Case study in Artificial Intelligence

1 Students

Name Surname, serial number, email
Federico Canistro, 775605, f.canistro@studenti.uniba.it
Ivan De Cosmis, 787066, i.decosmis@studenti.uniba.it

2 Introduction

In this report, we present an Artificial Intelligence project. In the first section, we introduce a script that reads a CSV file containing event data and converts it into an event log. It then applies three different process discovery algorithms (Heuristics Miner, Alpha Miner, and Inductive Miner) to the event log to discover Petri net models and subsequently compare some properties. The discovered Petri nets are then exported in PNML format and saved as images.

In the second section, we present a Prolog program that reads a series of facts defining a Petri net and converts the read information into a PNML file, which is one of the most widely used standards for Petri net representation.

In the third section, we present a Prolog program that reads a "wf" file containing information about the transitions analyzed by the system and outputs a series of files useful for visualizing the different transitions.

3 Comparison with Previous Experiments

In the previous case study, the "timestamp" column in the CSV file was used as the case identifier during the conversion of the data frame into an event log. This means that each row in the CSV file was treated as a separate case with a single event. As a result, the duration of each case was always 0.0 because there was only one event in each case.

To correctly calculate the case duration, it was necessary to specify a column in the CSV file that uniquely identifies each case when calling the *pm4py.format_dataframe* function.

In the new version of the code, we are using the "id_exec" column of the CSV file as the case identifier when converting the data frame into an event log. This means that rows in the CSV file that have the same value for the "id_exec" column will be grouped into the same trace within the event log.

The difference from the previous event log is that now traces in the event log can contain more than one event. This allows for calculating statistics such as the case duration, which measures the time elapsed between the first and last event in each trace.

Each output of the used algorithms is saved in a PNML file and also in PNG format to observe its graphical visualization.

3.1 Secondary Functions

get_all_duration_case: Checks the duration of all cases.

compare_pnml_files: Compares the number of 'places' and 'transitions' in the files.

3.2 Heuristics Miner

The Heuristics Miner is a more advanced process discovery algorithm that can better handle common characteristics of event logs. As a result, the Heuristics Miner was able to produce a more accurate Petri net model compared to the other algorithms for the same event log.

3.3 Alpha Miner

The Alpha Miner algorithm is a process discovery algorithm that can be used to discover a Petri net model from an event log. However, the Alpha Miner has some limitations and was not able to properly handle certain common characteristics of event logs, such as parallel activities and long-term dependencies between activities. In this case, the Alpha Miner produced a Petri net model that does not accurately represent the underlying process.

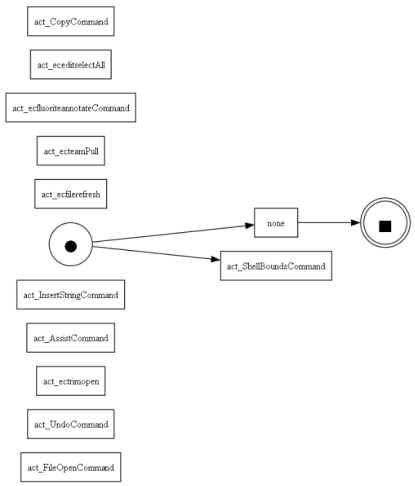

Fig.1. Part of alpha output on the complete dataset.

3.4 Inductive Miner

The Inductive Miner is a process discovery algorithm that can be used to discover a Petri net model from an event log. In general, the Inductive Miner is able to handle large and complex event logs efficiently. However, in some cases, the Inductive Miner may take longer to discover a Petri net model, depending on the characteristics of the event log and the parameters used.

In our case, the Inductive Miner was running for over an hour on the complete dataset and still did not finish. However, it responded very well when tested on separate datasets.

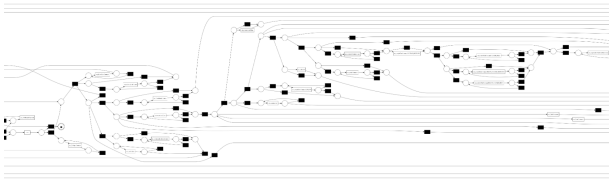

Fig.1. parte di output inductive corretto su uno dei dataset.

