Comparing stories with the use of Petri Nets

Loïs Mooiman
10318364

Bachelor thesis
Credits: 18 EC

Bachelor Opleiding Kunstmatige Intelligentie

University of Amsterdam
Faculty of Science
Science Park 904
1098 XH Amsterdam

Supervisors
G. Sileno & A. Boer

Leibniz Center for Law
Faculty of Law
University of Amsterdam
Vendelstraat 8
1012 XX Amsterdam

June 26th, 2015

Abstract

In this thesis twenty basic legal stories are modelled into Petri nets to compare them in terms of different similarities. Petri nets are state-transition systems which are used to represent the behaviour of systems with the use of transitions, places, arcs and tokens. They can depict the storyline well and causation is based on the proceedings of the story. Therefore Petri nets are useful for representing legal stories however there is not yet a comparison measure to compare these stories. A consistent model is created in this paper to represent the stories in Petri nets. After which three similarity measures are considered: label, structural and execution similarity. Label similarity compares the labels of the nodes and thus compares the Petri nets on their subject and events. Structural similarity computes the difference between two stories by inserting and deleting nodes and compares them therefore in complexity. The difference in execution uses the flow of the tokens to measure them and this is partically useful to represent the storyline. Label and structural similarity are implemented and there results are ranked in matrices. Beforehand the stories are ranked by a person and the results of the computational measures and are correlated with the use of Spearman's rank correlation algorithm. Label similarity has the highest correlation because subject and events are important for human comparison instead of the complexity of the story. However label comparison is still a mechanical way to compare stories and the execution comparison measure would probably be higher correlated because of a better representation of the story flow. This thesis concludes that label similarity is a better way to compare stories then structural similarity.

Contents

1 Introduction

For the past few years programs and algorithms have been build and taught of to compare stories such as movieplots, literature and business cases, because categorization of those stories would be easier if a program could categorize them based on different similarities. However each different kind of story has different important aspects. For example, the main characters and events are important in literature, whereas the execution of steps and subject of the events are important in business cases. In the legal domain discourse and causation are important for categorising legal cases (Lehmann et al., 2005). Thus discourse and causation need to be represented well to compare legal cases. Judges could use a comparison measure to find similar cases and their verdict to base the new verdict on the old one.

Petri nets are a way to represent the flow of a system, which is important to represent the storyline and thus causation (Lehmann et al., 2005). That is why Petri nets could be used to represent legal cases (Sileno and Boer, 2014). However, comparing stories with a program is mechanical and thus the flow of these stories cannot be compared in a humanlike way (Fisseni and Löwe, 2012). An appropriate comparison measure is needed to compare stories without loss of the storyline which could be used for comparing legal cases.

In this thesis a comparison measure is introduced to find similarities between legal stories represented in Petri nets. The data set consists of simple legal cases, the stories, which need to be represented in Petri nets and the basics for each representation needs to be universal. An similarity matrix is made beforehand to compare the legal stories and three comparison measures, label similarity, structural similarity and behavioural similarity (Dijkman et al., 2011), are researched to apply to this problem. Based on these types of similarity and the assumption that the story flow needs to be represented a comparison measure is created to achieve appropriate results.

This thesis focusses on finding one comparison measure that will satisfy the specified evaluation. The Petri nets will be evaluated on multiple similarities beforehand to ensure a comparison measure with appropriate results. First they are ranked to find out which elements are important for human comparison and secondly the correlation matrix is computed. This evaluation is used to create a comparison measure which will compare the stories the same.

The remainder of this thesis is structured as follows. First, relevant literature is discussed and linked to this thesis in section 2. Secondly, the procedure of creating the story base is explained, the comparison measures and Petri nets are explained in section 3. After which the implementation of the program is explained in section 4 and the results and evalution is discussed in the next section. The section 6 discusses the whole thesis and presents conclusions and the last section states future research to be done.

2 Literature Review

In literature multiple models are proposed to represent stories. The studies focussing on this topic are constructing a structural model to represent the narrative in stories. Schank and Abelson (1979) is an example of such a study. Schank and Abelson (1979) introduce scripts as a structural model. Scripts are linear models and are about creating models which represent multiple events. However Sileno and Boer (2014) discusses a systematic representational model. Sileno and Boer (2014) want to describe stories in the legal domain accurately and therefore it needs to represent discourse and causation. Lehmann et al. (2005) explains why causation is essential to attribute legal responsibility. Thus causation needs to be represented to assign value to representations of legal cases.

Sileno and Boer (2014) propose a new method to represent these aspects. They use Petri nets to represent the story flow of stories from the legal domain. A Petri net is a state-transition system which uses places, transitions and tokens. Murata (1989) writes about them: "Petri nets are a promising tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic." Behrens and Dix (2007) eloborate on Petri nets and propose Simple Logic Petri nets which are Petri nets based on logical expressions and an extension to the original ones. Petri nets have not been used before in the legal domain to represent stories. Sileno and Boer (2014) show the newest en first research on this subject.

Sileno and Boer (2014) discuss the representation of legal stories and they mention the topic of comparing stories but do not eloborate on this subject. Psychological studies find that people compare stories in several different ways. Fisseni and Löwe (2012) suggest that human judgement for comparing objects or events is not based on structural dimensions but on event structure and causal links. Therefore structural comparison can be used to compare stories. However to depict the story flow for humans event structure and causal links need to be considered as well.

A structural comparison model used in computation is structural similarity. Structural similarity is determined on the number of steps used to get from one graph to another. Löwe (2011) discusses this kind of similarity and states the advantages and disadvantages. High precision and accuracy is an advantage but relations or subjects of nodes are discarded. Dijkman et al. (2011) wants to find a comparison method which can compare business models displayed in graphs. One of these methods is structural similarity. Dijkman et al. (2011) compare the results of structural similarity to two other methods: node and behavioural similarity. Node similarity calculates the word similarity between the labels of the nodes by matching words or short sentences. Behavioural similarity calculates the similarity of relations between events which could simulate an human-like comparison of stories if the theories of Fisseni and Löwe (2012) about human judgement are correct. The results of Dijkman et al. (2011) conclude that structural similarity has the average highest precision over the trials. However behaviour similarity is better to represent the similarity between relations and sequences of events.

Behavioural similarity, introduced by Dijkman et al. (2011), could be used to compare stories from the legal domain represented in Petri nets, like Sileno and Boer (2014) state, because it stresses the sequences and relations of events and therefore contributes to represent discourse and causation. This subject will be eloborated upon in the upcoming paragraphs.

3 Approach

First a database of Petri nets based on legal stories is created to use as a data set for the comparison measure and second different options are explored to find an appropriate comparison measure. This process is described in the following paragraphs.

3.1 Stories

The twenty stories included in A are all legal cases and consist of short sentences which display the occuring events. The subjects of the stories are sale, robbery, theft, etc and can be malfunctioning of social instution. For example the Failed Sale 2 story listed below displays a sale where the good does not arrive and thus one of the agents, in this case Y, is not pleased and this is for him considered as a failure.

Failed Sale 2 Story:

X makes an offer.
The offer concerns a good exchanged for a certain amount of money.
Y accepts the offer.
Y pays the money.
X does not deliver.

Similar events over multiple stories are phrased the same and if needed the intent is described as well, for instance in the Theft False story it says explicitly that the agent did steal for an emergency which is deemed as a good intent. The stories do not state what the exact ending is for the agents. This is induced from them when they need to be modelled into Petri nets.

3.2 Petri nets

Petri nets are state-transition systems which are used to represent the behaviour of systems with the use of transitions, places, arcs and tokens. Transitions can only be connected to places and vice versa because the transitions fire the tokens from one place to the other along the lines of the arcs. An emitter starts firing the tokens to the first places and each connected place gets a token and once fired it cannot be fired again. The places hold the tokens until the connected transition starts firing and the transition can only start firing when each of the input places has a token. Therefore it needs to wait when not all his inputs has token until other transitions are executed. Places can hold multiple tokens, however it will only pass one for each time the transition is fired. When the tokens activate the collector and the collector collects the tokens when executed the Petri net is finished. Arcs regulate this whole flow of the tokens and point from place to a transition and vice versa. They can also be a biflow which means that the tokens go both ways therefore when the transition is fired a token will flow back into the place where the token also came from. (Murata, 1989)
The software used for this thesis to model and create the Petri nets is Yasper (ASPT, 2005) and these transitions are displayed as green blocks, the places as yellow dots and the arcs as arrows. The emittor and collector are yellow blocks with respectively the letter E or C. The flow of the tokens is the same as described above.

