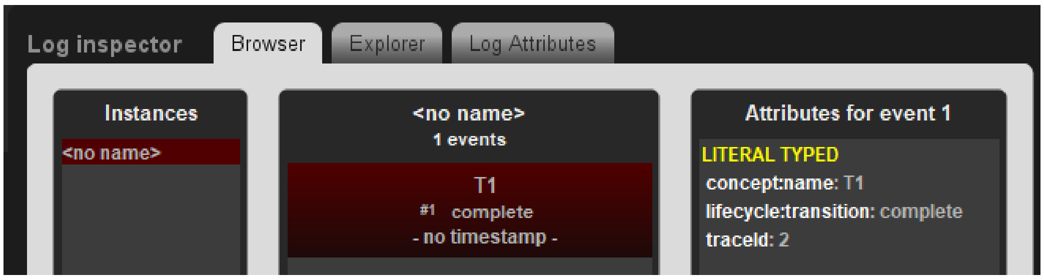
Figure 6.3: Log explorer for the log shown in Listing 4.2

When replaying the log on the model in 4.16, we will first fire transition T1 with the color 1 for the caseId variable which is `x`, which we will be able to do. When replaying the log on the model in 4.17, we will first fire transition T1 with the color 1 for the caseId variable which is `x`, which we will be able to do since we have tokens of color 1 in P1. When replaying the log on the model in 4.18, we will first fire transition T1 with the color 1 for the caseId variable which is `x`, we don't care what the value of `y` is (since we did not record the value) so we are able to do this transition with the token `(1,2)`.

CPN model from Figure 4.19

The fourth CPN model can generate two logs, the first log is the log shown in Listing 4.2 which we have discussed in Section 6.3.1. Also the log shown in Listing 4.3 is able to be generated from model 4.19. This log is uploaded to the log explorer and shown in Figure 6.4. This overview shows that there is one trace and this trace has one event which has activityName T1 and traceId 2.

When replaying the log shown in Listing 4.2 on the model in 4.19, we will first fire transition T1 with the color 1 for the caseId variable which is `x` and



<no name>
1 events

T1
#1 complete
- no timestamp -

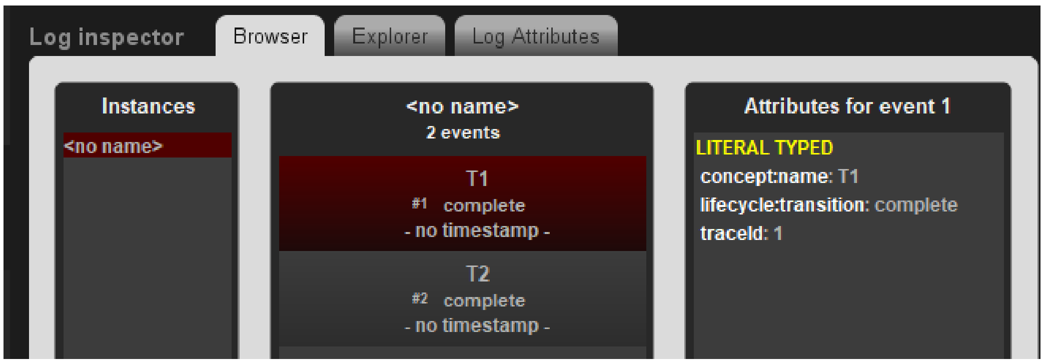
Attributes for event 1
LITERAL TYPED
concept:name: T1
lifecycle:transition: complete
traceId: 2

Figure 6.4: Log explorer for the log shown in Listing 4.3

color 2 with the variable y (which is not shown in the log.), thus replaying the log. When replaying the log shown in Listing 4.2 on the model in 4.19, we will first fire transition T1 with the color 2 for the caseId variable which is x and color 1 with the variable y (which is not shown in the log.), thus replaying the log. Thus we will be able to replay both logs on the model.

CPN model from Figure 4.20

For the fifth model we have generated two logs the log shown in Listing 4.4 was generated when x was the caseId variable, the log shown in Listing 4.5 was generated when y was the caseId variable. Listing 4.4 is uploaded to the log explorer and shown in Figure 6.5, Listing 4.5 is uploaded to the log explorer and shown in Figure 6.6. Figure 6.5 shows one trace with first an event with activityName T1 and then an event with activityName T2. Both have the traceId event attribute 1. Figure 6.6 shows one trace with first an event with activityName T1 and then an event with activityName T3. Both have the traceId event attribute 1.