We have also tried to use variants of the Inductive miner algorithm, but always without success (always on the full dataset). Furthermore, we also tried to modify the parameters of the algorithm to improve them performance. We used the noise threshold parameter to specify a noise threshold for the algorithm. Increasing the noise threshold can help the algorithm better handle noisy event logs and improve performance performance, but in our case this led to incorrect output as in the case of the alpha miner.

4 Petri Net Conversion from Woman Format to PNML Format

The idea of this algorithm is to read from an input file in the format:

1. place(X).
2. transition(X).
3. arc(X, Y, T).

The information is stored in three independent lists (conversion_to_pnml.pl, 1):

```
% Declare the lists as dynamic to allow modification at runtime
:- dynamic string_list1/1.
:- dynamic string_list2/1.
:- dynamic string_list3/1.
```

Next, predicates are created to access the lists (conversion_to_pnml.pl, 6):

```
% Add string to the input list (1, 2, or 3)
add_string(ListNumber, String) :-
    (   retract(string_list(ListNumber, List))
    ->  assertz(string_list(ListNumber, [String|List]))
    ;   assertz(string_list(ListNumber, [String]))
    ).

% Return the list based on the input number (1, 2, or 3)
get_strings(ListNumber, Strings) :-
    (   string_list(ListNumber, Strings)
    ->  true
    ;   Strings = []
    ).
```

The **add_string** predicate adds the input element to the list with the given index. The **get_strings** predicate returns the list with the given index.

4.0.1 Counter

During the creation of the file **pnml** it is necessary to assign id to places, transitions and arcs. To do this we use a counter that is incremented each time to obtain a new id. Code to manage the counter(conversion_to_pnml.pl, 20):

```
% Inizializzo contatore usato nel salvataggio dei dati nel file pnml
:- dynamic(counter/1).
counter(0).

% Inizializzo contatore per gli id dei place, delle transition e degli arc nel file pnml
increment_counter :-
    retract(counter(Count)),
    CountPlusOne is Count + 1,
    assert(counter(CountPlusOne)).
```

4.1 Reading Data from File

4.1.1 process_file(FileName)

The procedure that reads the file and saves the places, transitions, and arcs into the lists is called **process_file(FileName)**. It takes the file path as input (conversion_to_petri.pl, 30).

```
% Process the input file and export the new pnml file
process_file(FileName) :-
    retractall(string_list(1, List)),
    retractall(string_list(2, List)),
    retractall(string_list(3, List)),
    consult(FileName),
    process_net,        % Start analyzing the net
    get_strings(1, Places),
    get_strings(2, Transitions),
    get_strings(3, Arcs),
    ask_folder_path(Path),
    write_pnml(Path, Places, Transitions, Arcs).
```

"retractall" is used to reset all the lists every time the file is consulted. "process_net" is the procedure that processes the net.

4.1.2 process_net

This procedure contains the operations performed to extract data from the facts read in the input:

1. extract_places(NewPlaces): extracts all the places from the database and saves them in the NewPlaces list.
2. copy_list(1, NewPlaces): copies the input list to dynamic list 1.
3. extract_transitions(NewTransitions): extracts all the transitions from the database and saves them in the NewTransitions list.
4. copy_list(2, NewTransitions): copies the input list to dynamic list 2.
5. extract_arcs(NewArcs): extracts all the arcs from the database and saves them in the NewArcs list.
6. copy_list(3, NewArcs): copies the input list to dynamic list 3.

(conversion_to_pnml.pl, 69)

```
% Extract information from the facts
process_net :-
    extract_places(NewPlaces),
    copy_list(1, NewPlaces),
    extract_transitions(NewTransitions),
    copy_list(2, NewTransitions),
    extract_arcs(NewArcs),
    copy_list(3, NewArcs).
```

4.1.3 extract_places(Places)

Let's analyze this procedure, which is very similar to **extract_transitions(Transitions)** and **extract_arcs(Arcs)**, with the only difference being the syntax based on the type of data to extract (conversion_to_pnml.pl, 49):

```
% Extract all places into a list
extract_places(Places) :-
    findall(Place, place(Place), Places).
```

The `findall` predicate searches for all facts in the format `place(Place)` and saves them in the list `Places`.

