eAKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

Praca dyplomowa

System wydobywania wzorców zachowania z logów systemowych i generowanie ich specyfikacji logicznej
Workflow mining from system logs generating their logical specifications

Autor: *Julia Emilia Witek*
Kierunek studiów: Informatyka i Systemy Inteligentne
Opiekun pracy: *dr hab. inż. Radosław Klimek*

Kraków, 2023

Spis treści

1 Wstęp

W czasie, gdy coraz więcej procesów biznesowych i technologicznych odbywa się w świecie cyfrowym, rejestrowanie zachowań użytkowników może być kluczowym dla przedsiębiorstw źródłem informacji o zachodzących procesach. Wiele firm i organizacji decyduje się na gromadzenie logów systemowych, które mogą służyć do monitorowania oraz zarządzania procesami zachodzącymi wewnątrz systemu. Ze względu na rosnącą ilość rejestrowanych danych, wydobycie istotnych informacji staje się czasochłonne, zwłaszcza przy skomplikowanych projektach informatycznych o licznych funkcjach. Istnieje więc potrzeba wykorzystania technik do automatycznego wydobywania informacji z logów, które pomogą w analizie i zarządzaniu danymi.

Analiza logów systemowych stanowi kluczowy element zagadnienia zwanego eksploracją procesów, umożliwiając automatyczną rekonstrukcję przebiegu poszczególnych instancji wykonywanego procesu na podstawie tzw. dzienników zdarzeń. Ponieważ ciągła optymalizacja kosztów ma kluczowe znaczenie w kontekście działalności przedsiębiorstw, konieczne staje się nieustanne doskonalenie działań w różnych obszarach działalności. Głównym celem każdej firmy powinno być gromadzenie jak największej ilości danych historycznych dotyczących prowadzonych procesów oraz efektywne ich wykorzystywanie. Analiza logów systemowych pozwala na szerokie zrozumienie przebiegu procesów, identyfikację ewentualnych problemów, a także na wskazanie obszarów, w których można wprowadzić usprawnienia.

Podejście oparte na analizie logów systemowych wspomaga podejmowanie decyzji opartych na rzeczywistych danych, co z kolei przekłada się na bardziej efektywne zarządzanie procesami i lepszą kontrolę kosztów. W efekcie, przedsiębiorstwo może zyskać konkurencyjną przewagę poprzez zoptymalizowanie swoich operacji oraz szybkie reagowanie na zmiany w otoczeniu biznesowym. W związku z tym, inwestycje w narzędzia i kompetencje związane z analizą logów systemowych stają się kluczowym elementem strategii rozwoju organizacji.

Ponieważ metody formalne są powszechnie stosowane w procesach wspomagających wytwarzanie oprogramowania, naturalnym wydaje się wykorzystanie ich także w celu analizy i weryfikacji procesów zachodzących wewnątrz systemu. Weryfikacja jest procesem nieodłącznie związanym z implementacją oraz udoskonalaniem systemu, każdy cykl implementacyjny powinien być poprzedzony oraz potwierdzony przez odpowiednią weryfikację [1]. Metody formalne mogą istotnie przyczynić się do poprawy funkcjonalności oraz niezawodności systemu, gdyż dostarczają narzędzia pozwalające na matematycznie poprawną oraz precyzyjną specyfikację wymagań oraz ich weryfikację. Dzięki temu minimalizowane jest ryzyko błędów, zwiększana jest wydajność oraz ogólna jakość projektu.

Celem pracy jest opracowanie metody przekształcającej wygenerowane modele procesu do postaci równoważnych specyfikacji logicznych. Przekształcanie to odbywa się w oparciu o znany w literaturze algorytm [20]. Praca skupia się w szczególności na modelu procesu przedstawionym w notacji drzewa procesu, ze względu na jego prostą, hierarchiczną strukturę, odpowiadającą wejściu wspomnianego algorytmu. Tak przedstawiony problem pozwolił na zdefiniowanie kolejnego celu pracy, tj. analizy właściwości logicznych otrzymanych specyfikacji przy użyciu systemów automatycznego dowodzenia twierdzeń logicznych, dla logiki pierwszego rzędu. Ponieważ wygenerowana specyfikacja przyjmuje postać formuł logiki temporalnej, omówiono przemienność operatorów temporalnych oraz kwantyfikatorów rachunku pierwszego rzędu.