<no name>
2 events

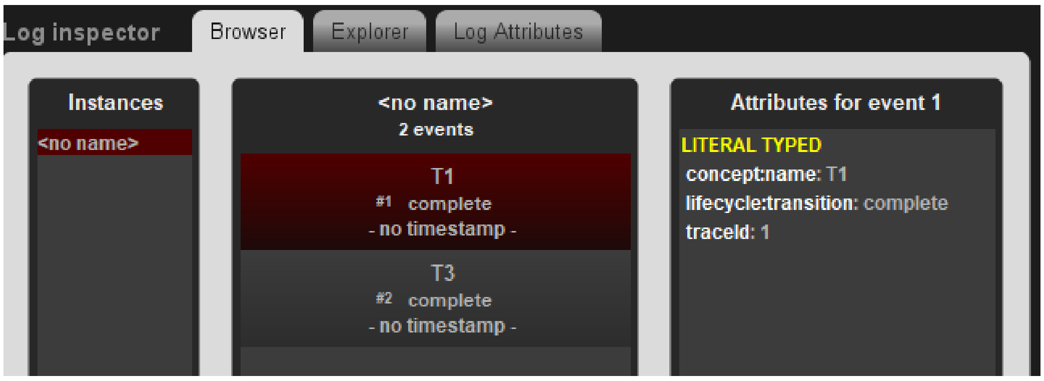
T1
#1 complete
- no timestamp -

T2
#2 complete
- no timestamp -

Attributes for event 1
LITERAL TYPED
concept:name: T1
lifecycle:transition: complete
traceId: 1

Figure 6.5: Log explorer for the log shown in Listing 4.4

When replaying the log shown in Listing 4.4 on the model in 4.20, we will first fire transition T1 with the color 1 from P1 for the caseId variable



LITERAL TYPED
concept:name: T1
lifecycle:transition: complete
traceId: 1

Figure 6.6: Log explorer for the log shown in Listing 4.5

which is x and color 1 from P3 with the variable y (which is not shown in the log.), then we will fire T2 with color 1 again. We will also fire T3, but since T3 does not have the caseId variable in any of its bindings this event does not show up in the log. When replaying the log shown in Listing 4.5 on the model in 4.20, we will first fire transition T1 with the color 1 from P3 for the caseId variable which is y and color 1 from P1 with the variable y (which is not shown in the log.), then we will fire T3 with color 1 again. We will also fire T2, but since T2 does not have the caseId variable in any of its bindings this event does not show up in the log.

CPN model from Figure 4.21

For the sixth model shown in Figure 4.21 the listing 4.5 was generated. We have discussed the correctness of this log in 6.3.1. When replaying the log on the model shown in Figure 4.21. We will first fire transition T1, with color 1 from P1 for the caseId variable. Then the log shows a T3 transition which we are not yet able to do in the model. Thus we will first fire the silent transition in the model, producing a token with color 1 in P3. Now we are able to fire transition T3 with color 1 for the caseId variable.

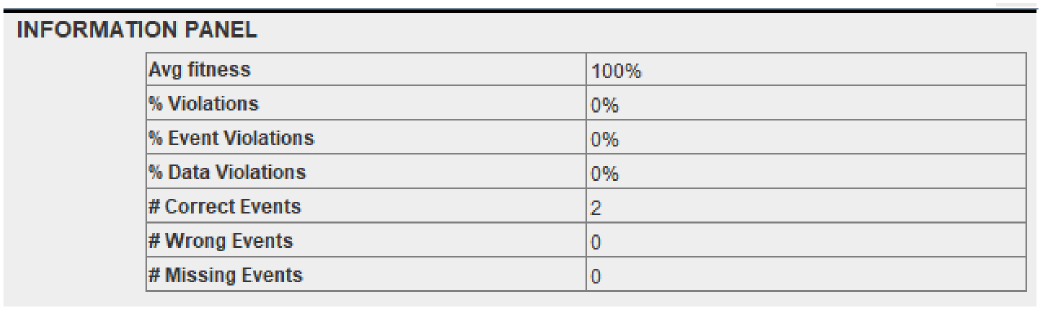
6.3.2 Setup & Result for section 4.2

Setup for Section 4.2

To check whether the program generates a correct log for a PN, I will assume that a correct log can be generated for a CPN. Thus checking whether the log adheres to the XES Standard is not necessary. Secondly I will check whether the function I built are transforming the log as expected. Lastly I will try to replay the log generated from the transformed model using a log replayability tool.

Result from PN shown in model 4.32

For the first PN model shown in Figure 4.32 the log from listing 4.8 was generated. This log was then replayed over the model shown in Figure 4.32 MPPE. The information panel from the MPPE is shown in Figure 6.7. This figure shows that there was a fitness of 100% and there were two events that were correct and no events that were wrong.



INFORMATION PANEL

Avg fitness	100%
% Violations	0%
% Event Violations	0%
% Data Violations	0%
# Correct Events	2
# Wrong Events	0
# Missing Events	0

Figure 6.7: Information panel of the replay of the log from Listing 4.8 on model 4.32

Result from PN shown in model 4.30

For the second PN model shown in Figure 4.30 the log from listing 4.7 was generated. This log was then replayed over the model shown in Figure 6.8 using the MPPE. The model for the generation was slightly modifies because th MPPE was not able to replay on the original model as the original model had two places that contained tokens in the initial state and this was not allowed by the MPPE. The information panel from the MPPE is shown in Figure 6.9. This figure shows that there was a fitness of 71.5%, there was one event that was correct and one event that was missing, the missing event is T2 as this was added to replay the log correctly.