3.2.1 Modelling the stories

Each Petri net has one emitter and one collector. Sileno and Boer (2014) describe the different layers of a story which need to be represented in Petri nets: the message layer and the agent layers. The message layer represents the basic story, for example: *X makes offer* is an event and is represented as the transition *made offer* in the message layer. However X is involved in this event thus in the agent layer of X transitions and places are added to create the event of *making the offer* as seen in figure 1. The message layer is always in the middle line of the Petri net and is flanked by the agent layers.
In the Failed Sale 2 story there is a failure for one of the agents because the package does not arrive. This is displayed in figure 1 as a transition labelled with *Failure*. Negation is represented in labeling as a *no* and does not have a representation in places or transitions. Thus negation is not considered when modelling the stories and is not handled.
Biflow arcs are used to represent an ongoing relation between the agent layer and the message layer. A loop is used to keep the tokens in place as seen in figure 1 when the agent is *making acceptance* while in the message layer *acceptance made* is active.

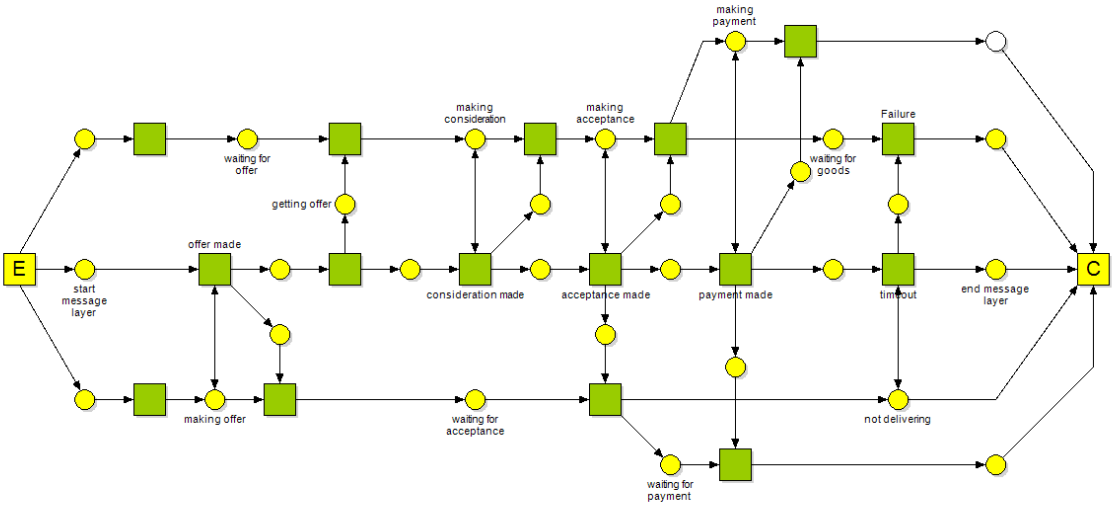

Figure 1: FailedSale2Story - Petri net representation

In some of the stories an extra layer is necessary to display the intent of the agents, for instance an intent to steal a car or use someone elses car for an emergency. Stealing a car is considered a crime whereas stealing a car for an emergency is not, therefore an intentional layer incoorperated into the agent layer displays the difference between the stories and Petri nets without explicitly modelling what is done with the car. In figure 2 one agent wants to own the good which is expressed by *!own good* and the ! characterises the intent in all the representations.

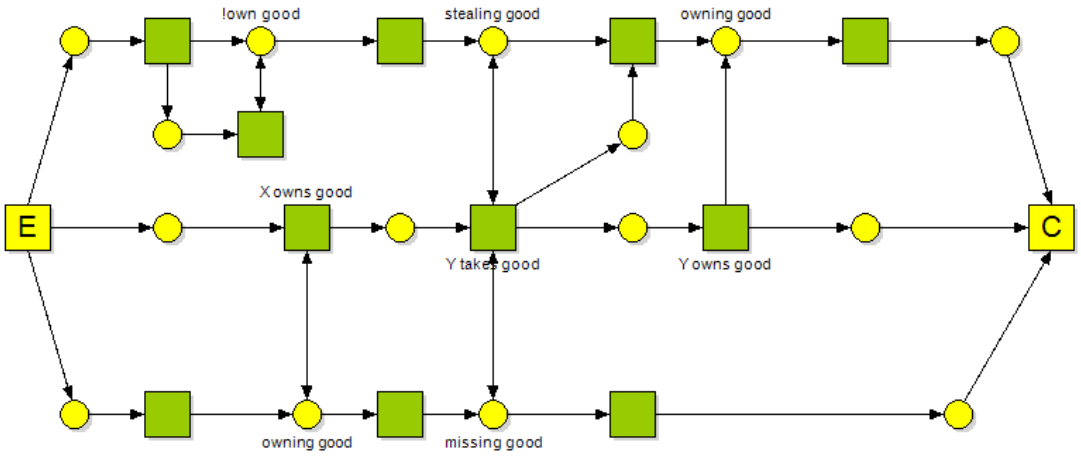

Figure 2: TheftStory - Petri net representation

3.2.2 Labeling

The following points are uphold for labeling the Petri nets: transitions are labelled as *offer made* or *good sold*, thus, each verb is after the noun and it consists of two or three words. Places are labelled

as *making offer* or *waiting for goods*. Each verb is a gerund and the length of the phrase is the same as with transitions. Furthermore the grammar is as small as possible and no synonyms are used to avoid differences while the same events are modelled.
Not each transition or place is labelled in the whole Petri net and the arcs are not labelled. The transitions of the message layer are labelled but the places are not labelled, however in the agent layers the places are labelled and the transitions are only labelled when a failure occurs as seen in figure 1. The loops which are used to display interaction between the layers without losing the flow of the tokens are only labelled in one place.

3.2.3 Execution

The program Yasper (ASPT, 2005) provides a system which executes the Petri net and displays the flow of the tokens in the net itself. In this case only one token is present in one place at the time when this is not the case then the Petri net is not modelled accurately and needs to be redefined. Another point is that each transition and place needs to be visited otherwise not the whole story is executed and there most not be a token left in the net when the collector is already activated. When the Petri net executes well and is modelled as described above then it can be used for the next phase.

3.3 Comparison measure

The short stories represented in Petri nets are different from eachother and the resemblance and differences of the stories need to be analyzed correctly. After examining the Petri nets three methods could be used to compare the stories and compare them in different ways to find other results. Each one of the methods is examined on its advantages and usefulness.

3.3.1 Label similarity

Transitions and places are labeled in the Petri nets and the same wording or phrases are used to indicate similar events. For example in the *Sale story* and the *Failed Sale 2 story* an offer is made in the beginning by an agent and this place is labeled as *making offer* (figure 1). It is in both stories the same event and therefore has exactly the same label. The structure of the wording in all the labels is consistent as far as possible and for places this means that the verb is before the noun and for transitions it is vice versa.
Thus the labels are generally alike eachother. However the placing of these labels differentiate for each Petri net. The event *buying* occurs earlier in agent layer in the *Complicated sale story* then the *Sale story*. Also, not all labels occur in each Petri net. Both techniques combined can compare Petri nets on their labels. Dijkman et al. (2011) called this technique on business models node similarity and uses cosine similarity to calculate the difference between the two labels. Cosine similarity computes the similarity between two vectors in a space. It maps the labels in such a space and then calculates the difference of the angle with the use of the cosine.
Label similarity does not take into account the flow of the tokens and therefore not the execution of the story which is important in the legal domain to show why a person did the action.
In Petri nets label similarity could be done on each independent layer to find out if the story is the same but the agents act differently.

3.3.2 Structure similarity

Each Petri net has a message layer and agent layers and it has an emitter and a collector to fire and to catch the tokens as well. Between these two nodes the structure depends on the story represented.
Calculating the count of steps it takes to transform one Petri net into another is a measure of structural comparison. Dijkman et al. (2011) and Löwe (2011) prove that structural comparison are efficient ways

of comparing two models. Structure similarity accounts for the complexity of a story however it does not account for the subject of the story which label similarity does account for, however it can also not account for the flow of the tokens.

3.3.3 Difference in execution

Execution similarity is based on the flow of the tokens and it changes for each Petri net. Therefore each execution is unique to the story and could potentially be a similarity measure that does not look at the mechanical way of the Petri nets but at the flow and behaviour. However it depends on the structure of the Petri net and thus the complexity of a story needs to be taken into account as well when comparing both Petri nets on execution.
Dijkman et al. (2011) suggests something similar with his behaviour similarity but he uses business models which are different in respect of Petri nets because of the tokens.

4 Implementation

The program is written in Groovy and the program IntelliJ IDEA (Jetbrains, 2015) is used to execute it. The Petri nets are read into the program and broken up into a transition list, places list and an arc list and this part of the program was already provided for by Sileno and Boer (2014). All the lists are not sorted and only convey the charactistics of that node, which include the label, the given id and other properties depending on the node.
Label similarity and structure similarity are implemented in the file *comparison.groovy* and executed over all the stories to find the similarities between them. The file *connection.groovy* consists of functions used in the other file.