4.1.4 copy_list

This predicate has two cases (conversion_to_pnml.pl, 61):

1. `copy_list(IdList, [])`: Case when the input list is empty, indicating the completion of the copy process.
2. `copy_list(IdList, [H|T])`: Case when the list still needs to be copied.

```
% Predicate that allows us to copy a list to one of the three
%dynamic lists based on the input index
copy_list(IdList, []) :-
    true.

copy_list(IdList, [H | T]) :-
    add_string(IdList, H),
    copy_list(IdList, T).
```

With this function, we have finished analyzing the part related to reading data from the input file.

4.2 Writing the pnml File

To write to the pnml file, we use the **write_pnml** procedure.

4.2.1 write_pnml

This function opens the file specified by the path entered in the console and starts writing the various parts of the pnml file (conversion_to_petri.pl, 78).

```
% Write the pnml file with place, transition, and arc data
write_pnml(File, Places, Transitions, Arcs) :-
    open(File, write, StreamWrite),
    write_header(StreamWrite),
    write_places(StreamWrite, Places),
    write_transitions(StreamWrite, Transitions),
    write_arcs(StreamWrite, Arcs),
    write_footer(StreamWrite),
    close(StreamWrite).
```

4.2.2 ask_folder_path(String)

This procedure prompts the user to enter the path where the output file should be saved. The entered path is concatenated with the string containing the output file name using the **atom_concat** predicate (conversion_to_petri.pl, 43).

```
ask_folder_path(String) :-
    write('Enter the path where you want to save the output file (e.g., C:/User/Desktop): '),
    read_line_to_string(user_input, Read),
    atom_concat(Read, "/petri_net.pnml", String).
```

4.2.3 write_header(StreamWrite)

This function writes the header of the PNML file (conversion_to_petri.pl, 88).

```
% Scrivo intestazione file
write_header(StreamWrite) :-
    write(StreamWrite, '<?xml version="1.0" encoding="UTF-8"?>'), nl(StreamWrite),
```

```
write(StreamWrite, '<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">'), nl(StreamWrite),
write(StreamWrite, '  <net id="net1" type="http://www.pnml.org/version-2009/grammar/ptnet">'), nl(StreamWrite),
write(StreamWrite, '    <name>'), nl(StreamWrite),
write(StreamWrite, '        <text> Petri Net </text>'), nl(StreamWrite),
write(StreamWrite, '    </name>'), nl(StreamWrite),
write(StreamWrite, '    <page id="page1">'), nl(StreamWrite).
```

4.2.4 write_places, write_transitions, write_arcs

Given that they are all very similar, the only difference being the formatting to comply with the PNML format, let's analyze the **write_places** function.

This procedure has two cases:

1. Base case (conversion_to_petri.pl, 99):

```
% Questo predicato serve quando abbiamo finito tutti i place
write_places(StreamWrite, []) :-
    true.
```

This is the case when there are no more places to write to the file.

2. Case with a remaining list to empty (conversion_to_petri.pl, 103):

```
% Questo predicato serve quando ci sono ancora place da scrivere
write_places(StreamWrite, [Place|Places]) :-
    write(StreamWrite, '        <place id="'),
    counter(Count),
    write(StreamWrite, Count), write(StreamWrite, '">'), nl(StreamWrite),
    increment_counter,
    write(StreamWrite, '            <name>'),
    nl(StreamWrite),
    write(StreamWrite, '                <text>'),
    write(StreamWrite, Place), write(StreamWrite, '</text>'), nl(StreamWrite),
    write(StreamWrite, '            </name>'), nl(StreamWrite),
    write(StreamWrite, '        </place>'), nl(StreamWrite),
    write_places(StreamWrite, Places).
```

In this case there are places to write on file and thanks to the write function we are going to write the place respecting the syntax of the pnml format. "increment_counter" increments the counter representing the id of the place, transition and arc. Finally, the function calls itself by iterating through the list without the first element.