6.3.3 Setup & Result for Section 5.1

Setup for Section 5.1

To check whether the program generates a correct log which has lifecycle-transitions. We will first check whether the logs adhere to the XES Standard.

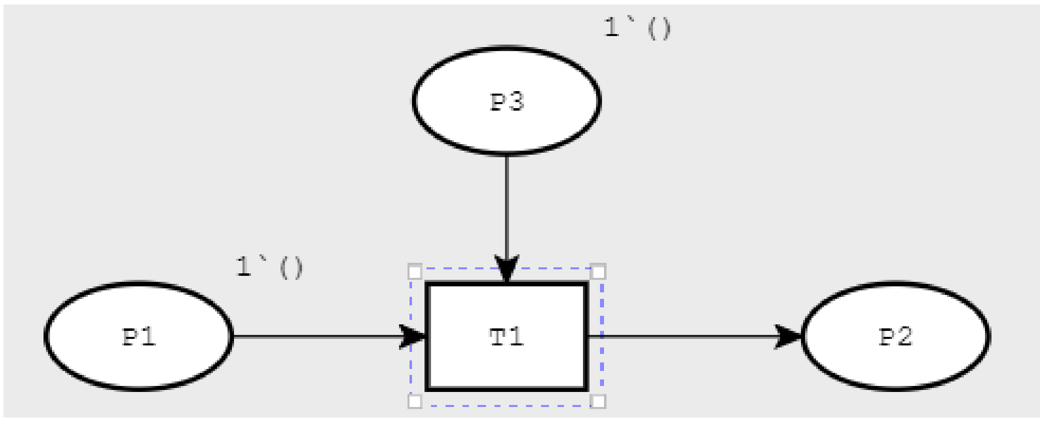
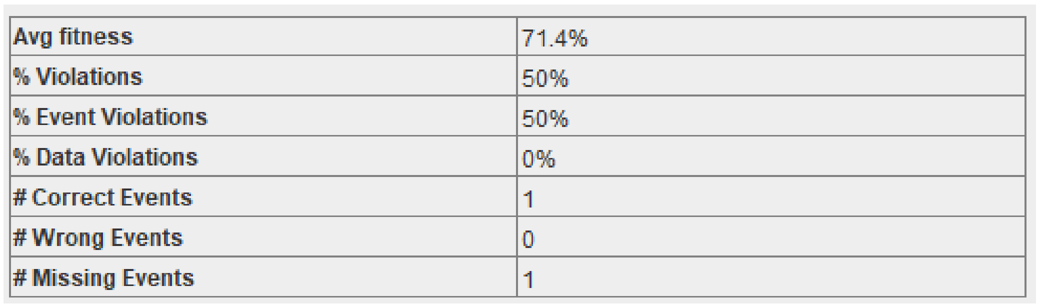


Figure 6.8: The model used to replay the log on



Avg fitness	71.4%
% Violations	50%
% Event Violations	50%
% Data Violations	0%
# Correct Events	1
# Wrong Events	0
# Missing Events	1

Figure 6.9: Information panel of the replay of the log from Listing 4.7 on model 6.8

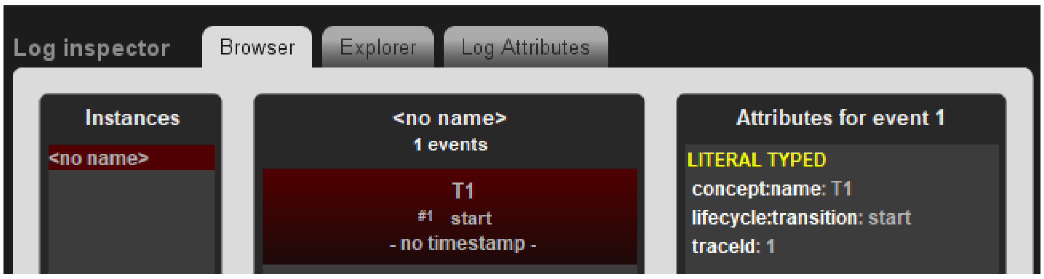
Then we will check whether the lifecycle-transition attribute has the correct value for the different options for lifecycle-transition attribute checking all the possible options for lifecycle-transition attribute.

Result from CPN shown in model 5.4

For the model shown in Figure 5.4, the log from Listing 5.3, Listing 5.4 and Listing 5.5 were created with `complete`, `start` and `start & complete` as LTA respectively.

When viewing the the log from Listing 5.4 in the log explorer tool as shown in Figure 6.10, we can see that the log has one trace, this trace has one event which has activityName T1 and LTA value `start`, this is as expected as we choose the `start` value as LTA. We can easily see that this log can be replayed on the model.