4.1 Label comparison

Label comparison takes both the transitions and the places list to find all the nodes with labels and it compares each label of two stories to eachother. After which the labels it calculates the distance from the emitter to the current place or transition by adding the number of steps taken and uses the difference between the distance to compute the similarity with the other label. The calculation made is:

$$\frac{2 * Difference}{Lengthoflabellednodes}$$

Length of labelled nodes consists of the total number of node from both Petri nets which are labelled. The results should be a number between 0 and 1 with 1 as the highest similarity. However when tested on two exactly the same stories it returned 0 instead of 1. Therefore another implementation was necessary to get accurate results.
The second implementation also compares the labels of both stories but calculates the difference between them with cosinus similarity, which is a build-in function of Groovy. It uses the same calculation as the other implementation to compute the similarity between the two stories. The results are also from 0 to 1 with 1 meaning that the labelling is exactly similar.
However there is no preprocessing done on the wording of phrases and a misspelling or ambiguity could lead to a difference in similarity while it should be counted as a similar label but this is avoided by labelling the stories the same.

4.2 Structural comparison

The arc list is used for the structural comparison to create a connection which counts the number of incoming and outgoing arcs for each place or transition. It starts with the emitter and finds the next nodes by looking into the connection list and the function does this each time for both stories.
It checks the two nodes on outgoing arcs and looks for the next connection. Each time a node is visited it transfers the node from the connection list to a visited list thus it does not repeat the same node. Nodes with multiple outgoing arcs have two connection options and when one is visited the program needs to go back to do the other option as well. The second connection is therefore placed in a remember list which is used when there is no next connection.
The following pseudocode is the loop which is executed to see if an connection needs to be inserted or deleted.

```
while connectionlist is not empty do:
    if outgoing arcs are not zero:
        if outgoing arcs are equal:
            Find next connection in list
        if outgoing arcs for first story is greater:
             Add a connection
             Find next connection for first story
             and keep new connection
        if outgoing arcs for first story is lower:
            Delete connection
            Find next connection
    else reached the end:
        if startnode is left:
            Begin with startnode
        else:
            Find next connection in remember list
```

Finding the next connection is based on the connection list and if there is no other connections then there are two other options. The first option is to find another start connection which has not been explored before and the second option is the remember list where other connections are stored. It always finds a next connection depending on these three options.
During the while loop the number of insertions and deletions are counted and at the end they are added up to eachother. This is the measure used to compare structural similarity.

5 Evaluation

The story are compared to eachother on three different fronts. First the stories are ranked by human comparison after which the program computes the label similarities and structural similarities. These results are ranked as well and are correlated with the human comparison by using Spearman's rank correlation matrix algorithm.

5.1 Human comparison

The stories are evaluated beforehand on their similarities by a human. The twenty stories are ranked in table 1 from 1 till 20. There are not compared to itself thus the x in the table and each story has a different ranking therefore the highest rank for one story could be equivalently lower for the compared story as seen with most of the stories.
When evaluating the stories three aspects arose: subject, events and intent and the stories are ranked higher when they are positively combined.

5.1.1 Subject

There are five big subjects among the stories: sale, theft, burglary, robbery and good-exchange which are stories that do not mention sale and the swap or gift story belong in this category.
Subject is an important aspect which means that it ranks higher than the other aspects as the table 1 shows when looking at the subject sale. These stories are in eachother top ten.

5.1.2 Events

Same events rank higher as well, like the event *buying* or *making an offer* and multiple similar events in an story can lead to higher similarity. Therefore *Failed Sale 2* and *Sale* will rank high even though the intent is different.

5.1.3 Intent

In legal stories the intent of the agents is important. Stories with bad intent will be associated with eachother and therefore intent is also important when ranking stories. The stories *Burglary False* and *Theft False* are high ranked to eachother because they share the same intent while the subject and events differentiate.
However intent is less important than subject. Sale stories will be paired together even though one might be malfunctioning of social institution like the *Failed Sale 2*. *Higher offer* is a story which does not have a specific intent and is therefore categorized in the middle rank of all the stories.

5.2 Computational comparison

Table 2 displays the result of the label comparison measure and from this table there can be concluded that label similarity depends heavily on the aspects subject and events, because they have more exact labels. However it does not take intent into account and therefore it only groups by events.
When short stories are compared with larger stories the results are low even though the subject is the same. *Failed Sale 1* is a short story and only returns low results.
Table 3 displays the ranking of the results of the label comparison measure. *Failed Sale 2* receives a high rank in six other stories and these stories all involve sale. However this subject grouping does not apply to the *Gift* story where two theft stories are ranked as second but the *Gift* story has a lot of events in common even though human comparison will not place them together as one subject.

Table 4 shows the results of the structural comparison and in this table the number of total insertions and deletions are displayed. However comparing two exact same stories to eachother should result in a 0 and some stories do not have a zero. This could be because of the multiple ways to get from the emitter to the collector but it does not account for uneven numbers and the program does an insertion or deletion too many which it should not do.
The rankings are shown in table **??** and the rankings are scattered over multiple subjects and events. This is because structural comparison only looks ate the number of arcs, places and transitions and it does not include label matching, therefore it computes the difference in complexity of the stories and short stories have a higher ranking to eachother.

Table 1: Manual - Ranking of the stories

	Burglary False	Burglary	Complicated Sale 1	Complicated Sale 2	Failed Sale 1	Failed Sale 2	Gift	Higher Offer	Loan	Offer Extortion	Offer Invitation	Robbery succes	Robbery Unsucces	Sale	Swap Evasion	Swap	Theft false	Theft joy	Theft rather	Theft
BurglaryFalse	x	**2**	8	9	12	13	5	6	10	4	19	7	18	17	14	6	**1**	**3**	15	16
Burglary	6	x	15	14	17	9	19	11	13	8	18	**3**	4	16	10	12	7	5	**1**	**2**
ComplicatedSale1	14	19	x	**1**	5	**3**	11	7	10	6	4	18	17	**2**	9	8	12	13	15	16
ComplicatedSale2	14	19	**1**	x	6	**3**	11	7	10	5	4	18	17	**2**	8	9	12	13	15	16
FailedSale1	14	17	**3**	4	x	**1**	9	5	8	11	7	19	18	**2**	10	6	12	13	16	15
FailedSale2	19	14	**3**	5	**2**	x	10	4	9	6	8	15	16	**1**	12	11	18	17	13	7
Gift	11	17	7	8	6	9	x	13	**1**	18	**3**	19	16	5	**2**	4	10	12	14	15
HigherOffer	16	15	9	10	4	**2**	16	x	11	5	**1**	17	18	**3**	12	6	14	8	13	7
Loan	12	19	7	8	6	10	**1**	13	x	9	**2**	18	17	5	4	**3**	11	14	16	15
OfferExtortion	14	**3**	11	10	13	7	19	8	17	x	18	**1**	**2**	12	9	16	15	5	6	4
OfferInvitation	7	17	9	10	**3**	13	**1**	**2**	4	19	x	16	15	8	12	5	6	11	14	18
RobberySucces	15	**2**	11	9	12	6	19	10	17	8	18	x	**1**	13	7	16	14	5	4	**3**
RobberyUnsucces	14	**2**	11	9	13	7	19	10	17	5	18	**1**	x	12	8	16	15	6	4	**3**
Sale	13	17	4	5	**3**	**1**	10	8	11	9	7	19	18	x	6	**2**	12	14	16	15
SwapEvasion	13	18	5	6	8	9	7	11	**3**	10	4	19	16	**2**	x	**1**	12	14	15	17
Swap	13	19	4	5	6	8	10	7	9	11	**2**	18	17	**3**	**1**	x	12	14	15	16
TheftFalse	5	**1**	15	16	12	14	8	11	9	19	10	7	6	13	18	17	x	**2**	4	3
ThefUoy	4	7	15	14	17	10	19	11	12	9	18	5	6	16	8	13	**1**	x	**2**	**3**
TheftRather	**1**	11	14	13	15	6	19	10	16	5	18	4	8	12	7	17	9	**3**	x	**2**
Theft	5	11	14	13	16	7	19	10	18	4	17	6	8	15	9	12	**3**	**1**	**2**	x