4.2.5 write_footer

This predicate writes the footer of the pnml file (conversion_to_petri.pl, 159)

```
% Questo predicato serve quando dobbiamo scrivere il pie di pagina pnml
write_footer(StreamWrite) :-
    write(StreamWrite, '    </page>'), nl(StreamWrite),
    write(StreamWrite, '  </net>'), nl(StreamWrite),
    write(StreamWrite, '</pnml>'), nl(StreamWrite).
```

With this procedure we have finished describing the code of this section.

4.2.6 Description of the test case

This program is loaded into SWI-Prolog using the command:
consult('Path_to_to_file/conversion_to_petri.pl').
Then the procedure is called:
process_file('Path_to_to_file/petri_net.pl').
When asked for the path where to save the file enter:
Path_to_to_where_save_the_file/name_file.pnml

The program will return true and save the resulting file in the indicated path. In our case in the github repository the file é:
conversione_petriWoman_pnml/petri_net.pnml

5 Viewing transitions from a wf file

In this section we are going to write the possible successive transitions inside a text file, based on the input action and the correspondence between cases. Furthermore, two dot files are generated, having different properties, which are used to graphically generate the graph representing the continuity between transitions. Furthermore, possible transitions are also indicated, which even if they have never been observed by the cases of the system, could still take place.

5.0.1 Module management

In order to use the loaded transitions we need to understand the module in which they belong. To avoid hard-coding the module name used for testing (monday) and thus forcing syntax like:

monday:transition(Input-Output,Cases,Repetition)

We have used a dynamic variable in which we save the atom of the module read from the key-board(model_explorator.pl, 1).

```
% Soluzione per poter rendere meno vincolante il codice al modulo
% inserito.
% In questo modo basta inserire il nome del modulo da cui sono state
% lette le transition per poter visualizzare nel file i dati
% senza la necessita di fare hardcode col nome del modello
:- dynamic module/1.

% Predicato che imposta il modulo
set_module(Atom) :-
    retractall(module(_)),
    assertz(module(Atom)).

% Predicato che legge il modulo
get_module(Atom) :-
    module(Atom).
```

The module is requested by the procedure **ask_module_name(String)** (model_explorator.pl, 164):

```
% Chiedo il nome del modulo usato
ask_module_name(String) :-
    write('Enter the module name used: '),
    read_line_to_string(user_input, String).
```

5.0.2 Transition management in analysis

During the execution of the code it is necessary to recover the information of the transition under analysis while printing the transitions on the different files, for this we have used another dynamic variable capable of memorizing the transition under analysis (model_explorator.pl, 27)

```
% Dynamic variable that is used to save the transition in analysis and to be able to
% retrieve its information from anywhere in the code
:- dynamic current_transition/1.

% Predicate to set the transition in analysis
set_current_transition(Transition) :-
    retractall(current_transition(_)),
    assertz(current_transition(Transition)).
```

```
% Predicate to recover the transition in analysis
get_current_transition(Transition) :-
    current_transition(Transition).
```

5.0.3 Counter

While analyzing the **aruba.wf** file we found in transition/2 unlike the **monday.wf** file that they were transition/3. For this reason we have added a new predicate that converts the fact transition/2 to transition/3 by arbitrarily adding the unique value for the iterations to each transition. Code to manage the counter(model_explorator.pl, 17):

```
% Counter initialization used when saving data to pnml file
:- dynamic(counter/1).
counter(0).

% Initialize counter for place, transition and arc ids in pnml file
increment_counter :-
    retract(counter(Count)),
    CountPlusOne is Count + 1,
    assert(counter(CountPlusOne)).
```

5.0.4 transition_check

This predicate is used to convert transition/2 to transition/3 by adding a number, which increases with each iteration, as the third parameter (model_explorator.pl, 247):

```
% This predicate is used in case we find facts
% transition/2, which are converted to transition/3 by adding a
% unique value as third element
transition_check :-
    get_module(Module),
    clause(Module:Module:transition(FirstInput-FirstOutput, FirstList), true),
    counter(Count),
    increment_counter,
    assert(Module:Module:transition(FirstInput-FirstOutput, FirstList,Count)),
    fail.

transition_check.
```