2 Eksploracja procesów

Eksploracja procesów (ang. Process Mining) to zbiór wszystkich metod wykorzystywanych w analizie, weryfikacji oraz udoskonalaniu procesów biznesowych. W szczególności tyczy się to automatycznego wykrywania zależności między czynnościami zachodzącymi wewnątrz badanego procesu. Metodologie te wykorzystują zwykle logi systemowe rejestrowane przez systemy informatyczne podczas wykonywania określonych akcji przez użytkownika, stanowiących rzeczywisty przebieg procesów biznesowych. Dane w takiej postaci są następnie punktem wyjścia dla algorytmów eksploracji procesów, które generują model procesu w odpowiedniej notacji. Otrzymany model stanowi generalizację badanego procesu oraz umożliwia późniejszą analizę i optymalizację systemu. Do przykładowych algorytmów stosowanych w eksploracji procesów należą Alpha Miner, Inductive Miner czy Heuristics Miner.

2.1 Dziennik zdarzeń

Dane, stanowiące źródło informacji dla algorytmów eksploracji procesów, przechowywane są zwykle w pliku zwanym dziennikiem zdarzeń (ang. Event Log). Zapis do takiego pliku odbywa się przy założeniu, iż istnieje możliwość uporządkowanego rejestrowania zdarzeń wykonywanych w obrębie systemu w taki sposób, że każde zdarzenie odzwierciedla wykonane działanie w procesie i jest skorelowane z konkretnym przypadkiem procesu. Dziennik zdarzeń przyjmuje postać tabeli składającej się z dowolnej ilości kolumn oraz wierszy. Choć można w nim wskazać różnorodne kolumny, większość algorytmów eksploracji procesów wymaga trzech: identyfikatora przypadku, śladu czasu oraz nazwy samej czynności. Przykładowy dziennik zdarzeń zaprezentowany został jako tabela 1.

Tab. 1: Przykładowy dziennik zdarzeń [2].

Case ID	Timestamp	Activity	Resource	Cost
1	01-01-2018:08.00	Register (R)	Frank (F)	1000
2	01-01-2018:10.00	Register (R)	Frank (F)	1000
3	01-01-2018:12.10	Register (R)	Joey (J)	1000
3	01-01-2018:13.00	Verify-Documents (V)	Monica (M)	50
1	01-01-2018:13.55	Verify-Documents (V)	Paolo (P)	50
1	01-01-2018:14.57	Check-Vacancies (C)	Frank (F)	100
2	01-01-2018:15.20	Check-Vacancies (C)	Paolo (P)	100
4	01-01-2018:15.22	Register (R)	Joey (J)	1000
2	01-01-2018:16.00	Verify-Documents (V)	Frank (F)	50
2	01-01-2018:16.10	Decision (D)	Alex (A)	500
5	01-01-2018:16.30	Register (R)	Joey (J)	1000
4	01-01-2018:16.55	Check-Vacancies (C)	Monica (M)	100
1	01-01-2018:17.57	Decision (D)	Alex (A)	500
3	01-01-2018:18.20	Check-Vacancies (C)	Joey (J)	50
3	01-01-2018:19.00	Decision (D)	Alex (A)	500
4	01-01-2018:19.20	Verify-Documents (V)	Joey (J)	50
5	01-01-2018:20.00	Special-Case (S)	Katy (K)	800
5	01-01-2018:20.10	Decision (D)	Katy (K)	500
4	01-01-2018:20.55	Decision (D)	Alex (A)	500

2.2 Notacje wykorzystywane w eksploracji procesów

Jak wspomniano, kluczowym zadaniem algorytmów eksploracji procesów jest wygenerowanie modelu procesu, którego celem jest jak najdokładniejsze odzwierciedlenie wzorców zachowań obserwowanych w dzienniku zdarzeń. W celu graficznego przedstawienia takiego modelu procesu, często wykorzystuje się jedną z trzech notacji: graf bezpośrednich podążeń (ang. Directly Follows Graph), sieć Petriego (ang. Petri Net) lub drzewo procesu (ang. Process Tree). Każda z tych notacji zostanie dokładniej omówiona w dalszej części pracy.