Table 2: Label similarity

	Burglary False	Burglary	Complicated Sale 1	Complicated Sale 2	Failed Sale 1	Failed Sale 2	Gift	Higher Offer	Loan	Offer Extortion	Offer Invitation	Robbery succes	Robbery Unsucces	Sale	Swap Evasion	Swap	Theft false	Theft joy	Theft rather	Theft
BurglaryFalse	1	0.538	0.0758	0.0780	0.111	0.051	0.191	0.060	0.196	0.179	0.126	0.155	0.146	0.055	0.111	0.049	0.178	0.159	0.325	0.111
Burglary	0.538	1	0	0.019	0	0.065	0.099	0	0.042	0.056	0.029	0.121	0.167	0	0.048	0.056	0.048	0.130	0.603	0.149
ComplicatedSale1	0.076	0	1	0.513	0.159	0.323	0.254	0.194	0.253	0.190	0.272	0.104	0.099	0.280	0.151	0.184	0.230	0.212	0	0.23042
ComplicatedSale2	0.078	0.019	0.513	1	0.092	0.220	0.189	0.154	0.173	0.162	0.218	0.100	0.097	0.150	0.143	0.136	0.121	0.113	0	0.121
FailedSale1	0.111	0	0.159	0.092	1	0.370	0.029	0.276	0.074	0.095	0.167	0.078	0.074	0.286	0	0.290	0	0	0	0
FailedSale2	0.051	0.065	0.323	0.220	0.370	1	0.017	0.533	0.046	0.053	0.149	0.089	0.114	0.739	0	0.381	0	0	0.037	0
Gift	0.191	0.099	0.254	0.189	0.029	0.017	1	0.105	0.533	0.379	0.386	0.126	0.063	0.034	0.262	0.099	0.390	0.347	0	0.390
HigherOffer	0.059	0	0.194	0.154	0.276	0.533	0.105	1	0.126	0.219	0.282	0.083	0.080	0.618	0.123	0.345	0.089	0.083	0	0.089
Loan	0.196	0.042	0.253	0.173	0.074	0.046	0.533	0.126	1	0.375	0.355	0.105	0.100	0.067	0.259	0.132	0.455	0.430	0	0.399
OfferExtortion	0.179	0.056	0.190	0.162	0.095	0.053	0.379	0.219	0.375	1	0.506	0.268	0.250	0.077	0.310	0.097	0.249	0.218	0	0.249
OfferInvitation	0.126	0.029	0.272	0.218	0.167	0.149	0.386	0.282	0.355	0.506	1	0.166	0.158	0.167	0.264	0.227	0.264	0.238	0	0.264
RobberySucces	0.155	0.121	0.104	0.100	0.078	0.089	0.126	0.083	0.105	0.267	0.166	1	0.825	0.086	0	0.078	0	0	0.133	0
RobberyUnsucces	0.146	0.167	0.099	0.097	0.074	0.114	0.063	0.080	0.100	0.250	0.158	0.825	1	0.083	0	0.075	0	0	0.125	0
Sale	0.055	0	0.280	0.149	0.286	0.739	0.034	0.618	0.067	0.077	0.167	0.086	0.083	1	0.029	0.457	0.029	0.027	0	0.029
SwapEvasion	0.111	0.048	0.151	0.143	0	0	0.262	0.123	0.259	0.310	0.264	0	0	0.029	1	0.053	0.292	0.259	0	0.2916667
Swap	0.049	0.056	0.184	0.136	0.290	0.381	0.099	0.345	0.132	0.097	0.227	0.078	0.0754	0.457	0.053	1	0.074	0.069	0.028	0.074
TheftFalse	0.178	0.048	0.230	0.121	0	0	0.391	0.089	0.455	0.249	0.264	0	0	0.029	0.292	0.074	1	0.833	0	0.771
ThefJoy	0.157	0.130	0.212	0.113	0	0	0.347	0.083	0.430	0.218	0.238	0	0	0.027	0.259	0.069	0.833	1	0.088	0.741
TheftRather	0.325	0.603	0	0	0	0.033	0	0	0	0	0	0.133	0.125	0	0	0.028	0	0.088	1	0.101
Theft	0.111	0.149	0.230	0.121	0	0	0.390	0.089	0.399	0.249	0.264	0	0	0.029	0.292	0.074	0.771	0.741	0.101	1

Table 3: Ranking of label similarity

	Burglary False	Burglary	Complicated Sale 1	Complicated Sale 2	Failed Sale 1	Failed Sale 2	Gift	Higher Offer	Loan	Offer Extortion	Offer Invitation	Robbery succes	Robbery Unsucces	Sale	Swap Evasion	Swap	Theft false	Theft joy	Theft rather	Theft
BurglaryFalse	x	**1**	15	14	11	18	4	16	**3**	5	10	8	9	17	11	19	6	7	**2**	11
Burglary	**2**	x	16	15	16	8	7	16	13	10	14	6	**3**	16	11	9	11	5	**1**	4
ComplicatedSale1	17	18	x	**1**	13	**2**	5	10	6	11	4	15	16	**3**	14	12	7	9	18	7
ComplicatedSale2	17	18	**1**	x	16	**2**	4	7	5	6	**3**	14	15	8	9	10	11	13	19	11
FailedSale1	7	14	6	9	x	**1**	13	4	11	8	5	10	11	**3**	14	**2**	14	14	14	14
FailedSale2	12	10	5	6	4	x	15	**2**	13	11	7	9	8	**1**	16	**3**	16	16	14	16
Gift	9	14	8	10	17	18	x	12	**1**	5	4	11	15	16	7	13	**2**	6	19	**2**
HigherOffer	17	18	7	8	5	**2**	11	x	9	6	4	15	16	**1**	10	3	12	14	18	12
Loan	9	18	8	10	15	17	**1**	12	x	5	6	13	14	16	7	11	**2**	**3**	19	4
OfferExtortion	12	17	11	13	15	18	**2**	9	**3**	x	**1**	5	6	16	4	14	7	10	19	7
OfferInvitation	17	18	5	11	12	16	**2**	4	**3**	**1**	x	14	15	12	6	10	6	9	19	6
RobberySucces	4	7	9	10	14	11	6	13	8	**2**	**3**	x	**1**	12	16	15	16	16	5	16
RobberyUnsucces	5	**3**	9	10	14	7	15	12	8	**2**	4	**1**	x	11	16	13	16	16	6	16
Sale	12	18	5	7	4	**1**	13	**2**	11	10	6	8	9	x	14	**3**	14	17	18	14
SwapEvasion	11	13	8	9	15	15	5	10	6	**1**	4	15	15	14	x	12	**2**	6	15	**2**
Swap	18	16	6	7	4	**2**	9	**3**	8	10	5	11	12	**1**	17	x	13	15	19	13
TheftFalse	9	13	8	10	15	15	4	11	**3**	7	6	15	15	14	5	12	x	**1**	15	**2**
ThefJoy	9	10	8	11	16	16	4	13	**3**	7	6	16	16	15	5	14	**1**	x	12	**2**
TheftRather	**2**	**1**	9	9	9	7	9	9	9	9	9	**3**	**4**	9	9	8	9	6	x	5
Theft	11	9	8	10	16	16	4	13	**3**	7	6	16	16	15	5	14	**1**	**2**	12	x

Table 4: Structural similarity - number of insertions and deletions

	Burglary False	Burglary	Complicated Sale 1	Complicated Sale 2	Failed Sale 1	Failed Sale 2	Gift	Higher Offer	Loan	Offer Extortion	Offer Invitation	Robbery succes	Robbery Unsucces	Sale	Swap Evasion	Swap	Theft false	Theft joy	Theft rather	Theft
BurglaryFalse	0	12	57	69	10	55	37	80	37	26	32	36	33	62	36	51	30	39	35	30
Burglary	12	1	54	64	7	56	44	76	34	24	34	32	32	64	31	46	29	38	29	29
ComplicatedSale1	57	54	11	58	28	66	41	80	46	57	40	34	34	70	46	60	24	33	33	24
ComplicatedSale2	69	64	58	8	40	90	72	88	59	63	73	55	46	91	57	71	66	70	58	66
FailedSale1	10	7	28	40	0	55	45	73	37	22	25	32	32	63	34	43	29	38	29	29
FailedSale2	55	56	66	90	55	1	55	62	53	57	24	41	41	11	51	37	38	47	31	38
Gift	37	44	41	72	45	55	0	82	47	27	42	36	38	76	41	54	57	45	21	57
HigherOffer	80	76	80	88	73	62	82	40	42	39	33	34	37	60	49	40	42	51	35	42
Loan	37	34	46	59	37	53	47	42	5	35	27	33	32	81	48	60	27	36	42	27
OfferExtortion	26	24	57	63	22	57	27	39	35	8	30	40	41	68	34	46	34	43	29	34
OfferInvitation	32	34	40	73	25	24	42	33	27	30	5	30	35	52	41	39	29	38	44	29
RobberySucces	36	32	34	55	32	41	36	34	33	40	30	0	13	70	47	59	26	36	44	26
RobberyUnsucces	33	32	34	46	32	41	38	37	32	41	35	13	0	69	47	66	28	37	47	28
Sale	62	64	70	91	63	11	76	60	81	68	52	70	69	1	52	43	37	46	31	37
SwapEvasion	36	31	46	57	34	51	41	49	48	34	41	47	47	52	0	51	36	45	31	36
Swap	51	46	60	71	43	37	54	40	60	46	39	59	66	43	51	0	38	47	26	38
TheftFalse	30	29	24	66	29	38	57	42	27	34	29	26	28	37	36	38	0	10	27	0
ThefJoy	39	38	33	70	38	47	45	51	36	43	38	36	37	46	45	47	10	5	40	10
TheftRather	35	29	33	58	29	31	21	35	42	29	44	44	47	31	31	26	27	40	1	28
Theft	30	29	24	66	29	38	57	42	27	34	29	26	28	37	36	38	0	10	28	0

