UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in
Ingegneria Informatica e dell'Automazione

Process Repairing basato su sottoprocessi anomali frequenti

Professori:
Prof. Potena Domenico
Dott.ssa Genga Laura

Sviluppatori:
Bedetta Alessandro
Caporusso Chiara Amalia

Anno Accademico 2021-2022

Indice

Capitolo 1

Introduzione

Nel seguente articolo verrà illustrato lo svolgimento di questo progetto svolto nell'ambito del process-mining e del process-repairing. Il nostro lavoro può essere visto come l'estensione del lavoro svolto da Fabio rossi nel suo articolo *"Process repairing basato su sottoprocessi anomali frequenti"*[1]. Nel seguito verranno illustrati i risultati ottenuti da quest'ultimo e di come essi sono stati di fondamentale importanza nel raggiungimento di quelli che erano i nostri obiettivi nell'ambito del process repairing con pattern.

1.1 Process Mining e Repairing

Quando parliamo di *Process Mining*[2] ci stiamo riferendo all'estrazione di informazioni da processi. Questi ultimi vengono rappresentati tramite reti di Petri, ed il loro obiettivo principale è quello di fornire una rappresentazione il quanto più fedele possibile di tutte quelle che sono le sequenze di azioni che possono avvenire nella realtà.
Nelle situazioni reali di applicazione può succedere che di tanto in tanto vengano eseguite serie di eventi che escono da quello che è il modello, in questi casi la sequenza di azioni in questione viene definita *"non replyable"*, in quanto la struttura del modello non riesce a rappresentarla fedelmente. Ad esempio, in caso di processi aziendali succede molto spesso che a causa del così detto "Learning by doing" alcuni dipendenti trovano vie alternative e più efficienti per svolgere determinate attività; questo fa si che quello che indica il modello non rispecchia più quello che succede nella realtà.
Sono questi i motivi che ci portano a studiare il *Process Repairing*[3], ovvero la branca del Process Mining finalizzata a riparare modelli preesistenti al fine di renderli di nuovo rappresentativi delle realtà che si sono venute a creare.

1.1.1 Process Repairing basato su Sub

Un elemento fondamentale del Process Mining e del Process Repairing è il **log**, ovvero un insieme di **tracce**. Queste ultime non sono altro che sequenze di azioni

che illustrano una successione di avvenimenti interni al contesto che stiamo analizzando e possono essere rappresentate tramite grafi orientati detti Istance graphs. Il concetto di "Sequenza di azioni non replayable" prende forma nella **sub**, ovvero un sotto-grafo della nostra traccia che racchiude un comportamento anomalo e non previsto dal modello. Queste sub sono di fatto le porzioni di grafo che devono essere inserite nel modello in modo da riuscire a rappresentare anche la situazione anomala. Nei log reali quello che succede è che potremmo avere molte tracce contenenti situazioni anomale non replayable, quindi, essendo impossibile riparare inserendo ogni casistica possibile quello che si fa è selezionare le sub anomale più frequenti. La risoluzione di questo problema di ricerca, affrontato nell'articolo *Model Repair Supported by Frequent Anomalous Local Instance Graphs*[4] ci consente di avere tra le mani un elenco di quelle che sono le alterazioni più frequenti, di conseguenza sarà possibile riparare il nostro modello in maniera più efficace e efficiente.

1.1.2 Process Repairing basato su Pattern

Il process repairing basato su pattern, come si evince dall'articolo *Discovering anomalous frequent patterns from partially ordered event logs*[5] rappresenta l'evoluzione di quello basato su sub. Partiamo col dare una breve definizione del conetto di pattern; un **pattern** non è altro che un insieme di sub unite da determinate relazioni d'ordine. Un pattern potrebbe darci informazioni del tipo : "La sub x viene subito prima della sub y" oppure "La sub x è sovrapposta alla sub y e la z è collegata direttamente alla y". Di conseguenza il repairing basato su pattern indica un operazione al termine della quale il modello viene aggiustato aggiungendo le sub in questione rispettando le relazioni d'ordine imposte. Di sequito verranno riportate le principali tipologie di relazioni che legano le sub presenti nei pattern :

- **Strictly sequential** : Le due sub sono messe una di seguito all'altra andando a costituire un unico flusso di esecuzione, finita la prima si entra subito nella seconda senza possibilità di seguire altre strade nel grafo.

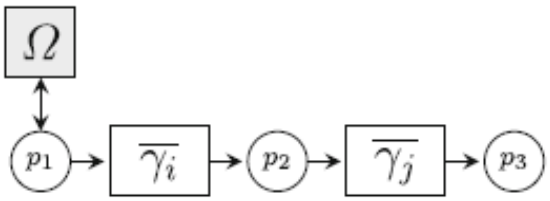

Figura 1.1: Pattern con sub in Strictly Sequential

- **Sequentially** : Le due sub sono poste una di seguito all'altra ma a differenza del caso precedente, terminata la prima sub si aprono due percorsi, uno che continua nella seconda sub e uno che continua nel modello

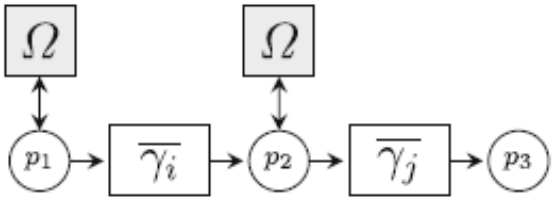

Figura 1.2: Pattern con sub in Sequentially

- **Eventually** : Questa relazione ci indica semplicemente che la seconda sub viene dopo la prima, senza contatto diretto; deve esistere quindi un percorso nel modello che permetta di raggiungere la seconda sub una volta terminata la prima.

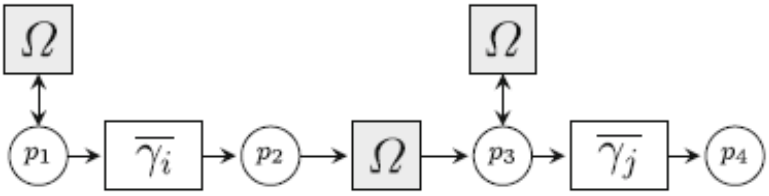

Figura 1.3: Pattern con sub in Eventually

- **Interleaving** : Quest'ultimo tipo di relazione costituisce il caso più complesso e intricato, in quanto se due sub sono legate da una relazione interleaving allora esistono alcune attività della seconda sub che iniziano ancora prima che tutte quelle della prima abbiano potuto giungere a compimento.
  È facile immaginare come questo generi due casistiche possibili: infatti potremmo avere le due sub parzialmente/completamente in parallelo oppure avere una parte della prima sub che è sovrapposta a parte della seconda.