2.2.1 Sieć Petriego

Sieć Petriego jest to graficzna oraz matematyczna notacja wykorzystywana do modelowania oraz analizy systemów. Najpopularniejszą oraz najprostszą klasą sieci Petriego jest sieć miejsc i przejść [3]. Definicja takiej sieci przyjmuje następującą postać:

$$PN = (P, T, F, M_0) \tag{1}$$

gdzie:

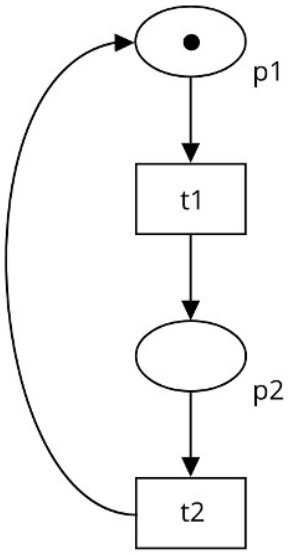
$P -$ skończony zbiór miejsc,

$T -$ skończony zbiór przejść,

$F \subseteq (P \times T) \cup (T \times P) -$ skończony zbiór łuków,

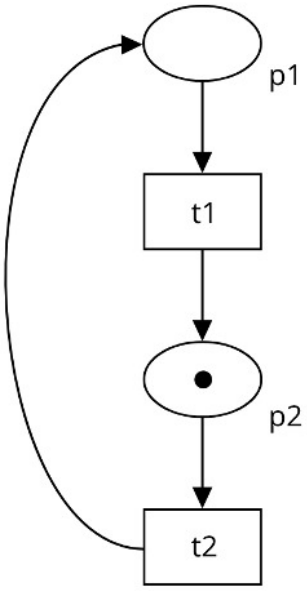
$M_0 : P \to \mathbb{N} \cup \{0\} -$ znakowanie początkowe.

W podanej definicji łuki nie posiadają przydzielonej wagi, gdyż na potrzeby eksploracji procesów wartość ta wynosi w domyśle 1. Działanie sieci odbywa się poprzez przepływ znaczników między przejściami, przy podanych założeniach: każde przejście może być uruchomione tylko jeżeli w miejscach wchodzących znajdują się znaczniki i w miejscach wychodzących znajdują się miejsca na znaczniki. Jeśli w miejscu wchodzącym nie występują znaczniki to uruchomienie przejścia jest niemożliwe [3]. Graficzną reprezentacją sieci Petriego jest dwudzielny graf skierowany.



Rys. 1: Przykładowa sieć Petriego.

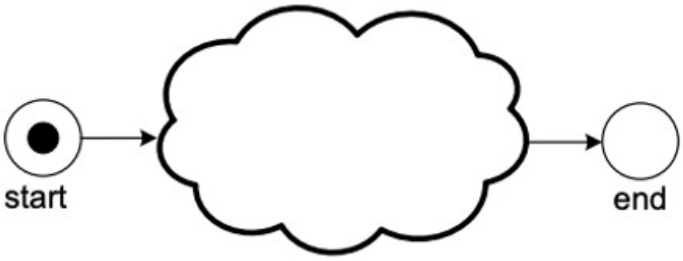
Przejścia reprezentują zdarzenia, natomiast miejsca stany - systemu. Ważnym założeniem w sieciach Petriego jest to, że dwa przejścia nigdy nie są ze sobą wprost połączone. Zawsze istnieje między nimi miejsce. Ponadto, w sieciach Petriego wygenerowanych przez algorytmy eksploracji procesów, mogą wystąpić tzw. ciche przejścia, które są oznaczone czarnym prostokątem i nie mają powiązań z żadnymi aktywnościami. Są one używane do kontroli przepływu, takiego jak oczekiwanie na dane z zewnątrz lub opóźnienie w działaniu systemu. Rys. 1. Przedstawia sieć Petriego, w której p1 oraz p2 oznaczają miejsca, natomiast t1 i t2 - przejścia. W miejscu p1 znajduje się jeden znacznik. Odpalenie przejścia spowoduje usunięcie znacznika z miejsca p1 oraz umieszczenie go w miejscu p2. Sytuacja ta została przedstawiona na rys. 2.



Rys. 2: Przykładowa sieć Petriego po odpaleniu przejścia t1.