Table 5: Ranking of structural similarity

	Burglary False	Burglary	Complicated Sale 1	Complicated Sale 2	Failed Sale 1	Failed Sale 2	Gift	Higher Offer	Loan	Offer Extortion	Offer Invitation	Robbery succes	Robbery Unsucces	Sale	Swap Evasion	Swap	Theft false	Theft joy	Theft rather	Theft
BurglaryFalse	x	**2**	16	18	**1**	15	11	19	11	**3**	6	9	7	17	9	14	4	13	8	4
Burglary	**2**	x	15	17	**1**	16	13	19	10	**3**	10	8	8	17	7	14	4	12	4	4
ComplicatedSale1	13	12	x	15	**3**	17	9	19	10	13	8	6	6	18	10	16	**1**	4	4	**1**
ComplicatedSale2	12	9	5	x	**1**	18	15	17	7	8	16	**3**	**2**	19	4	14	10	13	5	10
FailedSale1	**2**	**1**	5	14	x	17	16	19	12	**3**	4	9	9	18	11	15	6	13	6	6
FailedSale2	12	15	18	19	12	x	12	17	11	16	**2**	7	7	**1**	10	4	5	9	**3**	5
Gift	4	9	6	17	10	14	x	19	12	**2**	8	**3**	5	18	6	13	15	10	**1**	15
HigherOffer	15	16	17	19	13	14	19	x	8	5	**1**	**2**	4	13	11	6	8	12	**3**	8
Loan	9	6	13	17	9	16	14	11	x	7	**1**	5	4	19	15	18	**1**	8	11	**1**
OfferExtortion	**3**	**2**	16	18	**1**	16	4	11	10	x	6	12	13	19	7	15	7	14	5	7
OfferInvitation	8	10	14	19	**2**	**1**	16	9	**3**	6	x	6	11	18	15	13	4	12	17	4
RobberySucces	10	5	8	17	5	14	10	8	7	13	4	x	**1**	19	16	18	**2**	10	15	**2**
RobberyUnsucces	7	4	8	15	4	13	12	10	4	13	9	**1**	x	19	16	18	**2**	10	16	**2**
Sale	10	12	15	19	11	**1**	17	9	18	13	7	15	14	x	7	5	**3**	6	**2**	**3**
SwapEvasion	5	**1**	11	19	**3**	16	8	15	14	**3**	8	12	12	18	x	16	5	10	**1**	5
Swap	12	9	16	19	7	**2**	14	6	16	9	5	15	18	7	12	x	**3**	11	**1**	**3**
TheftFalse	11	8	**3**	19	8	15	18	17	5	12	8	4	7	14	13	15	x	**2**	5	**1**
ThefJoy	10	7	**3**	19	7	16	13	18	4	12	7	4	6	15	13	16	**1**	x	11	**1**
TheftRather	12	5	11	19	5	8	**1**	12	15	5	16	16	18	8	8	**2**	**3**	14	x	4
Theft	11	8	**3**	19	8	15	18	17	5	12	8	4	6	14	13	15	**1**	**2**	6	x

	Structure	Label
BurglaryFalse	0,036	0,311
Burglary	0,414	0,750
ComplicatedSale1	-0,521	0,707
ComplicatedSale2	-0,360	0,753
FailedSale1	-0.278	0,683
FailedSale2	-0,332	0,601
Gift	-0,341	0,162
HigherOffer	0,049	0,824
Loan	-0,546	0,353
OfferExtortion	-0,164	-0,271
OfferInvitation	-0,059	0,336
RobberySucces	0,223	-0,05
RobberyUnsucces	0,109	0,253
Sale	-0,346	0,026
SwapEvasion	0,082	0,026
Swap	-0,021	0,588
TheftFalse	0,43	0,046
TheftJoy	0,382	0,112
TheftRather	-0,281	0,518
Theft	0,303	0,001

Table 6: Correlation matrix of manual against structure and label

5.3 Correlation matrix

Spearman's rank correlation matrix algorithm is used to compare the ranking of each story from each measure to the manual ranking. However Spearman's ranking cannot compare ranks with the mulitple similar rankings, therefore an expansion is used, called PvRank, to computed the correlation matrix. PvRank (Amerise et al., 2015) is an package for R (for statistical Computing, 2014) with different correlation matrices and the algorithm *rankor* is used to create the ρ which stands the correlation between the two rankings. It ranges from -1 till 1 where 1 means positively correlated and -1 negatively correlated. The usage of the function rankor is:

rankor(x, y, index = "spearman", approx = "exact", tiex = "woodbury", CC = FALSE, type = "two-sided", print = TRUE, sizer = 1000000, repl = 1000).

Where x an y are the ranked stories which need to be compared and the index says which correlation algorithm to use. The variables sizer and repl are used for the same ranking problem. They are the number of random steps it will do to create a new ranking when it encounters a similar ranking.

The results are displayed in table 6. The ranking of the label comparison is mostly positively correlated and *OfferExtortion* and *RobberySucces* stories are low negativily correlated. The ranking of structural comparison, however, is mostly negatively correlated because human comparison is not based on the length or complexity of the story.
In both cases the *Burglary* story has a high correlation because it is a medium-sized story and has the typical labels for the subjects theft and burglary. The *Loan* story has the lowest correlation but only with structural comparison because it is a small story. However it has a high frequency of labels which can be compared well with the label comparison

6 Conclusion & Discussion

In conclusion, the label comparison measure is better than the structural comparison to compare stories on subject and events which two aspects of human comparison. However it does not account for intent or the storyline and it is a mechanical way of comparing two stories which was not the comparison measure this thesis wanted to find.
The results of the structural comparison measure are skewed, because two similar stories do not result in 0 insertions or deletions and the multiple way to go from the emitter to the collector do not account for fawlty measures. When inserting and deleting nodes the program probably sets too many nodes in the remember list, which it does not visit again therefore it forgets these nodes while it deletes or inserts another one.
The weakness of the label comparison measure is that it does not do label matching and therefore it only matches on exact labels. For example *waiting for goods* is not considered the same as *waiting for good* and *buying* is not the same as *buys*. Whereas this could be an extension to get a higher correlation. However comparing exact labels has an advantage as well. All of the stories overlap because they are from the same domain and exact label matching provides strong difference in the results. While partial label matching can flatten the results by matching keywords which are used in every legal story.
This thesis shows us that implementing one of these measures is not accurate enough for comparing stories in a humanlike manner. A combination of label and structural comparison could be a solution to achieve a higher correlation between the rankings, because it should insert and delete transitions and places based on their labels instead of just inserting and deleting when necessary. The labels are also compared with the cosine similarity and therefore it creates a measure which looks at the complexity of the story and the subject.
The difference in execution should be a measure to compare stories based on intent or the storyline, because it follows the firing of the tokens instead predefined labels or structure. This could receive a positive correlation with the manual ranking because it is less mechanical as label comparison or structural. It is explored in this thesis but not implemented.

7 Future work

There is not yet a conventional way to model and represent stories in Petri nets. Future research could be done to develop guidelines for this, thus the Petri nets can have similar representations for negation or failure.
Next research could be implementing the difference in execution measure. This could be very promising, because of its potential to represent the storyline and the flow of the tokens. The execution of the Petri net can also be added to the structural comparison to create a structural comparison where the tokens are taking into consideration.
Other future work is to correct the structural comparison measure and to receive better results which could be used to create a new correlation matrix.
As mentioned in section 6 a combination of label comparison and structural comparison could lead to an higher correlation as well. Combining this with the execution plan could maybe result in an improved correlation and therefore the comparison measure could imitate human comparison better.
In the end there is no complete comparison measure to simulate humanlike comparison yet.

8 Bibliography

Amerise, I., Marozzi, M., and Tarsitano, A. (2015). pvrank: Rank correlations. `www.cran.r-project.org/web/packages/pvrank/index.html`.

ASPT, S. (2005). Yasper. `https://www.yasper.org`.

Behrens, T. and Dix, J. (2007). Model checking multi-agent system with logic based petri nets. *Annals of Mathematics and AI*, 51:81–121.