5.0.5 Start of operations

The procedure from which all operations start is called **process_file(FileName)**(model_explorator.pl, 40)

```
% Reads the wf file and processes it
process_file(FileName) :-
    ask_folder_path(Path),
    atom_concat(Path,"/result.txt",ResultPath),
    use_module(FileName),
    ask_module_name(ModuleString),
    string_to_atom(ModuleString, Module),
    set_module(Module),
    transition_check,
    open(ResultPath, write, Stream),
    do_works(Stream, Path).
```

Let us now analyze its operations:

1. ask_folder_path(Path): asks from the terminal to enter the path in which to save the output file (procedure practically identical to that of reading the form)
2. atom_concat(Path,"/result.txt",ResultPath): it is used to add the result file name to the input path
3. use_module(FileName): load the input path module
4. ask_module_name(ModuleString): asks the name of the module to use from the terminal
5. string_to_atom(ModuleString, Module): converts the module string to Atom to be used during operations
6. set_module(Module): imports the dynamic variable with the atom just obtained
7. transition_check: it is used to verify if the transitions read from the input module are correct, if they are not they are converted into the accepted format
8. open(Path, write, Stream): create the file in the specified path and create the Stream to use
9. do_works(Stream, Path): starts the operations of searching and analyzing the transitions

5.0.6 do_works

This procedure takes care of finding all the transitions and inserting them in a list to be able to analyze them later by iterating on it, it takes advantage of the **get_module**(model_explorator, 52) procedure.

```
% Start main execution
do_works(Stream, Path) :-
    get_module(Module),
    findall(Module:transition(Input-Output, Cases, Repetition),
            Module:transition(Input-Output, Cases, Repetition),
            ListTransitions), % Gather all transitions into a list
    do_work(Module, Stream, ListTransitions),
    generate_dot_files(Path).
```

Call the procedure **do_work(Module, Stream, ListTransitions)** which will start the loop to check all the transitions. It then generates the dot files for the graphs.

5.0.7 do_work

The reason we give Module as input is because we need it to specify the module in the input transition. This procedure takes care of writing the introduction to the transition under analysis and then recalls the procedures for searching for the transitions to be displayed. It also iterates on itself to analyze all the input transitions(model_explorator.pl, 61).

```
% Perform file write and transition information retrieval operations
do_work(Module, Stream, [Module:transition(Input-Output, Cases, Repetition) | Rest]) :-
    set_current_transition(Module:transition(Input-Output, Cases, Repetition)),
    write(Stream, '-----------------------------------------------'), nl(Stream),
    write(Stream, 'Starting transition:'), nl(Stream),
    write(Stream, Input-Output),
    write(Stream, ','),
```

```
    write(Stream, Cases),
    write(Stream, ','),
    write(Stream, Repetition),nl(Stream),
    writeln(Stream, 'You can continue to the following ones:'),
    get_matching_transitions_with_case(Stream, Output-_, Cases, Transitions),
    do_work(Module, Stream, Rest).
```

5.0.8 get_matching_transitions_with_case

In this procedure, the operation of selecting the transitions following the one under analysis takes place. **findall** is used to find transitions that:

1. Have the input you are looking for
2. One of the cases of the transition under analysis is found in the list of cases

The first point is obtained using the predicate **contains_element** which returns **true** if there is an element in common between the two input lists(model_explorator.pl, 181).

```
% Predicate that allows us to check if
% there are elements in common between two lists
contains_element([], _) :-
    false.
contains_element([H|B], List) :-
    member(H, List),
    !.
contains_element([H|B], List) :-
    \+ member(H, List),
    contains_element(B, List).
```