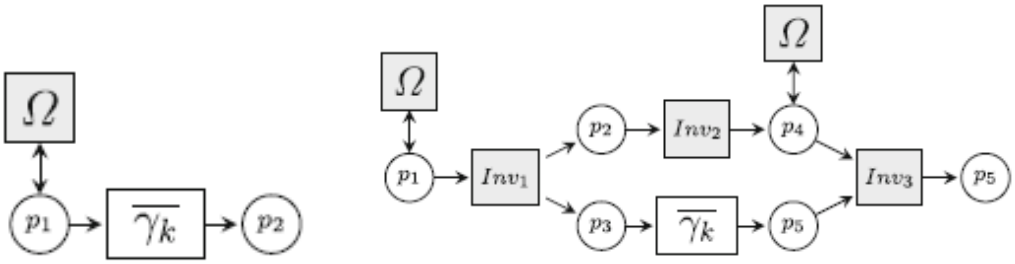

Figura 1.4: Pattern con sub in Interleaving

Capitolo 2

Stato dell'arte

2.1 Algoritmo a sub unica

Il nostro principale punto di partenza per affrontare questo problema è stato il lavoro svolto e trattato da Rossi Fabio nel suo articolo "*Process repairing basato su sottoprocessi anomali frequenti*" [1]in cui veniva approfondito lo sviluppo di un algoritmo finalizzato al repairing di modelli con sub anomale molto presenti nel log di riferimento. Di seguito verranno elencati i principali step di questo programma e i risultati da esso prodotti.

2.1.1 Input

Gli input dell'algoritmo a sub unica sono i seguenti:

- **File subs** : File .subs contenente tutte le principali sub anomale nel log; viene prodotto dall'algoritmo introdotto nell'articolo *Model Repair Supported by Frequent Anomalous Local Instance Graphs*[4]
- **Modello rete** : File con formato .pnml contenente il modello da riparare rappresentato come reti di petri
- **Log** : File con estensione .xes contenente tutte le tracce.
- **Matrice occorrenze sub** : Matrice prodotta dall'algoritmo introdotto nell'articolo *Model Repair Supported by Frequent Anomalous Local Instance Graphs*[4] che sulle righe ha id di grafi (tracce) e sulle colonne gli id delle sub. Se l'elemento di coordinate x,y è pari a 1 vuol dire che nella trace x occorre la sub y.

2.1.2 Fase preparatoria

Una volta passato l'id della sub con la quale si intende riparare il modello l'algoritmo a sub unica procede con il processo di individuazione del grafo avente il matching cost più basso; ovvero il grafo che al suo interno presenta un'istanza

(sottografo isomorfo) della sub con il minor numero di alterazioni possibili. Questo processo viene eseguito grazie ad un eseguibile chiamato **gm**.
Questo eseguibile viene applicato alla lista di tutti i grafi in cui la sub occorre, lista ottenuta andando a leggere la matrice delle occorrenze. Una volta ottenuto il grafo con matching cost più basso l'istanza della sub viene estratta dal grafo mandando in esecuzione un programma chiamato sgiso.

2.1.3 Riparazione

Ottenuta questa istanza, inizia il processo di semplificazione della sub; questo viene fatto al fine di tagliare fuori dalla sub tutte le eventuali porzioni già replayable dal modello. A tal fine vengono utilizzati i metodi *start_pre_process_repairing* e *end_pre_process_repairing* che sfruttano tecniche di replay tramite alignment[3].
Una volta ottenuta questa semplificazione l'algoritmo procede col cercare i place di aggancio della sub nel modello; per far questo vengono, come nel caso precedente, sfruttati gli alignment ed in particolare i metodi che svolgono questa operazione sono *dirk_marking_start* e *dirk_marking_end*.
Trovati questi place si passa ad effettuare l'aggancio della sub al modello tramite il metodo *repairing*; quest'ultimo controlla il numero di trasformazioni di ingresso e di uscita dalla sub, se questo numero è meggiore di uno allora viene inserita una trasformazione invisibile che raggruppa tutti gli archi verso le trasformazioni di ingresso, lo stesso viene fatto in uscita dalla sub.

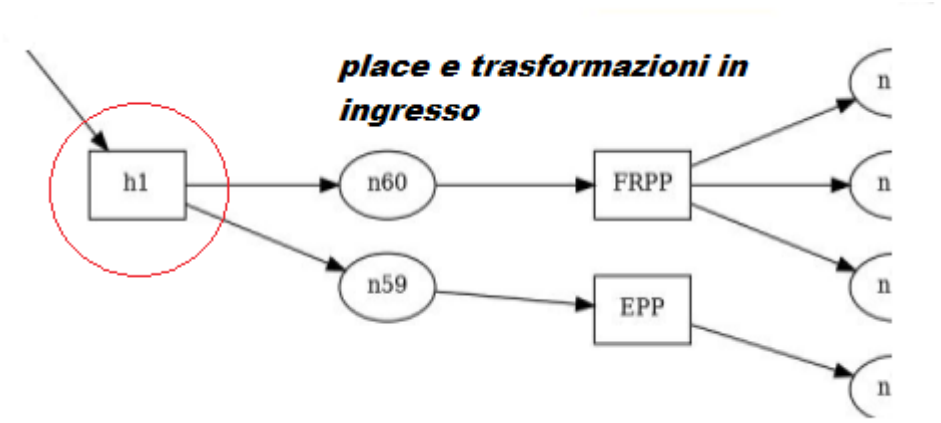

Figura 2.1: Aggancio della sub al modello

Se invece il numero di transizioni in ingresso/uscita è pari a uno allora non viene inserita nessuna transizione invisibile e i place vengono semplicemente collegate all'unica trasformazioni in ingresso/uscita.

Capitolo 3

Strumenti e metodi

3.1 Linguaggi e librerie

Tutto il codice presente nel progetto è stato scritto in linguaggio Python e sfruttando svariate librerie legate al mondo del Process Mining. La più importante fra queste è sicuramente **PM4PY**, ovvero una libreria finalizzata ad offrire una vasta gamma di metodi nell'ambito del process mining e repairing, mettendo anche a disposizione moltissimi algoritmi che costituiscono lo stato dell'arte in questa materia.
Oltre a Pm4Py sono state sfruttate alcune librerie aggiuntive atte a gestire svariate operazioni di base come la connessione a database SQL, la visualizzazione di informazioni relative l'esecuzione e la visualizzare i grafi. Un'altra libreria importante è pandas utilizzata per la manipolazione e l'analisi dei dati.

3.2 Strumenti

Per lo sviluppo di questo progetto, è necessario installare i seguenti tool:

- Installare la versione C di SUBDUE disponibile al seguente link. Posizionare gli eseguibili sgiso e gm generati nella cartella di dataset che si vuole testare;
- Installare le libreria Python PM4PY, pandas, IncrementalBar;
- Installare il tool XAMPP, verificare che il server mysql sia in run, importare il database presente nella cartella del dataset che si vuole testare. Controllare che i file relativi alla connessione al database siano conformi rispetto alle credenziali e alla porta in uso.
- Installare il software VirtualBox: è necessario predisporre una macchina virtuale per tutti coloro che non possiedono un sistema operativo macOS o qualsiasi altro sistema Unix-like per mandare in esecuzione il programma.

Ulteriore documentazione sugli strumenti utilizzati ed ulteriori passaggi specifici è disponibile al seguente link.

Capitolo 4

Sviluppo del progetto

4.1 Obiettivi

L'obiettivo principale del nostro lavoro è stato quello di espandere l'algoritmo di Fabrio Rossi al fine di implementare un software per il repairing di modelli fatto con **pattern di massimo due sub**. Queste operazioni di riparazione devono essere fatte ovviamente nel rispetto delle relazioni d'ordine fra le varie sub, inoltre bisogna anche tenere sotto osservazione la precisione di queste modifiche.
Nel fare un repairing, bisogna stare attenti a non complicare troppo il modello, mantenendo quindi un certo bilanciamento fra componenti aggiunte e semplicità; ad esempio sarebbe ottimale mantenere il numero di componenti duplicate al minimo possibile, andando a sfruttare al massimo quanto di già presente nel modello.