Dijkman, R., Dumas, M., van Dongen, B., Käärik, R., and Mendling, J. (2011). Similarity of business process models: Metrics and evaluation. *Information Systems*, 36:498–516.

Fisseni, B. and Löwe, B. (2012). Which dimensions of narrative are relevant for human judgements of story equivalence? *Workshop on Computational Models of Narrative*, 3:114–118.

for statistical Computing, T. R. F. (2014). R. `https://www.r-project.org`.

Jetbrains (2015). Intellij idea. `https://www.jetbrains.com/idea`.

Lehmann, J., Breuker, J., and Brouwer, B. (2005). Modeling causation in ai & law. *Law and the semantic web*, pages 77–96.

Löwe, B. (2011). Methodological remarks about comparing formal frameworks for narratives. *Proceedings of the 3rd Workshop in the Philosophy of Information*, pages 45–63.

Murata, T. (1989). Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.

Schank, R. and Abelson, R. (1979). Scripts, plans, goals and understanding, an inquiry into human knowledge structures. *Journal of Pragmatics*, 3(2):211–217.

Sileno, G. and Boer, A. (2014). Legal knowledge conveyed by narratives: Towards a representational model. *Proceedings of the Workshop on Computational Models of Narrative (CMN 2014)*, pages 182–191.

A Short stories

SaleStory
X makes an offer. The offer concerns a good exchanged for a certain amount of money. Y accepts the offer. Y pays the money. X delivers the good.

FailedSale1Story
X makes an offer. The offer concerns a good exchanged for a certain amount of money. Nobody accepts the offer.

FailedSale2Story
X makes an offer. The offer concerns a good exchanged for a certain amount of money. Y accepts the offer. Y pays the money. X does not deliver.

ComplicatedSale1Story
X buys a good from Y. Y sends the good to X. The good does not arrive. X complains to Y. The good arrives in the meanwhile.

ComplicatedSale2Story
X buys a good from Y. Y sends the good to X. The good does not arrive. X complains to Y. Y asks to wait some says more. The good does not arrive. X complains to Y, again. Y sends another good to X.

GiftStory
X wishes a certain good. Y gives that good to him.

SwapStory
X makes an offer. The offer concerns a good exchanged for another good. Y accepts the offer. X delivers the good. Y delivers the other good.

SwapEvasionStory
Goods of type A and B usually cost a certain price. When you sell a good of type A or B you have to pay taxes on it. X sells a good A to Y for half of the normal price. Y sells a good B to X for half of the normal price.

LoanStory
X needs a certain good. Y lends that good to him. X gives the good back to X.

TheftStory
X owns a certain good. Y takes it with him

TheftFalseStory
X owns a car. Y takes it to bring someone to the urgencies. [theft is characterized by the intent of permanently deprive the owner of it]

TheftJoyStory
X owns a beautiful car. Y takes the car without consent, drives it for a spin and returns it where it was.

BurglaryStory
X enters in the house of Y, in order to steal something. [burglary is characterized by entering without

consent, with criminal intents]

TheftRatherStory
Y invites X to his house. X enters in the house of Y, in order to steal something

BurglaryFalseStory
X enters in the house of Y, as the building was in flame and was trying an escape.

RobberySuccesStory
Y is cashier at the supermarket. X threatens to shoot Y unless he hands over the loot. Y hands over the loot.

RobberyUnsuccesStory
Y is cashier at the supermarket. X threatens to shoot Y unless he hands over the loot. Y says there is no money, it is too early. X sees that there is no money, and runs.

OfferInvitationStory
X offers Y to sell a good for a certain amount of money. Y accepts and sells the good to X.

OfferExtortionStory
X threats Y to sell a good for a certain amount of money. Y sell the good to X.

HigherOfferStory
X offers to sell good to Y. Y accepts the offer. Z offers to buy good at higher price. X accepts Z's offer. X sells good to Z.