When viewing the the log from Listing 5.3 in the log explorer tool as shown in Figure 6.11, we can see that the log has one trace, this trace has

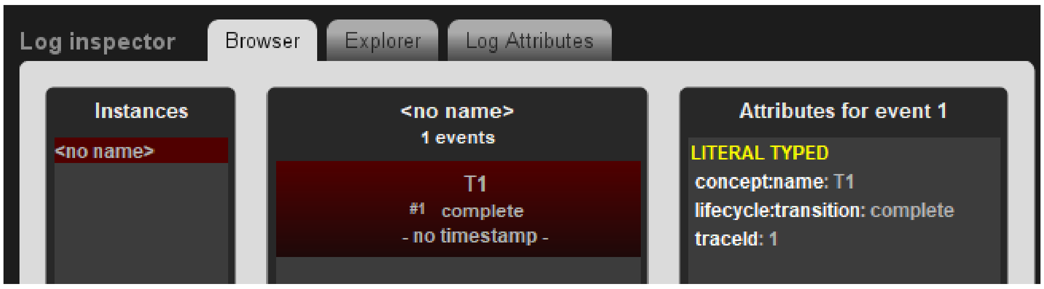


T1
#1 start
- no timestamp -

Attributes for event 1
LITERAL TYPED
concept:name: T1
lifecycle:transition: start
traceId: 1

Figure 6.10: Log explorer for the log shown in Listing 5.4

one event which has activityName T1 and LTA value `complete`, this is as expected as we choose the `complete` value as LTA. We can easily see that this log can be replayed on the model.



T1
#1 complete
- no timestamp -

Attributes for event 1
LITERAL TYPED
concept:name: T1
lifecycle:transition: complete
traceId: 1

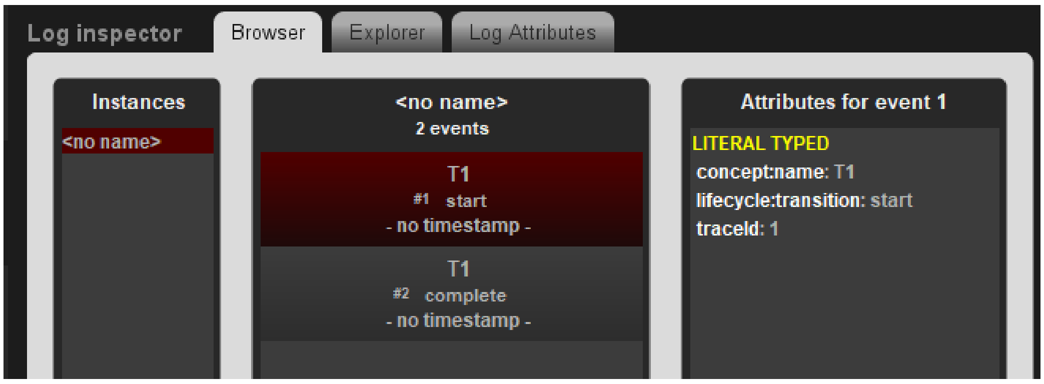
Figure 6.11: Log explorer for the log shown in Listing 5.3

When viewing the the log from Listing 5.5 in the log explorer tool as shown in Figure 6.12, we can see that the log has one trace, this trace has two event both have activityName T1, one has the LTA value `start` and the second one has the LTA value `complete`. This is as expected. To see if the log is replayable on the model we have to take in mind that each transition now has two events associated to it. A start event and an end event. When we take this in mind we can see that we can replay the log on the T1 transition thus we are able to replay the log on the model.

Result from CPN shown in model 5.3

For the model shown in Figure 5.3, the log from Listing 5.1 and Listing 5.2 were created with `complete` and `In transition name` as LTA respectively.

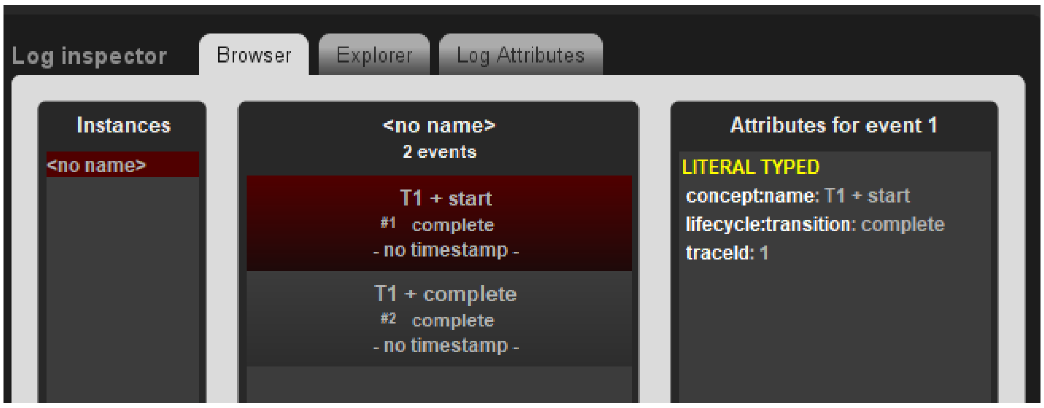
When viewing the log from Listing 5.1 in the log explorer tool as shown in Figure 6.13, we can see that the log has one trace, this trace has two events the first event has activityName `T1+start` and LTA `complete` and the second event has activityName `T1+complete` and LTA `complete`. This