4.2 Sviluppo

Come fatto nel caso dell'algoritmo a sub unica procederemo andando ad analizzare le principali fasi del software da noi sviluppato.

4.2.1 Input

Gli input al nostro programma sono molto simili a quelli descritti nel caso dell'algoritmo di repairing a sub unica, con la differenza che nel nostro caso sono stati considerati interi pattern.

- **Matrice occorrenza pattern** : Matrice che sulle righe presenta gli id dei grafi e sulle colonne gli id dei pattern; se l'elemento di posizione x,y è 1 allora vuol dire che nel grafo x abbiamo 1 occorrenza del pattern y.

- **Matrice occorrenze sub** : Matrice prodotta dall'algoritmo introdotto nell'articolo *Model Repair Supported by Frequent Anomalous Local Instance Graphs*[4] che sulle righe ha id di grafi e sulle colonne gli id delle sub. Se l'ele-

mento di coordinate x,y è pari a 1 vuol dire che nella trace x occorre la sub y.

- **Log** : Elenco delle tracce di riferimento.
- **Modello** : File .pnml contenente il modello della rete su cui effettuare il repairing sotto forma di rete di Petri.
- **File subs** : File .subs contenente tutte le principali sub anomale nel log;
- **File pattern** : File con estensione .subs contente tutti i principali pattern anomali, le sub che li compongono e le rispettive relazioni che le legano.

Di seguito vengono riportati i passaggi fondamentali:

```
1
2     #Selezione del pattern con cui si vuole riparare il modello (
    ad es. 15)
3     pattern_num = 15
4
5     #Variabile globale utilizzata per decidere se si vuole
    riparare o meno con precisione (solo nel caso sequentially)
6     precision_mode = True
7
8     #Estrazione del pattern dal file .subs
9     b = create_patterns_list(path+"
    testBank2000NoRandomNoise_new_patterns_filtered_original.subs")
10    selected_subgraphs = b[pattern_num]
11    for idx, x in enumerate(selected_subgraphs):
12        #Salvataggio del pattern selezionato
13
14    #import file di log
15    log = xes_importer.apply(path+path_cartella + dataset + '.xes'
    )
16
17    #import modello della rete
18    net, initial_marking, final_marking = pnml_importer.apply(path
    +path_cartella + dataset + '_petriNet.pnml')
```

4.2.2 Fase preparatoria

Una volta specificato l'id del pattern con il quale si intende riparare, l'algoritmo inizia con l'andare a leggere dal dataset la struttura del pattern in questione, quindi relazioni e sub che lo costituiscono.
Fatto questo vengono letti Log e Modello; successivamente, come succedeva per l'algoritmo a sub unica si procede con la ricerca del grafo avente costo di matching più basso.
In quanto stiamo operando con pattern, e non con sub uniche, quello che si fa in questo caso è ricorrere ad un approssimazione; ovvero come matching cost viene considerata la somma dei matching cost delle sub incluse nel pattern. Quindi, a

livello implementativo, viene di fatto messa in un ciclo iterativo la procedura vista per l'algoritmo ad una sub unica.

Di seguito vengono riportati i passaggi fondamentali:

```
    #estrazione delle trace dal database 'testbanklaura_new'
    dict_trace = create_dict_trace("testbanklaura_new")

    #Individuazione grafi in cui occorre il pattern (si utilizza
   il file .csv generato tramite il file OccurrencePatterns.py)
    selected_graphs = list_pattern_occurence(path + path_cartella+
    dataset + "_pattern_occurrence_matrix.csv", str(pattern_num+1)
   )

    #Vengono calcolati i costi associati a ciascun grafo,
   considerando entrambe le sub
    costs = {}
    for graph in selected_graphs:
        costs[graph] = 0
    for sub in selected_subgraphs:
        temp = create_dict_graph(path+path_cartella, "sub",
   selected_graphs, sub)
        for element in temp:
            costs[temp[element][0]]+=temp[element][1]

    #vengono ordinati i grafi per matching cost crescente
    ordered_costs = {k: v for k, v in sorted(costs.items(), key=
   lambda item: item[1])}

    #viene estratto il grafo con matching cost piu basso
    chosen_graph = next(iter(ordered_costs))
```

A questo punto, individuato il grafo con matching cost più basso, vengono estratte le istanze delle sub (sempre tramite l'eseguibile sgiso) e viene recuperata la traccia corrispondente al grafo. Fatto ciò dal pattern viene letta la relazione d'ordine tra le sub che lo compongono (nel nostro caso al massimo due):

```

    for sub in selected_subgraphs:
        #salvataggio delle istanze delle sub appartenenti al
   pattern
        #esecuzione sgiso all'interno del metodo find_istances
        graph_istances.append(find_instances(sub,chosen_graph,path
   +path_cartella))

    #individuazione della traccia
    trace = search_trace(log, dict_trace, chosen_graph)

    #split del pattern in sub e relazione
    a = split_subgraph(path+"
   testBank2000NoRandomNoise_new_patterns_filtered_original.subs")

    #split del grafo scelto con matching cost piu basso
```

```
14     chosen_graph_split = split_subgraph(path+path_cartella+"graphs
    /"+chosen_graph+".g")
15
16     for i in a:
17         #individuazione della relazione d'ordine tra le sub
```

4.2.3 Implementazione generale

Una volta preparati tutti i componenti necessari si può passare alla fase del repairing vero e proprio. Nel far ciò la logica principale seguita è stata quella dello sfruttare il più possibile l'algoritmo a sub unica e quindi, quando possibile cercare di ricondursi al caso più semplice.
Di seguito viene riportato in pseudo-codice il costrutto if che ingloba al suo interno la gestione delle quattro diverse tipologie di relazioni.

```
1      for relation in subs_relation:
2          if relation[2]=='strictlySeq' or relation[2]=='
    sequentially':
3              #uniamo le due sub
4              if (relation[2] == 'sequentially' and not
    precision_mode):
5                  #repairing approssimato
6              if (relation[2] == 'sequentially' and precision_mode):
7                  #applicazione algoritmo Bellman-Ford
8              if relation[2] == 'strictlySeq' or (relation[2] == '
    sequentially' and precision_mode):
9                  #repairing Strictly Sequential e repairing
    accurato
10         elif relation[2] == 'eventually':
11             #repairing eventually
12         elif relation[2] == 'interleaving':
13             #repairing interleaving
14
15         #visualizzazione rete riparata
16         visualizza_rete_performance(log,net_repaired,
    initial_marking,final_marking)
```