The second point is achieved using **member**. All found transitions are saved in the **NewTransitions** list. Now, however, we are not finished yet because we have found all the transitions that match only one case of the list (the first one), we have to analyze the other cases. To do this we are going to repeat the same procedure. To avoid getting duplicates in the list (transitions that share more than one case) we use the procedure **add_to_list_no_duplicates** which reads the list of new transitions as input, compares them with those already found (there will be none in the first iteration), and all new ones are added to the list, creating the new one that will be sent to the next iteration. All already found transitions will be discarded(model_explorator.pl, 67).

```
% Retrieves all transitions that have at least one completed action in the input list
% from the output list (using the contains_element predicate), identifying them
% as possible subsequent transitions
% At the end all the transitions that respect the search parameters are saved in the
% NewTransitions list
% All transitions that have not already been found (and therefore are not in the list
% Transitions), they come
% added to the final list which is called LastTransitions which will eventually be printed
get_matching_transitions_with_case(Stream, Input-Output, [Case|Cases], Transitions) :-
    get_module(Module),
    findall(Module:transition(ListInput-Output, TransitionsList, Repetition),
            (Module:transition(ListInput-Output, TransitionsList, Repetition),
             contains_element(Input,ListInput),
             member(Case, TransitionsList)),
            NewTransitions),
    add_to_list_no_duplicates(NewTransitions, Transitions, LastTransitions),
```

```
    get_matching_transitions_with_case(Stream, Input-Output, Cases, LastTransitions).


% Base case that ends identifying all the transitions that are
% theoretically possible but
% that have never been observed by the system and prints all
% transitions found
get_matching_transitions_with_case(Stream, Input-Output, [], Transitions) :-
    get_matching_transitions(Input-Output, ExtraTransitions),
    print_safe_list(Stream, Transitions),
    add_to_list_deleting_duplicates(ExtraTransitions, Transitions, NewExtraTransitions),
    writeln(Stream,'Possible transitions that could happen even if the cases do not match:'),
    print_possible_list(Stream, NewExtraTransitions).
```

The second case represents the situation where we have finished analyzing all the transitions concerning the current one, so we print the result. In this procedure we are also going to call **get_matching_transitions** which allows us to get all the transitions that share the same input regardless of the cases, we exploit this procedure to also propose transitions that could happen but that have not been covered by the system cases. To avoid duplicates, after having found all the transitions that share the input with the one under analysis we are going to remove those already found with the procedure **add_to_list_deleting_duplicates** which gives two lists in input returns a list containing the elements of the first without the elements of the second (thereby eliminating the valid transitions found previously). Finally print them.

5.0.9 add_to_list_no_duplicates

This procedure receives two lists as input and returns a third list containing the elements of the first list and the second list without duplicates (model_explorator.pl, 114).

```
% This predicate is used to copy all to the second list
% the elements of the first list that are not present in his
% internal (Used to find cases with match without
% keep duplicates)
add_to_list_no_duplicates([], List, List).

add_to_list_no_duplicates([H|B], SecondList, Output) :-
    member(H, SecondList),
    !,
    add_to_list_no_duplicates(B, SecondList, Output).

add_to_list_no_duplicates([H|B], SecondList, Output) :-
    add_to_list_no_duplicates(B, [H|SecondList], Output).
```

This procedure has 3 cases:

1. the first is the base case where the output list is merged with the second
2. the second case is where we check if the item at the head of the first list exists in the second, if it exists we simply skip it and repeat
3. the third case ís the case where the element did not belong to the list, made the second case fail and so in this one we are going to add it to the head of the second list and then iterate.

5.0.10 get_matching_transitions

This procedure allows us to find transitions that share the same input, it is used to find transitions that could occur but that have not been analyzed by the system. It uses the same mechanism as the **get_matching_transitions_with_case** procedure, but without checking the list of cases(model_explorator.pl, 104).

```
% I look for transitions only based on the input action
% It suffices that at least one is present to be identified as a possible condition
% next
get_matching_transitions(Input-Output, Transitions) :-
    get_module(Module),
    findall(Module:transition(ListInput-Output, TransitionsList, Repetition),
            (Module:transition(ListInput-Output, TransitionsList, Repetition),
            contains_element(Input,ListInput)),
            Transitions).
```