concept:name: T1
lifecycle:transition: start
traceId: 1

Figure 6.12: Log explorer for the log shown in Listing 5.5

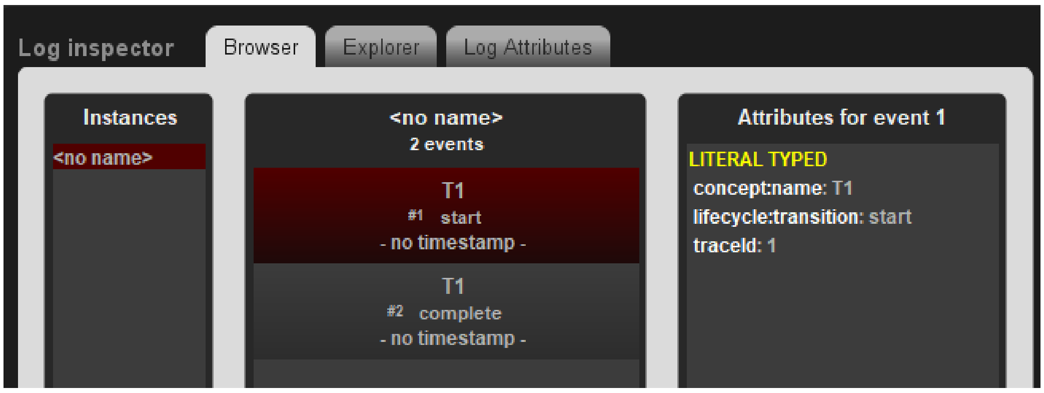
is as expected, since we have automatically given all the transition the LTA complete, the entire transition name will stay the same. It is trivial to see that the log can be replayed on the model.



concept:name: T1 + start
lifecycle:transition: complete
traceId: 1

Figure 6.13: Log explorer for the log shown in Listing 5.1

When viewing the log from Listing 5.2 in the log explorer tool as shown in Figure 6.14, we can see that the log has one trace, this trace has two events both events have the activityName `T1`, while the first event has LTA `start`, the second event has LTA `complete`. This is as expected, since the user has manually given each activityName a LTA, the LTA are extracted from the transition name and the transition name is correctly split between the transition name and the LTA. When replaying the log we first fire transition `T1+start` (we can do this as we are ignoring the `+ start` at the end of the first transition. Then we will fire transition `T1+complete`. Also ignoring the `+complete` part. Thus replaying the log.



T1
#1 start
- no timestamp -

T1
#2 complete
- no timestamp -

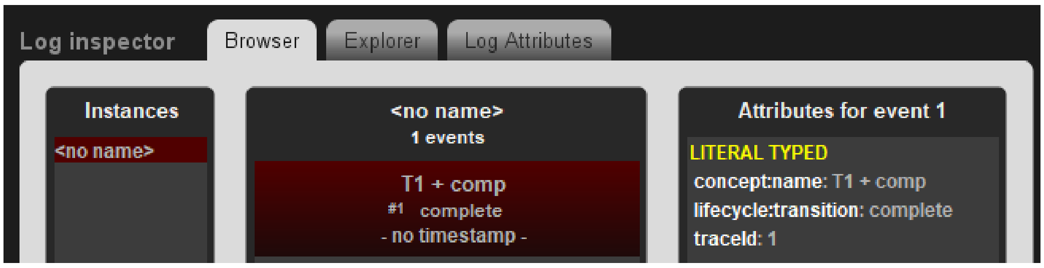
Attributes for event 1
LITERAL TYPED
concept:name: T1
lifecycle:transition: start
traceId: 1

Figure 6.14: Log explorer for the log shown in Listing 5.2

Result from CPN shown in model 5.5

For the model shown in Figure 5.5, the log from Listing 5.6 is generated created with `In transition name` as LTA.

When viewing the log from Listing 5.6 in th log explorer tool as shown in Figure 6.15, we can see that the log has one trace, this trace has one events both with activityName `T1+comp`, the event also does not have the LTA `comp` but the standard value for LTA which is `complete`. This is as expected, since the user has selected to manually given each activityName a LTA and has failed to give a valid LTA as discussed in Section 2.5. The replay of the log is trivial.



T1 + comp
#1 complete
- no timestamp -

Attributes for event 1
LITERAL TYPED
concept:name: T1 + comp
lifecycle:transition: complete
traceId: 1

Figure 6.15: Log explorer for the log shown in Listing 5.6

Since the examples provided in this example cover all the possible options the LTA can be used in generating the log we can confidently say that we can correctly add LTAs to the lag.