4.2.4 Strictly Sequential

Nel caso di due sub collegate da questo tipo di relazione, quello che si è pensato, come detto in precedenza quando si descriveva la logica generale, è stato di trattare il "blocco" costituito dalla sequenza delle due sub come una sub unica.
Di conseguenza è bastato trovare le trasformazioni finali (quelle con soli archi in ingresso) della prima istanza di sub e collegarle con quelle iniziali (archi solo in uscita) della seconda istanza.
Fatto ciò il risultato di questa unione è stato passato al metodo di repairing dell'algoritmo a sub unica che ha permesso di collegare come descritto nei capitoli precedenti il blocco al modello.

```
    if relation[2]=='strictlySeq' or relation[2]=='sequentially':
        #uniamo le due sub
        for x in range(len(istances[0])):
            #aggiunta delle attivita della prima sub al pattern
        for x in range(len(istances[1])):
            #aggiunta delle attivita della seconda sub al pattern
        for x in range(len(istances[0])):
            #aggiunta delle transizioni della prima sub al pattern
        for x in range(len(istances[1])):
            #aggiunta delle transizioni della seconda sub al pattern
        for y in chosen_graph_split:
            #aggiunta della transizione che collega la prima sub con la seconda
        if relation[2]=='strictlySeq' or (relation[2]=='sequentially' and precision_mode):

            #individuazione dei nodi di start e di end del pattern con cui si vuole riparare il modello
            start, end, sub_label = startend_node(final_pattern)
            #ricerca dell'alignment
            text = search_alignment(path+path_cartella, dict_trace, chosen_graph)
            #fase di repairing in cui viene semplificato il pattern
            new_final_pattern = start_pre_process_repairing(start, text, final_pattern)
            new_subgraph = end_pre_process_repairing(end, text, new_final_pattern)

            #individuazione dei nuovi nodi di start e di end del pattern
            start, end, sub_label = startend_node(new_subgraph)

            #individuazione dei marking di start e di end a cui agganciare il pattern al modello
            reached_marking_start = dirk_marking_start(dataset, start, text, trace, path+path_cartella, selected_subgraphs[0]+"_"+selected_subgraphs[1])
            reached_marking_end = dirk_marking_end(dataset, end, text, trace, path+path_cartella, selected_subgraphs[0]+"_"+selected_subgraphs[1])

            #aggangio al modello
            start_end_name, net_repaired = repairing(new_subgraph, net, initial_marking, final_marking, start, end, reached_marking_start, reached_marking_end, path+path_cartella, selected_subgraphs[0]+"_"+selected_subgraphs[1])
```

4.2.5 Sequentially: Algoritmo Bellman-Ford

Nella relazione di sequentially, a differenza del caso precedente dove finita la prima sub si è costretti a continuare nella seconda in un flusso unico, si ha una scelta; ovvero arrivati alla fine della prima sub si può decidere se continuare nella seconda, oppure passare dal modello per tornare nella sub numero due.
Dopo svariate implementazioni, si è optato per sfruttare l'algoritmo di Bellman-Ford. Quest'ultimo, nella sua versione più generale, dato un grafo orientato e un nodo di partenza, genera l'albero dei percorsi minimi dal punto di partenza verso ogni altro nodo. Nel nostro caso ovviamente i costi di tutti gli archi erano settati ad 1.
La logica dietro il suo funzionamento può essere compresa sapendo che il percorso minimo per arrivare a un nodo è dato dal percorso minimo per arrivare al nodo precedente sommato alla distanza più breve fra i due nodi.
Una volta terminata l'esecuzione dell'algoritmo, quello che si ha è un elenco di nodi legati a due valori: uno è un id che indica il nodo precedente su quel ramo e il secondo è un intero che rappresenta la distanza in termini di balzi dal nodo di partenza.
Tornando ora a parlare della nostra implementazione, quello a cui si è pensato, è stato applicare l'algoritmo di Bellman-Ford al grafo con matching cost più basso, considerando come nodi di partenza le trasformazioni finali della prima sub. In questo modo per ognuna di queste transizioni è stato generato un albero con al suo interno i percorsi minimi verso le trasformazioni di ingresso della seconda sub non direttamente collegate a quelle di end della prima (trasformazioni in strictly sequentially).
Ottenuto ciò, per ognuno dei nodi di start della seconda sub non già collegati alla prima viene identificato l'albero (e quindi il nodo di end della prima sub) avente il percorso di costo minore verso il nodo in questione.
A questo punto viene ricostruito a ritroso (ogni trasformazione dell'albero ha un puntatore a quella che la precede nel percorso) il path minimo selezionato.
Fatto tutto questo procediamo con il repairing trattando l'unione delle due sub e i percorso minimo come una sub unica, è quindi riutilizzando l'algoritmo a sub unica.
Nella pseudocodifica sotto riportata è possibile osservare una variabile chiamata **precision_mode**; si tratta di una variabile booleana che permette di scegliere quale repairing utilizzare per la relazione di sequential.
La scelta è fra l'algoritmo con Bellman-Ford (True) e l'algoritmo approssimato che sarà illustrato nel seguito (False).

```
1     if (relation[2]=='sequentially' and precision_mode):
2
3         for id in enumerate(final_pattern):
4             #individuazione nodi di start della seconda sub non
    agganciati direttamente ai nodi della seconda sub
5
6         #i nodi di partenza per l'algoritmo di Bellman-Ford
    saranno tutti i nodi di end della prima sub
```

```
7         bellman_starts = end_prima
8
9         for row in chosen_graph_split:
10            #individuazione nodi e archi
11
12        for bellman_start in bellman_starts:
13            #inizializzazione costi
14            for node in nodes:
15                #inizializzazione dei costi dei nodi diversi da
    quelli iniziali ad infinito ed inizializzazione del costo del
    nodo iniziale a zero
16
17            #applicazione algoritmo Bellman-Ford
18            for node in nodes:
19                #calcolo dei costi per ogni nodo e costruzione
    dell'albero
20
21        #per ogni nodo di start della seconda sub non agganciato
    direttamente ai nodi di end della prima sub
22        for nodo in end_shortest_path:
23            #per ogni albero
24            for key, value in trees.items():
25                if value[0][nodo] < min_cost:
26                    #viene preso l'albero di costo minimo
27
28            #fintanto che non si arriva al nodo di end della prima
     sub, considerato come nodo di start per l'algoritmo di Bellman
    -Ford
29            while(nodo != current_bellman_start):
30                for row in chosen_graph_split:
31                    #procedendo a ritroso viene costruito l'albero
     di costo minimo
32                for id in enumerate(final_pattern):
33                #controllo se i nodi e gli archi sono presenti o
    meno nel pattern finale, altrimenti vengono inseriti
34         if relation[2]=='strictlySeq' or (relation[2]=='sequentially'
    and precision_mode):
35        #repairing del pattern finale
```

4.2.6 Sequentially: Approssimazione

In questo caso è stata adottata una strategia differente rispetto alla precedente in quanto, al crescere del numero di nodi cresce anche il numero di alberi che bisogna andare a costruire. Inoltre il repairing con Bellman-Ford andando ad aggiungere percorsi alla "sub" finale , in caso di molte trasformazioni finali e iniziali nelle due sub, complicherebbe molto il modello prodotto.
È stato quindi deciso di implementare una strategia differente, anche se approssimata, della relazione d'ordine Sequentially.
Dopo aver individuato l'alignment, andiamo ad individuare i marking di start e di end di ciascuna sub, in modo da verificare successivamente se questi coincidono