5.0.11 add_to_list_deleting_duplicates

This procedure is a modified version of **add_to_list_no_duplicates**, because instead of adding new elements to an existing list, it adds only the elements that do not exist in the second list (in our case the transitions that do not share cases with the analyzed one, allowing us to identify those that are possible but not analyzed by the system) (model_explorator.pl, 117).

```
% This predicate is used to create a new list that
% contains the elements of the first list minus the elements
% of second (Used to see possible extra cases without
% correspondence)
add_to_list_deleting_duplicates(List, SecondList, Output) :-
    add_to_list_deleting_duplicates(List, SecondList, [], Output).

add_to_list_deleting_duplicates([], _, Temp, Temp).


add_to_list_deleting_duplicates([H|B], SecondList, Temp, Output) :-
    member(H, SecondList),
    !,
    add_to_list_deleting_duplicates(B, SecondList, Temp, Output).

add_to_list_deleting_duplicates([H|B], SecondList, Temp, Output) :-
    add_to_list_deleting_duplicates(B, SecondList, [H|Temp], Output).
```

We can find 4 cases:

1. The first case is the one that receives the first list and the second list as input and calls the procedure with a new parameter useful to finally unify the new list

2. The second case is the base case, i.e. when we have run out of elements to compare and therefore the new list is merged with the output one

3. The third case contains the case with the check that checks whether or not the element in the second list exists, if it exists it goes on and iterates again, otherwise it goes to the fourth case

4. The last case adds the found item to the new list and iterates

5.0.12 print_safe_list

With this predicate we print an input list in the received Stream(model_explorator.pl, 147).

```
% Predicate that prints the input list and divides with dashes
% formatting, does not indicate any particular properties on arcs
print_safe_list(Stream, []) :-
    write(Stream, '----------'), nl(Stream),
    flush_output(Stream).
print_safe_list(Stream, [H|B]) :-
    get_current_transition(Transition),
    assert(edge(Transition, H,"")),
    writeln(Stream, H),
    print_safe_list(Stream, B).
```

We can find 2 cases:

1. The first case is the one that considers the list empty and therefore the end of printing which is indicated with horizontal dashes, then the buffer is also emptied otherwise you risk filling it up and not being able to print to file anymore
2. The second case is the case with the list still having elements to print, the head of the list is printed and then the cycle is repeated.

In addition to printing on the text file we are going to create the edges which are useful for creating the dot file later. The predicate includes **safe** in the name because it is used to write only transitions considered by system cases.

5.0.13 print_possible_list

This predicate is very similar to the previous one, it is used to write the possible transitions but which have not been observed by the system cases, for this reason it adds "style="dashed"," in the edge properties, which will allow us 'a later to make these arcs dashed to be able to distinguish them easily (model_explorator.pl, 159).

```
% This predicate prints the input list and divides with dashes
% formatting.
% In particular, this predicate also prints the property indicating the style of the strings
print_possible_list(Stream, []) :-
    write(Stream, '----------'), nl(Stream),
    flush_output(Stream).
print_possible_list(Stream, [H|B]) :-
    get_current_transition(Transition),
    assert(edge(Transition, H,'style="dashed",')),
    writeln(Stream, H),
    print_possible_list(Stream, B).
```

5.0.14 generate_dot_files

This predicate allows us to generate dot files that contain the information needed to create a graph. Two files are generated from the path received as input, the first will have the property **ordering=out**, this property renders the nodes and arcs based on the reading order. The second will have the property **rankdir=LR**, this property tries to generate the graph concentrating the nodes from left to right. We use **atom_concat** to be able to add the names of the output files to the path received as input.

```
% This predicate generates a dot file containing all the description of the graph.
% Will be written with the ordering=out property for displaying the graph
generate_dot_files(Path) :-
    atom_concat(Path, "/orderingout.dot", OrderingPath),
    open(OrderingPath, write, OrderingStream),
    write(OrderingStream, "digraph G {\n"),
    write(OrderingStream, "ordering=out;\n"),
    write_edge(OrderingStream),
    write(OrderingStream, "}\n"),
    close(OrderingStream),
    atom_concat(Path, "/rankdirlr.dot", RankdirlrPath),
    open(RankdirlrPath, write, RankdirlrStream),
    write(RankdirlrStream, "digraph G {\n"),
    write(RankdirlrStream, "rankdir=LR;\n"),
    write_edge(RankdirlrStream),
    write(RankdirlrStream, "}\n"),
    close(RankdirlrStream).
```