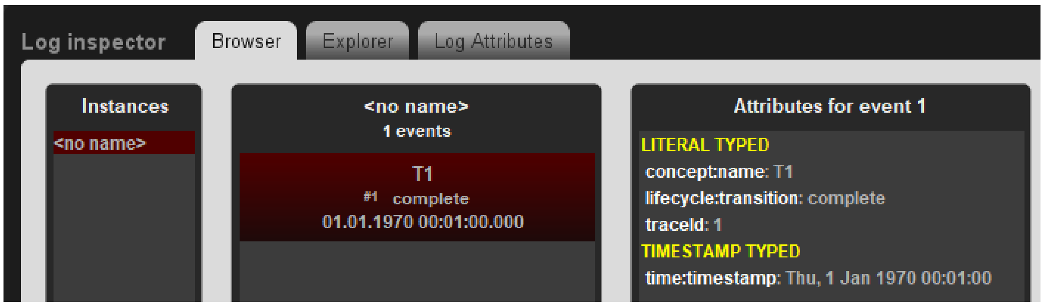
6.3.4 Setup & Result for section 5.2

Setup for Section 5.2

To check whether the program generates a correct log which has a time attribute, we will first check whether the log adheres to the XES Standard, more specifically whether the Time attribute is added to the log conforming to the XES Standard. Then we will check whether the end time of a transition is correctly calculated. Lastly we will check if the simulator time is correctly transformed into a real time.

Result from TCPN shown in model 5.8

From the model shown in 5.8 we generated the log shown in Listing 5.7 we also generated the logs which are partly shown Listing 5.8, 5.9, 5.10, 5.11 and 5.12. When viewing the log from listing 5.9 in the log explorer tool from ProM we can see the screen as in Figure 6.16. This log shows that there is one trace with one element in the trace. We can also see that this event has a time attribute, which is what we want. The timestamp of the event is 1970-01-01 00:01:000+1:00, which is one minute after the StartingMoment which is exactly what we expected.



Instances
<no name>

<no name>
1 events
T1
#1 complete
01.01.1970 00:01:00.000

Attributes for event 1
LITERAL TYPED
concept:name: T1
lifecycle:transition: complete
traceId: 1
TIMESTAMP TYPED
time:timestamp: Thu, 1 Jan 1970 00:01:00

Figure 6.16: Log explorer for the log shown in Listing 5.7

We also check the logs shown in Listing 5.8, 5.9, 5.10, 5.11 and 5.12. These logs are all expected to make one time step in their respective TimeUnit, we can see that this is the case.

Next we check the time generated in the log shown in Listing 5.13 we expect this time to be the time filled in in the StartingMoment field which is 1970-01-01 00:00. When checking the log in Listing 5.13 we can see that this is indeed the time of the first event (this event has LTA start).

Lastly for the model shown in 5.8 we generated the log in Listing 5.14. For this log we expect the time of the first event to be 2002-02-02 02:02, as that is the time we filled in as StartingMoment and we only did not add time to the simulation before the first event. We can clearly see that this is the case.

The above models show that the TimeLog and TokenController module work for transitions that take deterministic time and LTA that are `start`, `complete` or `start+complete`.

Result from TCPN shown in model 5.9

From the model shown in 5.9 we generated the log shown in Listing 5.15. When looking at the time of the events in the log we can see that the first event has timestamp 1970-01-01 00:00, which is as expected as this event happened at time 0. We can also see that the second event has timestamp 1970-01-01 00:01, this is correct as the TimeUnit was set to minutes and the second event was fired at simulator time 1. This proves that when the LTA is in the transition name the time of that event will be the time the transition fires.

Result from TCPN shown in model 5.10

From the model shown in 5.10 we generated the log shown in Listing 5.16. When looking at the time of the event in the log we can see that the first event has timestamp 1970-01-01 00:00, which is as expected as this event happened at time 0. We can also see that the second event has timestamp 1970-01-01 00:00:23.489, which is as we expected as the token with time 0.391483647815 is the token produced with the lowest time and 0.391483647815 and we have selected `minutes` as TimeUnit which means that 0.391483647815 in the simulator is $0.391483647815 * 60 = 23.48901$ seconds in the log. We also see that the third event has timestamp 1970-01-01 00:01:23:489, which is as expected as the token with the highest time consumed is the token with time 1.391483647815 and we have selected the `minutes` TimeUnit which means that 1.391483647815 in the simulator is $1.391483647815 * 60 = 1$ minute and 23.48901 seconds in the log.

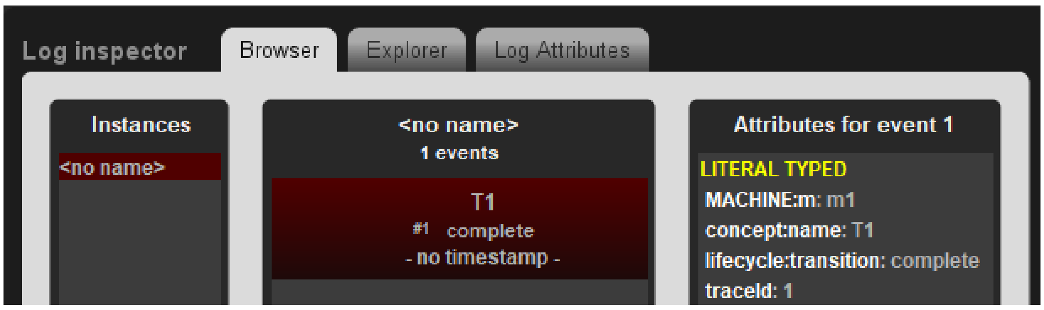
6.3.5 Setup & Result for Section 5.3

Setup for Section 5.3

To check whether the program generates a correct log which has correct resource attribute, we will first check whether the log adheres to the XES Standard, more specifically whether the newly added resource attribute are added to the log conforming to the XES Standard. Then we will check whether the resources are correctly added to the log.