B Groovy code

B.1 comparison.groovy

```
package org.leibnizcenter.pneu.comparison

import groovy.util.logging.Log4j
import org.leibnizcenter.pneu.components.petrinet.Net
import org.leibnizcenter.pneu.components.petrinet.Place
import org.leibnizcenter.pneu.components.petrinet.Node
import org.leibnizcenter.pneu.components.petrinet.Transition

import org.simmetrics.StringMetric;
import org.simmetrics.StringMetrics;

@Log4j
class Comparison {

    ////////////////// LABEL COMPARISON

    static Float labelComparison(Net sourceNet, Net targetNet) {
        List<Place> sourcePlaces = sourceNet.placeList
        List<Place> targetPlaces = targetNet.placeList

        List<Transition> sourceTransitions = sourceNet.transitionList
        List<Transition> targetTransitions = targetNet.transitionList

        //  With empty nodes:
        // Integer lengthPlace = sourcePlaces.size() + targetPlaces.size()

        // Define length without empty nodes
        Integer lengthSourcePlaces = sourcePlaces.findAll() { it.name != "" }.size()
        Integer lengthTargetPlaces = targetPlaces.findAll() { it.name != "" }.size()

        Integer lengthPlaces = lengthSourcePlaces + lengthTargetPlaces

        Integer diff1 = 0
        Float simPlace = 0
        Float simTrans = 0

        for (i in sourcePlaces) {
            Float sim = 0
            Integer diff2 = 0
            if (i.name != "") {
                for (j in targetPlaces) {
                    Integer diff = Math.abs(diff2 - diff1)
                    log.trace("comparing "+i.name + "=?=" + j.name)
                    if (i.name == j.name) {
                        sim = (2 * diff) / lengthPlaces
                        log.trace("found equal: $sim $diff $lengthPlaces")
                        simPlace = simPlace + sim
                    }
                    diff2++
                }
            }
            diff1++
        }

        Integer lengthSourceTransitions = sourceTransitions.findAll() { it.name != "" }.size()
        Integer lengthTargetTransitions = targetTransitions.findAll() { it.name != "" }.size()

        Integer lengthTransitions = lengthSourceTransitions + lengthTargetTransitions

        diff1 = 0

        for (i in sourceTransitions) {
            Float sim = 0
            Integer diff2 = 0
            if (i.name != "") {
                for (j in targetTransitions) {
                    Integer diff = Math.abs(diff2 - diff1)
                    if (i.name == j.name) {
                        sim = (2 * diff) / lengthTransitions
                        simTrans = simTrans + sim
                    }
                    diff2++
                }
            }
            diff1++
        }

        // Calculate label similarity
        return (2 * (simPlace + simTrans)) / (lengthPlaces + lengthTransitions)

    }

    static class NodeComparisonValue {
        Float value
        Boolean active

        String toString() { return "{$value, $active}" }
    }
```

```
    static class NodeComparisonKey {
        Node source
        Node target

        String toString() { return "{${source.id}, ${target.id}}" }
    }

    static Float labelComparison2(Net sourceNet, Net targetNet) {
        List<Place> sourcePlaces = sourceNet.placeList
        List<Place> targetPlaces = targetNet.placeList

        List<Transition> sourceTransitions = sourceNet.transitionList
        List<Transition> targetTransitions = targetNet.transitionList

        // source node, target node, similarity value, active
        Map<NodeComparisonKey, NodeComparisonValue> labelSimilarity = [:]

        StringMetric metric = StringMetrics.cosineSimilarity()

        // compute the similarity over all the nodes (no distinction between places and transitions)
        List<Node> sourceNodes = sourcePlaces + sourceTransitions
        List<Node> targetNodes = targetPlaces + targetTransitions

        // find how many nodes are labeled
        Integer nLabeledNodes = sourceNodes.findAll() { it.name != "" }.size() +
                                targetNodes.findAll() { it.name != "" }.size()

        for (Node i in sourceNodes) {
            for (Node j in targetNodes) {
                if (i.name && j.name)
//                  labelSimilarity[i, j] = metric.compare(i.name, j.name)
                  labelSimilarity[new NodeComparisonKey(source: i, target: j)] = new NodeComparisonValue(
                          value: metric.compare(i.name, j.name),
                          active: true
                  )
            }
        }

        // greedy algorithm
        Float max = 1
        Float similarity = 0

        while (max != 0) {
            Float newMax = 0

            // find max
            for (NodeComparisonKey key in labelSimilarity.keySet()) {
                NodeComparisonValue nodeComparisonValue = labelSimilarity[key]
                if (nodeComparisonValue.active) {
                    Float value = nodeComparisonValue.value
                    if (value == max) {
                        log.trace(key.toString() + " is a max ($max)")
                        similarity += value

                        // purge
                        for (NodeComparisonKey innerKey in labelSimilarity.keySet()) {
                            NodeComparisonValue nodeComparisonInnerValue = labelSimilarity[innerKey]
                            if (nodeComparisonInnerValue.active) {
                                if (key.source == innerKey.source || key.target == innerKey.target) {
                                    nodeComparisonInnerValue.active = false
                                    log.trace("remove " + key.toString())
                                }
                            }
                        }
                    } else {
                        if (newMax < value) {
                            log.trace("new potential max ($value)")
                            newMax = value
                        }
                    }
                }
            }

            max = newMax

        }

        // weighted by the number of labeled nodes
        return 2 * similarity / nLabeledNodes

    }

    ////////////////// STRUCTURAL COMPARISON


    /**********
     Structural

     - Use arcs to find target and source
     - Does not refer to labelling
     **********/

    def static structuralComparison(Net sourceNet, Net targetNet) {
```

```
List<Connection> sourceConnections
List<Connection> targetConnections

// count the number of incoming and outgoing arcs for source net
sourceConnections = Connection.readConnections(sourceNet)
// count the number of incoming and outgoing arcs for target net
targetConnections = Connection.readConnections(targetNet)

// find begin point
Connection sourceBeginPoint = Connection.searchCollector(sourceConnections)
Connection targetBeginPoint = Connection.searchCollector(targetConnections)

// initial values
List<Connection> visitedSourceConnections = []
List<Connection> visitedTargetConnections = []
Connection sourceNode = sourceBeginPoint
Connection targetNode = targetBeginPoint
List<Connection> newConnections = []
List<Connection> rememberSourceNodes = []
List<Connection> rememberTargetNodes = []
Integer addDiff = 0
Integer delDiff = 0

/*
  Inserting and deleting nodes if necessary,
  remember nodes if they have multiple arcs

  - Replaces nodes when visited into visitedConnection
  - finds next nodes depending on current node (end, beginning, remembered)
*/

while (sourceConnections.size() > 0 && sourceNode && targetNode) { // ADDED CHECK FOR VOID NODES?

    Connection nextSourceConnection, nextTargetConnection
    Boolean sourceBacktrack, targetBacktrack

    log.trace("######################### New cycle: sourceNode $sourceNode, targetNode $targetNode.")

    if (sourceNode.nTargetOutArcs != 0 && targetNode.nTargetOutArcs != 0 && targetNode.nTargetOutArcs!= -1) {
        log.trace("the sources of the arcs have both a number of output > 0..")

        if (sourceNode.nSourceOutArcs == targetNode.nSourceOutArcs) {
            log.trace("they have the same number of output arcs at the source")

            (nextSourceConnection, sourceBacktrack) = Connection.findNext(visitedSourceConnections, sourceConnections, sourceNode, remembe
            (nextTargetConnection, targetBacktrack) = Connection.findNext(visitedTargetConnections, targetConnections, targetNode, remembe

            // println "Next " + nextConnection + nextTargetConnection
            if (sourceBacktrack) {
                log.trace("backtrack!")
                nextTargetConnection = rememberTargetNodes[0]
                rememberSourceNodes.remove(nextSourceConnection)
                rememberTargetNodes.remove(nextTargetConnection)
            } else if(targetBacktrack){
                log.trace("backtrack!")
                nextSourceConnection = rememberSourceNodes[0]
                rememberSourceNodes.remove(nextSourceConnection)
                rememberTargetNodes.remove(nextTargetConnection)
            }

            List<Connection> rSourceNodes = Connection.remember(sourceConnections, sourceNode)
            List<Connection> rTargetNodes = Connection.remember(targetConnections, targetNode)
            if (rSourceNodes.size() > 0) {
                for (rSourceNode in rSourceNodes) {
                    for (rTargetNode in rTargetNodes) {
                        rememberSourceNodes << rSourceNode
                        rememberTargetNodes << rTargetNode
                        sourceConnections.remove(rSourceNode)
                        targetConnections.remove(rTargetNode)
                    }
                }
            }
            log.trace("Next $nextSourceConnection $nextTargetConnection")
        } else if (sourceNode.nSourceOutArcs > targetNode.nSourceOutArcs) { // Add
            log.trace("the first has more output than the second on the source, I add one the second")

            Integer diff = sourceNode.nSourceOutArcs - targetNode.nSourceOutArcs
            Connection.searchBySourceIdAdd(targetConnections, targetNode.sourceId, diff)

            for (int i = 0; i < diff; i++) {
                Connection newNode = new Connection(
                        sourceId: targetNode.sourceId,
                        targetId: 'n' + addDiff,
                        nSourceInArcs: targetNode.nSourceInArcs,
                        nSourceOutArcs: targetNode.nSourceOutArcs -1,
                        nTargetOutArcs: -1
                )
                newConnections.add(0,newNode)
                addDiff++
            }

            (nextSourceConnection, sourceBacktrack) = Connection.findNext(visitedSourceConnections, sourceConnections, sourceNode, remembe
            (nextTargetConnection, targetBacktrack) = Connection.findNext(visitedTargetConnections, targetConnections, targetNode, remembe

            if (sourceBacktrack) {
```

```
            log.trace("backtrack!")
            nextTargetConnection = rememberTargetNodes[0]
            rememberSourceNodes.remove(nextSourceConnection)
            rememberTargetNodes.remove(nextTargetConnection)
        }

        List<Connection> rSourceNodes = Connection.remember(sourceConnections, sourceNode)
        List<Connection> rTargetNodes = Connection.remember(targetConnections, targetNode)
        if (rSourceNodes.size() > 0) {
                rememberSourceNodes << rSourceNodes[0]
                rememberTargetNodes << newConnections[0]

                if (targetNode.nSourceOutArcs >= 2) {
                    rememberSourceNodes << rSourceNodes[1]
                    rememberTargetNodes << rTargetNodes[0]
                    sourceConnections.remove(rSourceNodes[1])
                    targetConnections.remove(rTargetNodes[0])
                }
                sourceConnections.remove(rSourceNodes[0])
        }

        println "Add " + nextSourceConnection + nextTargetConnection

    } else if (sourceNode.nSourceOutArcs < targetNode.nSourceOutArcs) { // del
        log.trace("the first has less outputs than the second on the source, I remove from the second")

        Integer diff = targetNode.nSourceOutArcs - sourceNode.nSourceOutArcs
        List<Connection> connectionList = Connection.searchBySourceIdAdd(targetConnections, targetNode.sourceId, -diff)

        for (int i = 0; i < diff; i++) {
            connectionList.remove(targetNode)
            if (connectionList[0])  // TOCHECK: there is some NULL around!
              visitedTargetConnections << connectionList[0]
            targetConnections.remove(connectionList[0])
            connectionList.remove(connectionList[0])
            delDiff++
        }

        (nextSourceConnection, sourceBacktrack) = Connection.findNext(visitedSourceConnections, sourceConnections, sourceNode, remembe
        (nextTargetConnection, targetBacktrack) = Connection.findNext(visitedTargetConnections, targetConnections, targetNode, remembe

        if (sourceBacktrack) {
            nextTargetConnection = rememberTargetNodes[0]
            rememberSourceNodes.remove(nextSourceConnection)
            rememberTargetNodes.remove(nextTargetConnection)
        }
        List<Connection> rSourceNodes = Connection.remember(sourceConnections, sourceNode)
        List<Connection> rTargetNodes = Connection.remember(targetConnections, targetNode)

        if (rSourceNodes.size() > 0) {
            for(rSourceNode in rSourceNodes){
                for(rTargetNode in rTargetNodes){
                    rememberSourceNodes << rSourceNode
                    rememberTargetNodes << rTargetNode
                    sourceConnections.remove(rSourceNode)
                    targetConnections.remove(rTargetNode)
                }
            }
        }

        println "Del " + sourceNode + targetNode
    }

// If one is an end node and the other one is not
} else if (sourceNode.nTargetOutArcs > 0 && targetNode.nTargetOutArcs == 0) {
    log.trace("the first is not an end node, the second yes")

    Connection newNode = new Connection(
            sourceId: "n"+addDiff,
            targetId: targetNode.targetId,
            nSourceInArcs: targetNode.nSourceInArcs,
            nSourceOutArcs: 1
    )
    newConnections << newNode
    addDiff++

    (nextSourceConnection, sourceBacktrack) = Connection.findNext(visitedSourceConnections, sourceConnections, sourceNode, rememberSou

    if (sourceBacktrack) {
        nextTargetConnection = rememberTargetNodes[0]
        rememberSourceNodes.remove(nextSourceConnection)
        rememberTargetNodes.remove(nextTargetConnection)
    } else {
        nextTargetConnection = newNode
    }

    List<Connection> rSourceNodes = Connection.remember(sourceConnections, sourceNode)
    if (rSourceNodes.size() > 0) {
        for (rSourceNode in rSourceNodes) {
            rememberSourceNodes << rSourceNode
            rememberTargetNodes << targetNode
        }
    }
} else if (sourceNode.nTargetOutArcs == 0 && targetNode.nTargetOutArcs > 0){
    log.trace("the first is an end node, the second no")
```

```
        //delete node
        delDiff++

        nextSourceConnection = sourceNode
        (nextTargetConnection, targetBacktrack) = Connection.findNext(visitedTargetConnections, targetConnections, targetNode, rememberTar

        //println "NODE " +targetNode + nextTargetConnection
        //DELETE ALL INSTANCES TILL 0 FROM targetConnections

        if (targetBacktrack) {
            log.trace("backtrack!")
            nextSourceConnection = rememberSourceNodes[0]
            rememberSourceNodes.remove(nextSourceConnection)
            rememberTargetNodes.remove(nextTargetConnection)
        } else {
            nextSourceConnection = sourceNode
        }

        List<Connection> rTargetNodes = Connection.remember(targetConnections, targetNode)
        if (rTargetNodes.size() > 0) {
            for (rSourceNode in rTargetNodes) {
                rememberTargetNodes << rSourceNode
                rememberSourceNodes << sourceNode
            }
        }

    //if both are end nodes, find the next remembered node or start from beginning
    } else if (sourceNode.nTargetOutArcs == 0 && targetNode.nTargetOutArcs <= 0) {
        log.trace("they are both end nodes")

        (nextSourceConnection, sourceBacktrack) = Connection.findNext(visitedSourceConnections, sourceConnections, sourceNode, rememberSou
        (nextTargetConnection, targetBacktrack) = Connection.findNext(visitedTargetConnections, targetConnections, targetNode, rememberTar

        //println "Next line " + nextConnection + nextTargetConnection
        if (nextSourceConnection == null && !rememberSourceNodes.isEmpty()) {
            nextSourceConnection = rememberSourceNodes[0]
            nextTargetConnection = rememberTargetNodes[0]
        }

        //REMEMBER IS [] THEN GO FROM THE END BACK TO BEGINNING - NEEDS TO BE IMPLEMENTED
    } else if (targetNode.nTargetOutArcs == -1) {
        log.trace("The new node needs more additions")
        Connection newNode = new Connection(
                sourceId: sourceNode.sourceId,
                targetId: "n"+addDiff,
                nSourceInArcs: 1,
                nSourceOutArcs: 1,
                nTargetOutArcs: -1
        )

        newConnections.add(0,newNode)
        nextTargetConnection = newConnections[0]
        addDiff++
        (nextSourceConnection, sourceBacktrack) = Connection.findNext(visitedSourceConnections, sourceConnections, sourceNode, rememberSou

        if (sourceBacktrack) {
            log.trace("backtrack!")
            nextTargetConnection = rememberTargetNodes[0]
            rememberSourceNodes.remove(nextSourceConnection)
            rememberTargetNodes.remove(nextTargetConnection)
        }else if(targetBacktrack){
            log.trace("backtrack!")
            nextSourceConnection = rememberSourceNodes[0]
            rememberSourceNodes.remove(nextSourceConnection)
            rememberTargetNodes.remove(nextTargetConnection)
        }
        List<Connection> rSourceNodes = Connection.remember(sourceConnections, sourceNode)

        if (rSourceNodes.size() > 0) {
            for (rSourceNode in rSourceNodes){
                rememberSourceNodes << rSourceNode
                rememberTargetNodes << newNode
                sourceConnections.remove(rSourceNode)
            }
        }
    }

    sourceConnections.remove(sourceNode)
    targetConnections.remove(targetNode)

    Integer removeIndex = rememberSourceNodes.indexOf(sourceNode)
    rememberSourceNodes.remove(sourceNode)
    if (removeIndex >= 0) {
        rememberTargetNodes.remove(removeIndex)
    }
    sourceNode = nextSourceConnection
    targetNode = nextTargetConnection

    //println "Rem " + rememberNode + rememberNode1
    log.trace "sourceConnections: " + sourceConnections
    log.trace "targetConnections: " + targetConnections
    log.trace "visitedSourceConnections: " + visitedSourceConnections
    log.trace "visitedTargetConnections: " + visitedTargetConnections
    log.trace "rememberSourceNodes: " + rememberSourceNodes
```

```
            log.trace "rememberTargetNodes: " + rememberTargetNodes
            log.trace "delDiff: " + delDiff
            log.trace "addDiff: " + addDiff

        }

        Integer conSize = targetConnections.size()
        //println conSize
        delDiff = delDiff + conSize
        //println "End Total del " + delDiff

        return [addDiff, delDiff]
    }

}
```