5.0.15 write_edge

This predicate allows us to write edges into dot files. The dot files can then be processed to obtain a graphical representation of the graph. We analyze each edge until the procedure examines all the edges present in memory and continues in the second procedure terminating successfully. The **get_transition** predicate is used which unifies the input transition to the output one allowing us to obtain the information inside it. The **number_of_items_in_common** predicate is used to check how many cases the start and finish transitions have in common, this value will indicate the weight of the arc. Finally we write all the relevant information of the transitions in the file(model_explorator.pl, 212).

```
% This predicate is used to print all the edges found in the dot file
write_edge(Stream) :-
    edge(From, To, Properties),
    get_transition(From, Module:transition(FirstInput-FirstOutput,
                                           FirstList, FirstRepetition)),
    get_transition(To, Module:transition(SecondInput-SecondOutput,
                                         SecondList, SecondRepetition)),
    length(FirstList, FirstWeight),
    length(SecondList, SecondWeight),
    number_of_items_in_common(FirstList, SecondList, Count),
    format(Stream, '"~w Weight:~w"->"~w Weight:~w"[~wlabel="~w"];\n',
           [Module:transition(FirstInput-FirstOutput, FirstList, FirstRepetition),
           FirstWeight,
           Module:transition(SecondInput-SecondOutput, SecondList, SecondRepetition),
           SecondWeight,
           properties,
           Count]),
    fail.
write_edge(_).

% This predicate serves to unify the input transition to the output one
get_transition(Transition, Transition).


% This predicate is used to find the number of elements in common between two lists
number_of_items_in_common([], [], 0).

number_of_items_in_common([], _, 0).

number_of_items_in_common(_, [], 0).

number_of_items_in_common([H|B], List, Count) :-
    member(H, List),
    number_of_items_in_common(B, List, PrevCount),
    Count is PrevCount + 1.

number_of_items_in_common([H|B], List, Count) :-
    \+ member(H, List),
    number_of_items_in_common(B, List, Count).
```

The weight of the nodes instead is indicated by the length of the list of cases, ie for how many cases the transition has been repeated in the system.

With this procedure we conclude the analysis of the code of this section.

5.0.16 Description of test case with monday.wf

Let's analyze the case with the **monday.wf** file This program is loaded into SWI-Prolog using the command:

consult('Path_to_to_file/model_explorator.pl').

Then the procedure is called:

process_file('Path_to_to_file/monday.wf').

When asked for the path where to save the file enter:

Path_to_to_where_save_the_output_files

When asked for the name of the module, enter the name of the loaded module:
monday

The program will return true and save the files in the indicated path. In our case in the github repository the file é:
visualizzazione_modello/output_monday

The same procedure is repeated with the aruba file, changing the name of the input module from **monday** to **aruba** and changing the path of the output files so as not to delete the previously generated ones. In our case in the github repository the file é:
visualizzazione_modello/output_aruba

The created dot files can be converted into a graphical representation thanks to Graphviz:

1. Make sure you have installed the Graphviz libraries from the site: `https://graphviz.org/download/`
2. Open the terminal at the path of the file to convert
3. Use the following command to generate the graph in pdf format: dot -Tpdf -o name_file_output.pdf name_file_input.dot

In our case the command to generate a graph file starting from the dot file was:

```
dot -Tpdf -o graph_orderingout.pdf orderingout.dot
```

All pdf graphs based on the input dot files have been generated and are present in their respective output folders. Depending on the complexity of the grade, the time required for its rendering could vary from a few seconds to several minutes.
In the case of **monday** the graphs were rendered in about 1-2 minutes;
In the case of **aruba** both renders took about 10 minutes, using a computer with the following specifications:

1. Processor: i7-8700k 3.70 GHz
2. Ram: 32GB, 3000MHz

Occupying about 16% of the processor's computational capacity during execution. This is also given by the fact that the graph must be rendered with the weight on the arcs and therefore becomes graphically more complex.

6 Conclusion

This project allowed us to deepen the Prolog language and therefore become familiar with a new programming and reasoning method that allowed us to find the solutions we sought to solve the various problems found by reasoning from different points of view.