Result from CPN shown in model 5.13

For the model shown in Figure 5.13, the log from Listing 5.18 was generated. When viewing the log from Listing 5.18 in the log explorer tool as shown in Figure 6.17, we can see that the log has one trace, this trace has one event which has activityName T1, lifecycle:transition attribute `complete` and a `MACHINE:M` attribute with value m1. This is exactly what we want. We can easily see that the log can be replayed on the model if we bind m1 to m and the caseId variable to x.



MACHINE:m: m1
concept:name: T1
lifecycle:transition: complete
traceId: 1

Figure 6.17: Log explorer for the log shown in Listing 5.18

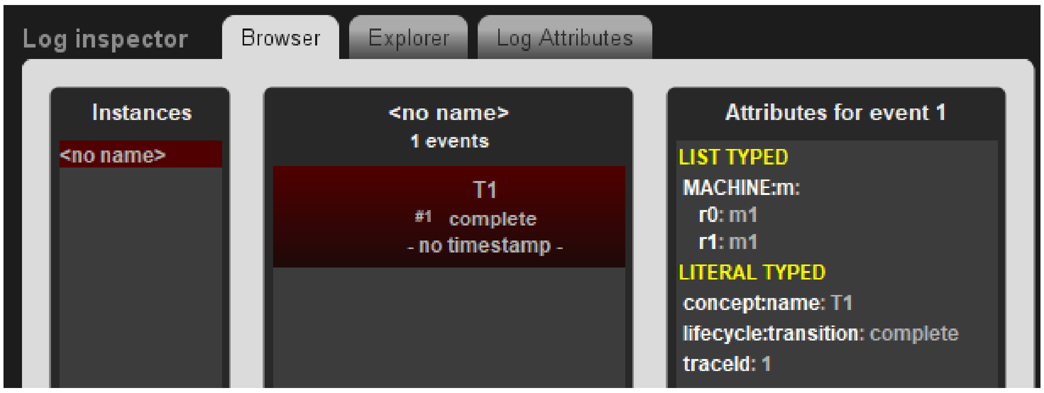
Result from CPN shown in model 5.14

For the model shown in Figure 5.14, the log from Listing 5.19 was generated. When viewing the log from Listing 5.19 in the log explorer tool as shown in Figure 6.18, we can see that the log has one trace his trace has one event which has activityName T1, lifecycle:transition attribute `complete`. We can also see that this event has a list attribute, the value of this list is `MACHINE:M`, and this list contains two value, r0 with value m0 and r1 with value m0. This is exactly what we want. Replaying the log on the model is trivial.

Result from CPN shown in model 5.15

For the model shown in Figure 5.15, the log from Listing 5.19 was generated, we already checked this log for correctness and also in this setting this log is exactly what we wanted. Replaying the log on the model is also trivial.

As we are able to correctly display a resource in a log when it is a singular resource and when it is a list of resources we are able to successfully use resources in the log.



<no name>

<no name>
1 events

T1
#1 complete
- no timestamp -

Attributes for event 1

LIST TYPED
MACHINE:m:
r0: m1
r1: m1
LITERAL TYPED
concept:name: T1
lifecycle:transition: complete
traceId: 1

Figure 6.18: Log explorer for the log shown in Listing 5.19

6.4 Discussion

Section 6.3.1 shows that the generated logs are conforming to the XES Standard and that the generated logs can be replayed on the initial CPN. We conclude from this that the logs are correctly generated from a CPN. Section 6.3.2 shows that when the requirements shown in the section hold, we can transform a PN into a CPN with the use of the `Petri Net Transformation` module and the `Token Generator` module while the flow of the model remains the same. A correct log can be generated from these CPNs as shown in Section 6.3.1. Combining the findings in Section 6.3.1 and Section 6.3.2 we conclude that we can correctly generate a log from a CPN and a PN (If the PN has certain requirements).

Section 6.3.3 shows that we can correctly add the lifecycle-transition to all events in the log. As we know that the log before adding lifecycle-transitions is conforming to the XES Standard, and we know that the added attribute is added conforming to the XES Standard, we can conclude that we can generate logs which have a lifecycle-transition attribute conforming to the XES Standard. Section 6.3.3 also shows that the log produced is still able to be replayed on the CPN thus we are able to correctly generate a log containing lifecycle-transition-attributes from a CPN.

Section 6.3.4 shows that we can correctly determine the real time of a time in the log. The section also shows that we can correctly determine the time of an event. Lastly the section shows that time is added to the log conforming to the XES Standard. As we know that the log before adding the time to the log is conforming to the XES Standard and we have only added time to the log we conclude that the newly generated log is also conforming to the XES Standard

Section 6.3.5 shows that we can correctly add resources to the log, both a single resource and a list of resources. As we know that the log before adding the resources to the log is conforming to the XES Standard and we only added resources to the log we can conclude that the newly generated log is also conforming to the XES Standard.