con quelli calcolati dopo la semplificazione. Si vanno quindi ad effettuare i tagli della prima sub, calcolando i nuovi marking di start e di end, dopodiché quest'ultima viene agganciata al modello. A questo punto si effettua il repairing con la seconda sub, seguendo i medesimi passaggi.
Dopo aver riparato il modello, se i marking di start e di end pre processing non coincidono con i marking di start e di end post processing, viene aggiunta una transizione invisibile tra i marking di start della seconda sub e i marking di end della prima sub, in modo da garantire comunque un passaggio nel modello.

```
    if (relation[2]=='sequentially' and not precision_mode):

        #individuazione alignment
        text = search_alignment(path+path_cartella, dict_trace,
    chosen_graph)

        #individuazione dei marking raggiungibili di start/end per
     la prima e la seconda sub, prima del repairing
        start_1, end_1, sub_label = startend_node(istances[0])
        reached_marking_start1_pre = dirk_marking_start(dataset,
    start_1, text, trace, path+path_cartella, selected_subgraphs
    [0])
        reached_marking_end1_pre = dirk_marking_end(dataset, end_1
    , text, trace, path+path_cartella, selected_subgraphs[0])
        start_2, end_2, sub_label = startend_node(istances[1])
        reached_marking_start2_pre = dirk_marking_start(dataset,
    start_2, text, trace, path+path_cartella, selected_subgraphs
    [1])
        reached_marking_end2_pre = dirk_marking_end(dataset, end_2
    , text, trace, path+path_cartella, selected_subgraphs[1])

        #semplificazione della prima sub
        start_pre1, end_pre1, sub_label = startend_node(istances
    [0])
        sub1 = start_pre_process_repairing(start_pre1, text,
    istances[0])
        new_sub1 = end_pre_process_repairing(end_pre1, text, sub1)
        start1, end1, sub_label = startend_node(new_sub1)

        #individuazione dei nuovi marking raggiungibili della
    prima sub dopo la semplificazione
        reached_marking_start1 = dirk_marking_start(dataset,
    start1, text, trace, path+path_cartella, selected_subgraphs[0])
        reached_marking_end1 = dirk_marking_end(dataset, end1,
    text, trace, path+path_cartella, selected_subgraphs[0])

        #aggancio prima sub semplificata al modello
        start_end_name, net_repaired = repairing(new_sub1, net,
    initial_marking, final_marking, start1, end1,
    reached_marking_start1, reached_marking_end1, path+
    path_cartella, selected_subgraphs[0])

        #semplificazione della seconda sub
```

```
28          start_pre2, end_pre2, sub_label = startend_node(istances
    [1])
29          sub2 = start_pre_process_repairing(start_pre2, text,
    istances[1])
30          new_sub2 = end_pre_process_repairing(end_pre2, text, sub2)
31          start2, end2, sub_label = startend_node(new_sub2)
32
33          #individuazione dei nuovi marking raggiungibili della
    seconda sub dopo la semplificazione
34          reached_marking_start2 = dirk_marking_start(dataset,
    start2, text, trace, path+path_cartella, selected_subgraphs[1])
35          reached_marking_end2 = dirk_marking_end(dataset, end2,
    text, trace, path+path_cartella, selected_subgraphs[1])
36
37          #aggancio della seconda sub semplificata al modello,
    utilizzando la rete riparata
38          start_end_name, net_repaired = repairing(new_sub2,
    net_repaired, initial_marking, final_marking, start2, end2,
    reached_marking_start2, reached_marking_end2, path+
    path_cartella, selected_subgraphs[1])
39
40
41          if(reached_marking_end1_pre != reached_marking_end1 or
    reached_marking_start2_pre != reached_marking_start2):
42              #se i marking raggiungibili di start/end pre
    processing non coincidono con quelli post processing, viene
    aggiunta una transizione invisibile
43              for v in reached_marking_start2:
44                  #si aggiunge un arco tra la transizione invisibile
     e i marking di start della seconda sub
45              for v in reached_marking_end1:
46                  #si aggiunge un arco tra i marking di end della
    prima sub e la transizione invisibile
```

4.2.7 Eventually

La soluzione adottata per la relazione di Eventually è per certi aspetti simile a quella illustrata nel caso del Sequentially approssimato. Di fatti, quello che si va a fare è un repairing sequenziale delle due sub, andandole a trattare quindi in maniera completamente separata. L'unica differenza sta nel fatto che, effettuate le due riparazioni, viene collegata una transizione invisibile fra le due sub in modo particolare.
Ovvero, per garantire che la seconda sub venga eseguita solo sopo aver terminato la prima, il collegamento con la transizione h non parte dai place finali della prima sub, ma bensì dalle trasformazioni finali. Questa operazione, che viene effettuata anche per le transizioni iniziali della seconda sub, consente di inserire dei place aggiuntivi agli estremi della trasformazione h.
Quindi quest'ultima nel caso di eventually forza l'esecuzione della sub 1 prima che sia possibile eseguire la 2. Questo appunto perché i place aggiuntivi collegati alle

transizioni di start della seconda sub ne bloccano l'esecuzione fino a che non viene eseguita la h.

```
    elif relation[2]=='eventually':

        #ricerca alignment
        text = search_alignment(path+path_cartella, dict_trace,
chosen_graph)

        #per ogni sub andiamo a fare il repairing
        for idx,graph_istance in enumerate(istances):

            #individuazione dei nodi di start/end della sub in
considerazione
            start, end, sub_label = startend_node(graph_istance)

            #semplificazione della sub
            new_graph_istance = start_pre_process_repairing(start,
 text, graph_istance)
            final_graph_istance = end_pre_process_repairing(end,
text, new_graph_istance)

            #individuazione nuovi nodi di start/end e marking
raggiungibili
            start, end, sub_label = startend_node(
final_graph_istance)
            reached_marking_start = dirk_marking_start(dataset,
start, text, trace, path+path_cartella, selected_subgraphs[idx
])
            reached_marking_end = dirk_marking_end(dataset, end,
text, trace, path+path_cartella, selected_subgraphs[idx])

            #attacco al modello
            start_end_name, net_repaired = repairing(
final_graph_istance, net, initial_marking, final_marking, start
, end, reached_marking_start, reached_marking_end, path+
path_cartella, selected_subgraphs[idx])

            #recupero delle transizioni di start della seconda sub
 e transizioni di end della prima sub
            if idx==0:
                string_trans_end = start_end_name[1]
            else:
                string_trans_start = start_end_name[0]

            #individuazione delle transizioni e dei places della
rete riparata
            transitions = net_repaired.transitions
            places = net_repaired.places

            for v in transitions:
                if v.name in string_trans_end:
                    #recupero delle transizioni end della prima
sub dalla rete riparata
```

```
37
38              for v in transitions:
39                  if v.name in string_trans_start:
40                      #recupero delle transizioni di start della
    seconda sub dalla rete riparata
41
42              #individuazione della transizione invisibile da
    aggiungere alla rete riparata per garantire che l'esecuzione
    della prima sub termini prima dell'inizio dell'esecuzione della
     seconda sub
43              n = transition_hidden_available(transitions)
44              t = PetriNet.Transition("h" + n, None)
45              net_repaired.transitions.add(t)
46
47              #aggiunta di nuovi place per il collegamento della
    transizione invisibile
48              n = places_name_available(places,transitions)
49              place1 = PetriNet.Place("n"+n)
50              net_repaired.places.add(place1)
51              n = places_name_available(places,transitions)
52              place2 = PetriNet.Place("n"+n)
53              net_repaired.places.add(place2)
54
55              for v in trans_end:
56                  #attacco delle transizioni di end della prima sub
    al primo place disponibile
57
58              for v in trans_start:
59                  #attacco delle transizioni di start della seconda
    sub al secondo place disponibile
```