B.2 connection.groovy

```
package org.leibnizcenter.pneu.comparison

import groovy.util.logging.Log4j
import org.leibnizcenter.pneu.components.petrinet.Arc
import org.leibnizcenter.pneu.components.petrinet.Net

@Log4j
class Connection {
    String sourceId
    String targetId
    Integer nSourceInArcs = 0
    Integer nSourceOutArcs = 0
    Integer nTargetInArcs = 0
    Integer nTargetOutArcs = 0

    static List<Connection> readConnections(Net net) {
        List<Arc> arcs = net.arcList
        List<Connection> connections = []
        for (arc in arcs) {
            // a list with all connections but not in execution order.
            connections << new Connection(sourceId: arc.source.id, targetId: arc.target.id)
        }

        for (i in connections) {
            for (j in connections) {
                // println i.toString() + " with " + j.toString()

                if (i.sourceId == j.targetId) {
                    i.nSourceInArcs++
                }
                if (i.sourceId == j.sourceId) {
                    i.nSourceOutArcs++
                }
                if (i.targetId == j.sourceId) {
                    i.nTargetOutArcs++
                }
                if (i.targetId == j.targetId) {
                    i.nTargetInArcs++
                }
            }
        }
        connections
    }

    static List<Connection> searchBySourceId(List<Connection> connections, String id) {
        log.trace("searchBySourceId ### connections: $connections ### id $id")
        List<Connection> result = []
        for (connection in connections) {
            if(connection!=null) {
                if (connection.sourceId == id) {
                    result << connection
                }
            }
        }
        return result
    }

    // search for nodes and delete or add an output arc on the source
    // useful for visiting nodes twice
    static List<Connection> searchBySourceIdAdd(List<Connection> connections, String id, Integer adding) {
        List<Connection> result = []
        for (connection in connections) {
            if (connection.sourceId == id) {
                connection.nSourceOutArcs = connection.nSourceOutArcs + adding
                result << connection
            }
        }
        return result
    }

    static List<Connection> searchByTargetId(List<Connection> connections, String id) {
        List<Connection> result = []
        for (connection in connections) {
            if (connection.targetId == id) {
                result << connection
```

```
            }
        }
        return result
    }

    // TOCHECK: is this correct? there may be places with no inputs/outputs
    static Connection searchCollector(List<Connection> connections) {
        for (connection in connections) {
            if (connection.nSourceInArcs == 0) {
                return connection
            }
        }
    }

    // remembering another path
    static List<Connection> remember(connections, node) {
        if (node.nSourceOutArcs > 1 && node.nSourceInArcs != 0) {
            List<Connection> rememberSourceNodes = searchBySourceIdAdd(connections, node.sourceId, -1)
            rememberSourceNodes.remove(node)
            return rememberSourceNodes
        } else {
            return []
        }
    }

    // finding next connection
    static def findNext(List<Connection> visitedConnections, List<Connection> connections, Connection node, List<Connection> remember) {
        log.trace("findNext. visited: $visitedConnections, connections: $connections, node: $node, remember: $remember")

        Connection next
        log.trace("Next: $next")
        Boolean backtrack = false

        for (connection in connections) {
            if (visitedConnections.contains(node) && connection.nSourceInArcs == 0) {
                next = connection
            } else {
                if (connection.sourceId == node.targetId && connection.targetId != node.sourceId && node.nTargetOutArcs != 0) {
                    next = connection
                } else if (node.nTargetOutArcs == 0 && connection.nSourceInArcs == 0) {
                    next = connection
                }
            }
        }

        if (!next) {

            // if no new connection found, look in visitedConnection
            // if remembered node is used, other node needs to be remembered as well
            List<Connection> nextN = searchBySourceId(visitedConnections, node.targetId)

            if (nextN.size() > 0) {
                if (!remember.isEmpty()) {
                    next = remember[0]
                    backtrack = true
                    log.trace("HI")
                } else {
                    for (connection in connections) {
                        if (connection.nSourceInArcs == 0) {
                            next = connection
                            backtrack = true
                        }
                    }
                }
            }

        }

        // TO CHECK: there is some NULL around
        if (node)
          visitedConnections << node

        return [next, backtrack]
    }

    String toString() {
        return sourceId + "{$nSourceInArcs,$nSourceOutArcs}->" + targetId + "{$nTargetInArcs,$nTargetOutArcs}"

    }
}
```