From the statements above we can conclude that we are able to generate a log that is conforming to the XES Standard and is replayable on the initial TCPN. This means that a log can be created that can includes a lifecycle-transition attribute for each event, a timestamp for each event and the resources used for each event.

Chapter 7

Conclusion

As discussed in Section 6.4 the application is able to generate a log conforming to the XES Standard from the simulation of a CPN in which the log can contain a lifecycle-transition attribute, a timestamp attribute, and resource attributes.

As discussed in Chapter 2.6 there does not yet exist a program that is able to generate logs from the simulation of a CPN. By using this program future researchers will be able to create more specific logs, which can be used for example for creating new discovery techniques or testing current discovery techniques. In theory researchers should be able to create most logs with this application.

However we are not yet able to construct all possible logs with the current implementation in CPN-IDE. The attributes missing in the current implementation can in my opinion be divided into two parts.

The first part is the log information. When generating logs it is normal to have some sort of log attributes, which tells something about the log, for example the distribution of activities, the timespan of the log or the default event-names. The functionality to create these properties is completely missing in this program and should be interesting to explore. Next to this trace attributes could also be added in more detail.

The second component that could be added is noise. Currently the log does not have any noise as it records the activities and transforms them into events. As we are generating a log based on a real simulation we can use this simulation to create more sensible noise, this is noise that would be more realistic to happen in the real world, for example instead of picking a random transition name to be the noise, we can pick a transition name that was enabled at the point the transition fired. There are tools that can add noise to existing logs, including the logs we created. These log generators however are often not user friendly, they cannot add sensible noise and they are not already in CPN-IDE. Thus the log first needs to be created and then exported to an external tool. An extension which would allow the user to add noise during the recording would be beneficial.

Glossary

process	A series of steps and decisions made in the way something is completed
Petri Net (PN)	A modelling language for process systems as explained in 2.1
Colored PN (CPN)	An extension on a PN that contains colors as explained in 2.2
Timed CPN (TCPN)	An extension of a CPN which contains time as explained in 2.3
flow	Every possible sequence of fired transitions from a certain marking of a model.
event	A specific activity happening on a specific time
trace	A sequence of events
case	an identifier a event belongs to
event log	a recording of events usually ordered by trace
transition name	the name of a transition in a PN
simple event log	An event log in which the events only contain the transition name
color	the value of a token as explained in 2.2
color set	the datatype of a token as explained in 2.2
real time	the time in which the event took place in the real life.

Bibliography

[1] Prom process mining software. `https://www.promtools.org/doku.php`. version 6.11.

[2] the bpic logs. `https://www.tf-pm.org/resources/logs`. latest page update 19-11-2021.

[3] the cpn-ide application. `http://cpntools.org/cpn-ide/`. Accessed before writing this thesis.

[4] the xes-standard. `https://www.xes-standard.org/_media/xes/xesstandarddefinition-2.0.pdf`. version 2.0.

[5] Timed colored petri nets. `http://cpntools.org/2018/01/16/timed-nets/`. Posted January 16, 2018.

[6] Andrea Burattin and Alessandro Sperduti. Plg: A framework for the generation of business process models and their execution logs. In Michael zur Muehlen and Jianwen Su, editors, *Business Process Management Workshops*, pages 214–219, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[7] Claudio Di Ciccio, Mario Luca Bernardi, Marta Cimitile, and Fabrizio Maria Maggi. Generating event logs through the simulation of declare models. In Joseph Barjis, Robert Pergl, and Eduard Babkin, editors, *Enterprise and Organizational Modeling and Simulation*, pages 20–36, Cham, 2015. Springer International Publishing.

[8] Jonghyeon Ko, Jongyup Lee, and Marco Comuzzi. Air-bagel: An interactive root cause-based anomaly generator for event logs. In *ICPM Doctoral Consortium/Tools*, pages 35–38, 2020.

[9] Luis Leiva, Jorge Munoz-Gama, Juan Salas-Morales, Victor Galvez, Wai Lam Jonathan Lee, Rene de la Fuente, Ricardo Fuentes, and Marcos Sepúlveda. Pomelog: Generating event logs from unplugged processes. *BPM (PhD/Demos)*, 2420:189–193, 2019.

[10] Alexey A. Mitsyuk, Ivan S. Shugurov, Anna A. Kalenkova, and Wil M.P. van der Aalst. Generating event logs for high-level process models. *Simulation Modelling Practice and Theory*, 74:1–16, 2017.

[11] James L Peterson et al. A note on colored petri nets. *Inf. Process. Lett.*, 11(1):40–43, 1980.

[12] Ivan Shugurov and Alexey Mitsyuk. Generation of a set of event logs with noise. 01 2014.

[13] Vasyl Skydanienko, Chiara Di Francescomarino, Chiara Ghidini, and Fabrizio Maria Maggi. A tool for generating event logs from multi-perspective declare models. *Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018*, 2196:111–115, 2018.

List of Figures

List of Tables