4.2.8 Interleaving

La relazione di Interleaving, come riportato nell'introduzione, si divide in due casistiche, una più generale e una più particolare. Nella maggior parte dei casi quello che si ha è una sovrapposizione, ovvero parti iniziali della seconda sub sono le stesse parti finali della prima. In altri casi invece può capitare che le componenti non siano sovrapposte, ma bensì parallele; quindi mentre stanno terminando le trasformazioni finali della prima sub, in maniera concorrente iniziano quelle di start della seconda.
Per modellare la prima casistica è stato seguito un approccio simile a quello attuato nel caso dello strictly sequentially, con la differenza che in questo caso bisogna fare attenzione a non inserire due volte quelle che sono le trasformazioni sovrapposte; le due sub vengono quindi unite e trattate come una sub unica.
La casistica di sub parallele è già implementata nell'algoritmo a sub unica. Una volta unite le due sub, se non ci sono transizioni in comune, avremo per forza di cose due o più transizioni di start e due o più transizioni di end, di conseguenza applicando l'algoritmo a sub unica a questa sub unificata, verranno aggiunte due

trasformazioni h invisibili alla fine e all'inizio (meccanismo descritto nel paragrafo 2.1.3) e il tutto poi viene collegato al modello.

```
    elif relation[2] == 'interleaving':

    if len(istances[0]) > len(istances[1]):
        #individuazione dell'istanza di sub piu grande tra le due

    for idx, element in enumerate(tiny_sub_instance):
        #individuazione di eventuali nodi e transizioni gia presenti nell'istanza di sub piu grande
        if(tiny_sub_instance[idx+1] not in big_sub_instance):
            #se non presenti, le attivita e/o le transizioni vengono aggiunte all'istanza di sub piu grande

    #ricerca dell'alignment
    text = search_alignment(path+path_cartella, dict_trace, chosen_graph)

    #fase di repairing con semplificazione della sub ed individuazione dei marking di start/end raggiungibili
    start, end, sub_label = startend_node(final_pattern)
    new_final_pattern = start_pre_process_repairing(start, text, final_pattern)
    new_subgraph = end_pre_process_repairing(end, text, new_final_pattern)
    start, end, sub_label = startend_node(new_subgraph)
    reached_marking_start = dirk_marking_start(dataset, start, text, trace, path+path_cartella, selected_subgraphs[0]+"_"+selected_subgraphs[1])
    reached_marking_end = dirk_marking_end(dataset, end, text, trace, path+path_cartella, selected_subgraphs[0]+"_"+selected_subgraphs[1])

    #aggancio al modello ed eventuale aggiunta di transizioni invisibili nel caso in cui le due sub fossero parallele e non avessero transizioni in comune
    start_end_name, net_repaired = repairing(new_subgraph, net, initial_marking, final_marking, start, end, reached_marking_start, reached_marking_end, path+path_cartella, selected_subgraphs[0]+"_"+selected_subgraphs[1])
```

Capitolo 5

Risultati

5.1 Modello pre-repairing

Nell'immagine 5.1 viene riportato il modello che verrà preso come esempio, sotto forma di rete di Petri, da riparare con i pattern anomali più frequenti:

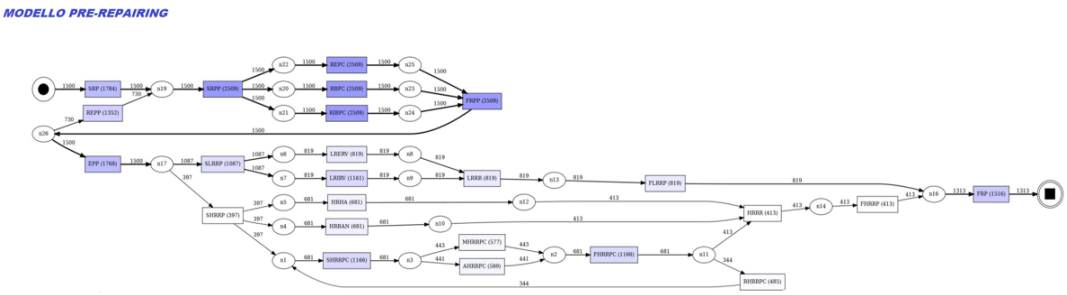

Figura 5.1: Modello pre-repairing

5.2 Strictly Sequential

Il pattern scelto per riparare il modello è il numero 11, in cui la sub 2 e la sub 17 sono collegate mediante la relazione d'ordine Strictly Sequential, come riportato nell'immagine 5.2:

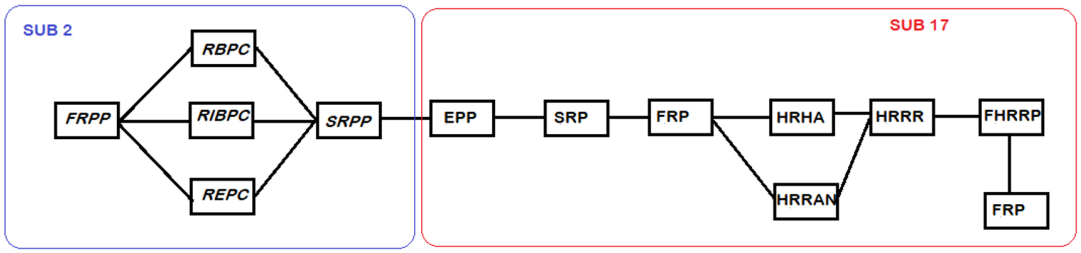

Figura 5.2: Pattern con sub in Strictly Sequential

Il risultato che si ottiene dopo la fase di repairing, con l'introduzione del pattern nel modello è riportato nell'immagine 5.3:

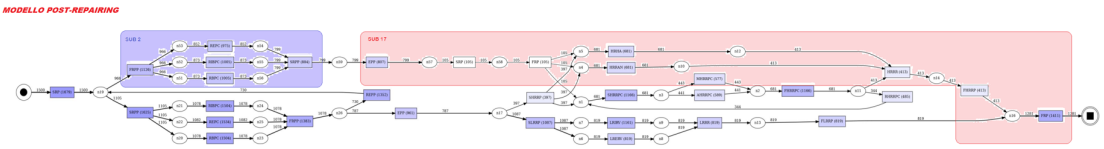

Figura 5.3: Risultato post-repairing Strictly sequential

5.3 Sequentially

Il pattern scelto per riparare il modello è il numero 17, in cui la sub 2 e la sub 14 sono collegate mediante la relazione d'ordine Sequentially, come riportato nell'immagine 5.4:

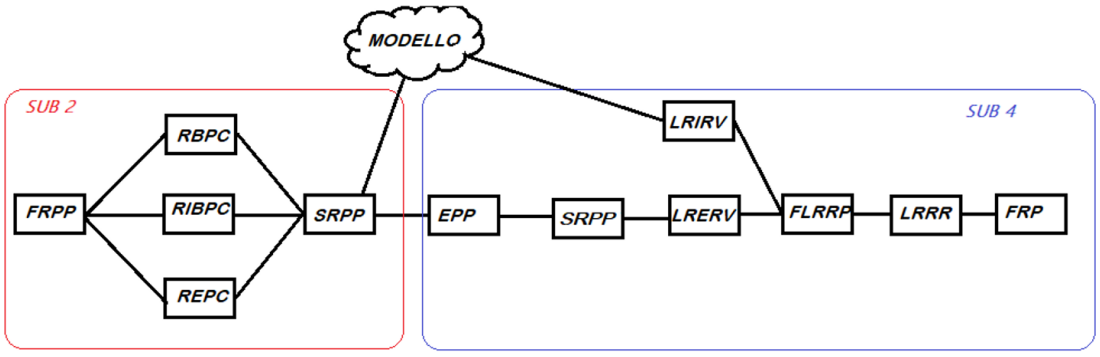

Figura 5.4: Pattern con sub in sequentially

5.3.1 Repairing preciso: Bellman-Ford

Il risultato che si ottiene dopo la fase di repairing, con l'introduzione del pattern nel modello è riportato nell'immagine 5.5:

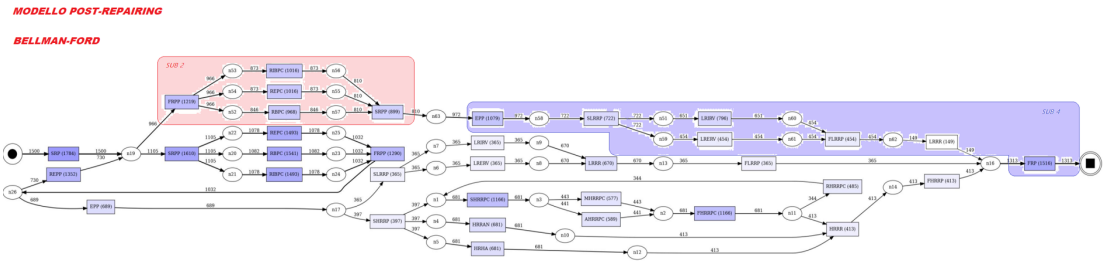

Figura 5.5: Repairing con Bellman-Ford

5.3.2 Repairing approssimato

Il risultato che si ottiene dopo la fase di repairing, con l'introduzione del pattern nel modello è riportato nell'immagine 5.6:

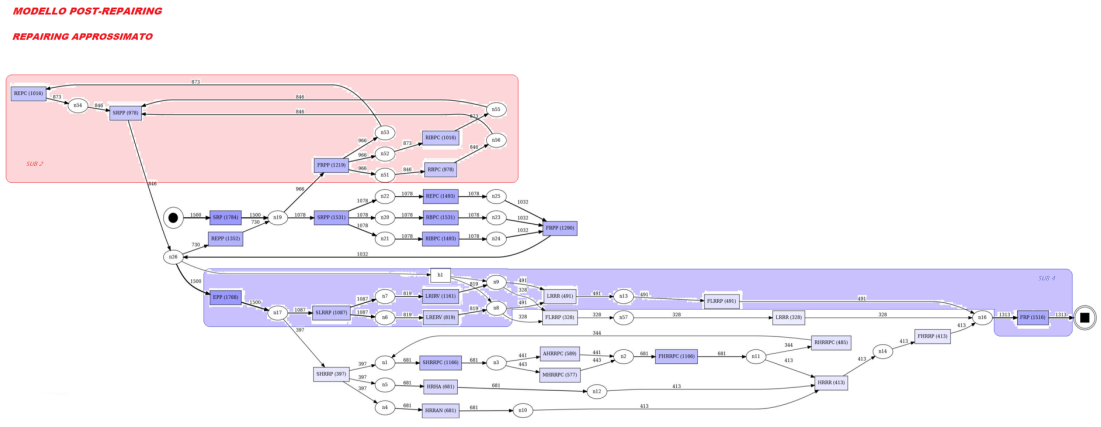

Figura 5.6: Repairing con approssimazione

5.4 Eventually

Il pattern scelto per riparare il modello è il numero 15, in cui la sub 105 e la sub 4 sono collegate mediante la relazione d'ordine Eventually, come riportato nell'immagine 5.7:

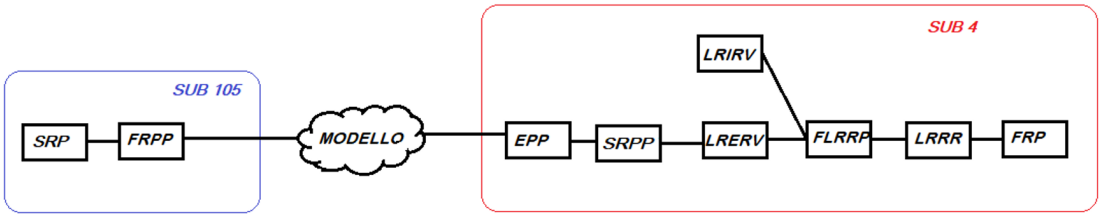

Figura 5.7: Pattern con sub in eventually

Il risultato che si ottiene dopo la fase di repairing, con l'introduzione del pattern nel modello è riportato nell'immagine 5.8:

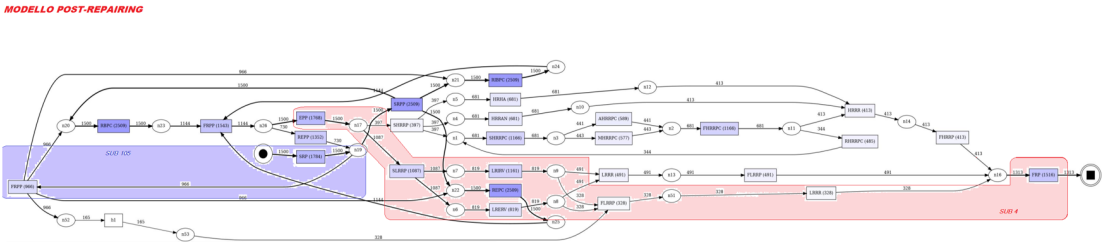

Figura 5.8: Risultato post-repairing con sub in eventually

5.5 Interleaving

Per quanto riguarda questa relazione d'ordine, viene fatta un'ulteriore distinzione nel caso in cui le due sub sono in overlapping oppure parallele.

5.5.1 Interleaving: caso overlapping

Il pattern scelto per riparare il modello è il numero 21 in cui la sub 63 e la sub 2 sono collegate mediante la relazione d'ordine interleaving, come riportato nell'immagine 5.9:

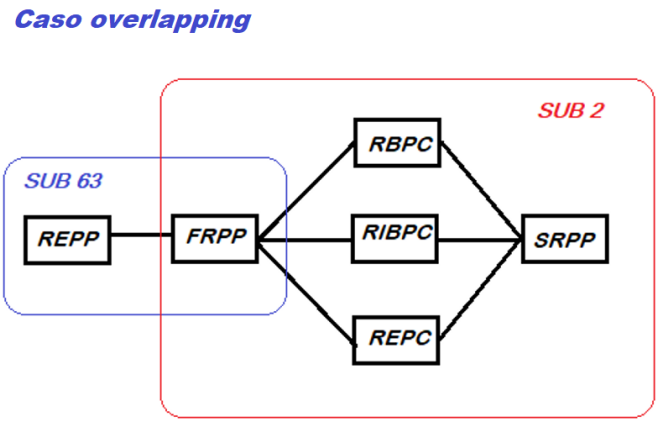

Figura 5.9: Pattern con sub in interleaving: caso overlapping

Il risultato che si ottiene dopo la fase di repairing, con l'introduzione del pattern nel modello è riportato nell'immagine 5.10:

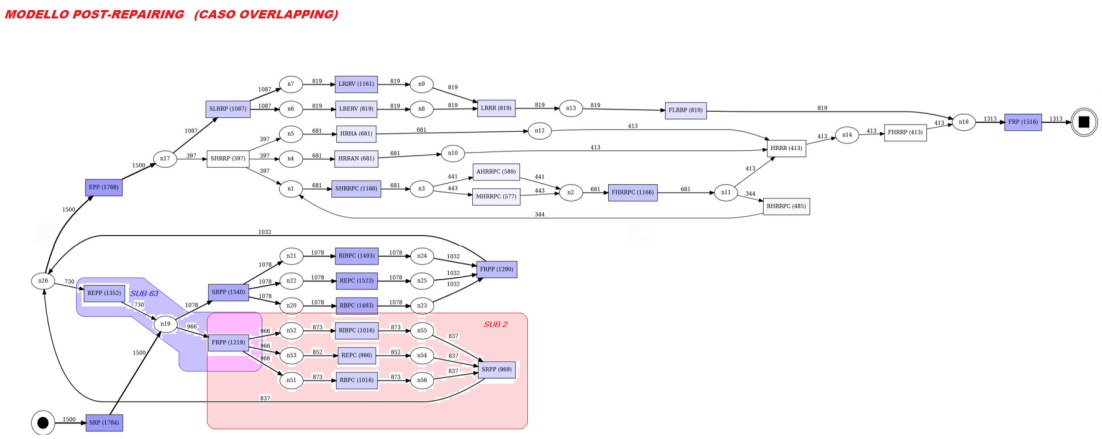

Figura 5.10: Risultato post-repairing con sub in interleaving: caso overlapping

5.5.2 Interleaving: caso parallelo

Il pattern scelto per riparare il modello è il numero 11, modificato in modo tale che la sub 2 e la sub 17 siano collegate mediante la relazione d'ordine interleaving, come riportato nell'immagine 5.11:

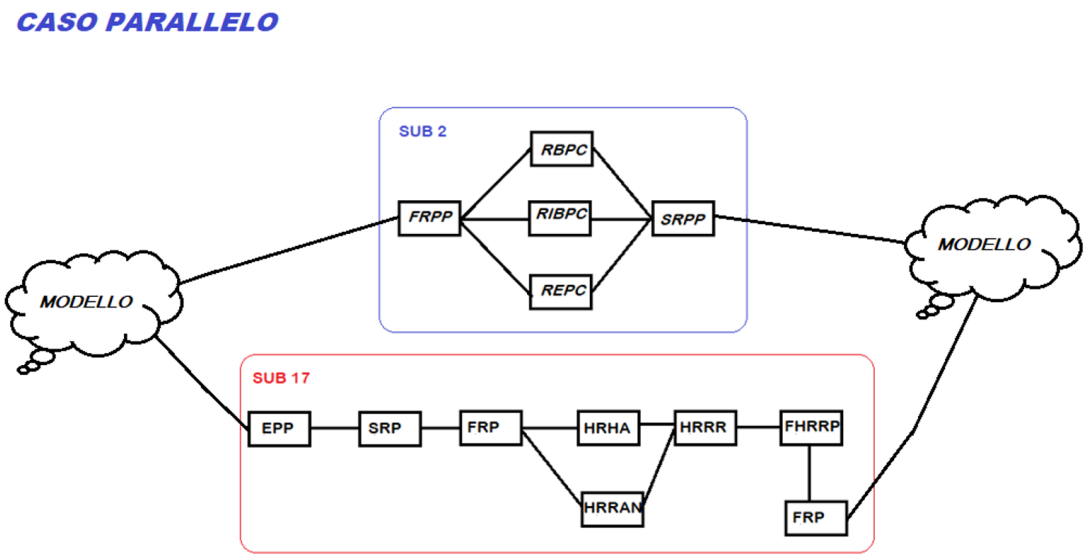

Figura 5.11: Pattern con sub in interleaving: caso parallelo

Il risultato che si ottiene dopo la fase di repairing, con l'introduzione del pattern nel modello è riportato nell'immagine 5.12:

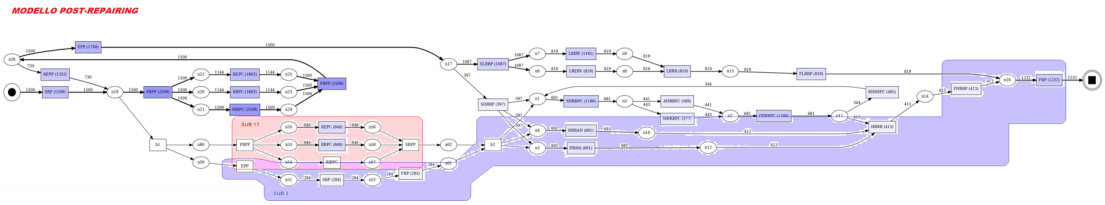

Figura 5.12: Risultato post-repairing con sub in interleaving: caso parallelo

Capitolo 6

Conclusioni e Sviluppi futuri

Ricapitolando, si è proposto un approccio esteso rispetto a quello principale, in quanto il modello viene riparato con pattern composti da due sub piuttosto che una sola sub. Si sono considerate le diverse relazioni d'ordine e come queste influiscono sul comportamento delle sub.
Sicuramente lo sviluppo successivo a questo lavoro è quello di riparare un modello nel caso generale, ovvero con pattern composti da n sub, comunque sempre con relazioni d'ordine definite per coppie di sub. È già presente nel progetto un modulo python chiamato *OccurrencesPatterns.py* il quale scopo è generare la matrice delle occorrenze dei pattern, che indica in quali grafi essi si manifestano.
Sarebbe interessante implementare una soluzione per il repairing a n sub, basato su un iterazione di quanto fatto dal nostro algoritmo; ciò vorrebbe dire procedere di coppia in coppia di sub, aggiornando di volta in volta gli alignment e il modello, magari procedendo seguendo un ordine preciso di relazioni (Es. prima le riparazioni in strictly sequential, poi le sequential etc...).
Concludendo possiamo affermare che troviamo molto interessante quanto il Process Mining sia in via di sviluppo, soprattutto per quanto riguarda la branca del Repairing di modelli che, come si può vedere, offre tante possibilità di espansione e miglioramento.

Tutto il codice dell'algoritmo di cui si è discusso è disponibile al seguente link

Bibliografia

[1] Fabio Rossi. "Process Repairing basato su sottoprocessi anomali frequenti". In: *Università Politecnica delle Marche* (2020).

[2] *Process Mining*. URL: `https://it.wikipedia.org/wiki/Process_mining`.

[3] Dirk Fahland e Wil M.P. van der Aalst. "Model Repair - Aligning process models to reality". In: *Eindhoven University of technology* (2015).

[4] Laura Genga, Claudia Diamantini Fabio Rossi e Domenico Potena Emanuele Storti. "Model Repair Supported by Frequent Anomalous Local Istance Graphs". In: *Università Politecnica delle Marche* (2020).

[5] Laura Genga, Domenico Potena Mahdi Alizadeh e Nicola Zannone Claudia Diamantini. "Discovering anomalous frequent patterns from partially ordered event logs". In: *Eindhoven University of Technology* (2018).

Elenco delle figure