2.2.2 Sieć Workflow

Osobliwym rodzajem sieci Petriego, występującym w kontekście eksploracji procesów, jest tzw. Sieć WF (ang. WF-net). Taka sieć ma jedno miejsce wejściowe (zwykle nazywane startem lub i) oraz jedno miejsce wyjściowe (zwykle nazywane końcem lub o), a wszystkie inne węzły znajdują się na ścieżce od wejścia do wyjścia. Taka sieć powinna być również wolna od oczywistych anomalii tj. poprawna (ang. Soundness) [4].



Rys. 3: Schemat sieci WF [4].

Sieć WF jest poprawna wtedy i tylko wtedy, gdy spełnione są następujące warunki [4]:

- Miejsca nie mogą zawierać więcej niż jednego znacznika w jednym miejscu w tym samym czasie.
- Jeśli miejsce wyjściowe posiada znacznik, wszystkie inne miejsca są puste.
- Z każdego miejsca, które jest osiągalne z miejsca wejściowego, można dotrzeć do miejsca wyjściowego.

- Dla każdego przejścia istnieje odpowiednia sekwencja uruchomień, która umożliwia odpalenie tego przejścia.

Niepoprawny model procesu może nadal być przydatny, ale takie aspekty jak precyzyjna ocena czy poszukiwanie wąskich gardeł mogą być trudne, jeśli nie niemożliwe. Dlatego w większości przypadków niepoprawny model procesu jest z góry odrzucany [5].

2.2.3 Graf bezpośrednich podążeń

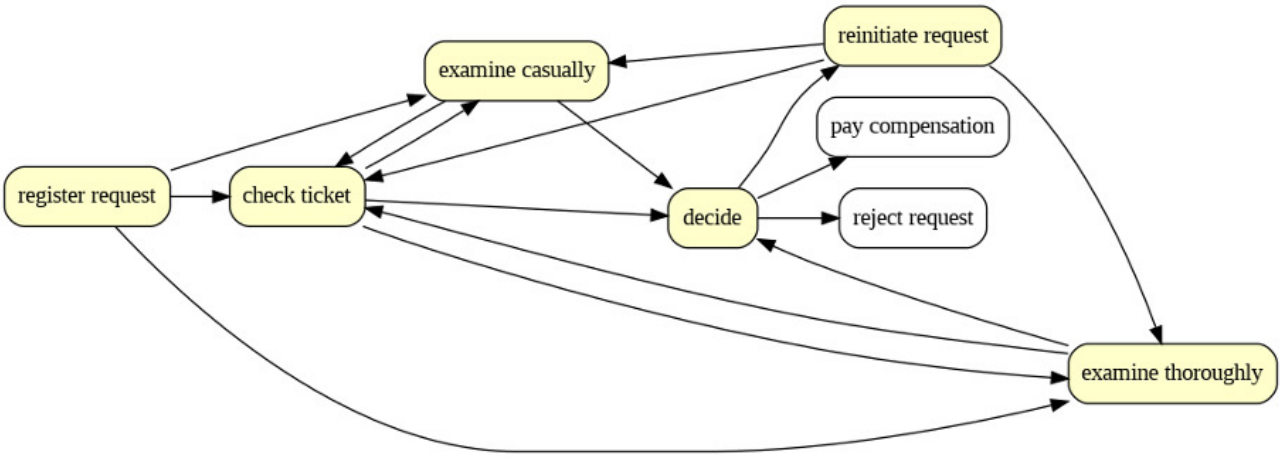
Graf bezpośrednich podążeń (ang. Directly Follows Graph) jest strukturą w postaci grafu skierowanego, wykorzystywaną w analizie procesów biznesowych. Formalna definicja grafu bezpośrednich podążeń przyjmuje następującą formę:

$$G = (V, D) \tag{2}$$

gdzie:

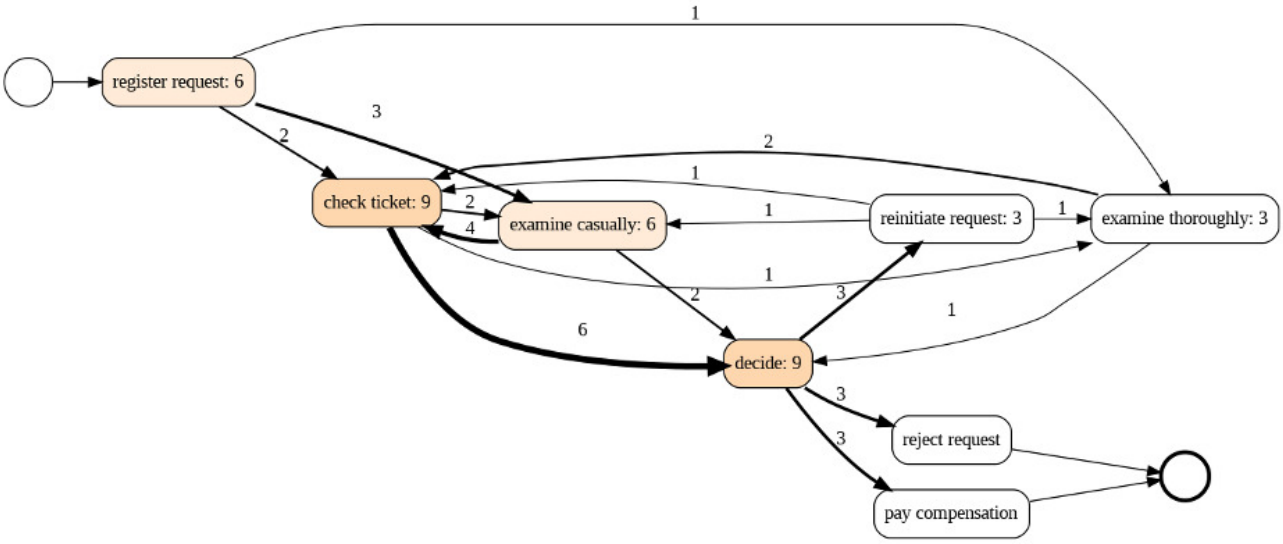
$V -$ niepusty, skończony zbiór wierzchołków reprezentujących zdarzenia,
$D -$ dowolny, skończony zbiór skierowanych krawędzi $D \subseteq P(V, V)$, reprezentowanych w postaci uporządkowanych par elementów zbioru V.

Głównym zadaniem tej struktury jest reprezentacja zależności między aktywnościami w procesie, a każda z tych aktywności jest reprezentowana przez wierzchołek grafu. Krawędzie między wierzchołkami wskazują, że jedna aktywność następuje bezpośrednio po innej. Na przykład, jeśli w procesie aktywność A jest bezpośrednio po aktywności B, to w grafie bezpośrednich podążeń istnieje krawędź skierowana od wierzchołka B do wierzchołka A. Graf bezpośrednich podążeń może pomóc w identyfikacji sekwencji aktywności, często występujących opóźnień lub innych wzorców w procesie. Może być również wykorzystywany do analizy przepływu informacji, zrozumienia struktury procesu oraz identyfikacji potencjalnych punktów optymalizacji lub problemów. Koncentruje się on jednak wyłącznie na bezpośrednich zależnościach między aktywnościami. Jest zwykle stosowany jako prosty model do wstępnej analizy procesów, ale nie uwzględnia ich pełnej złożoności i szczegółowości [6].



Rys. 4: Przykładowy graf bezpośrednich podążeń.

Aby dokładniej zobrazować proces, zdarzenia mogą różnić się kolorem w zależności od ilości ich wystąpień w dzienniku zdarzeń, a także zyskać odpowiednie etykiety w zależności od tego, jak często w logu występowało dane zdarzenie albo przepływ. Na tej podstawie, możliwe jest również przeprowadzenie filtracji rzadkich zdarzeń lub przepływów, tj. można ustalić dowolny próg, poniżej którego określone zdarzenia lub przepływy zostaną pominięte w przefiltrowanym grafie.



Rys. 5: Pokolorowany graf bezpośrednich podążeń wraz z etykietami.

Graf bezpośrednich podążeń jest notacją stosunkowo prostą do otrzymania oraz nie wymaga znajomości żadnego ze wspomnianych algorytmów eksploracji procesów. Jest jednak wykorzystywany jako forma pośrednia przez algorytm Inductive Miner, podczas generowania drzewa procesu. Modyfikacja tego algorytmu, zwana Inductive Miner Infrequent, dodatkowo stosuje filtracje według podanego przez użytkownika progu.

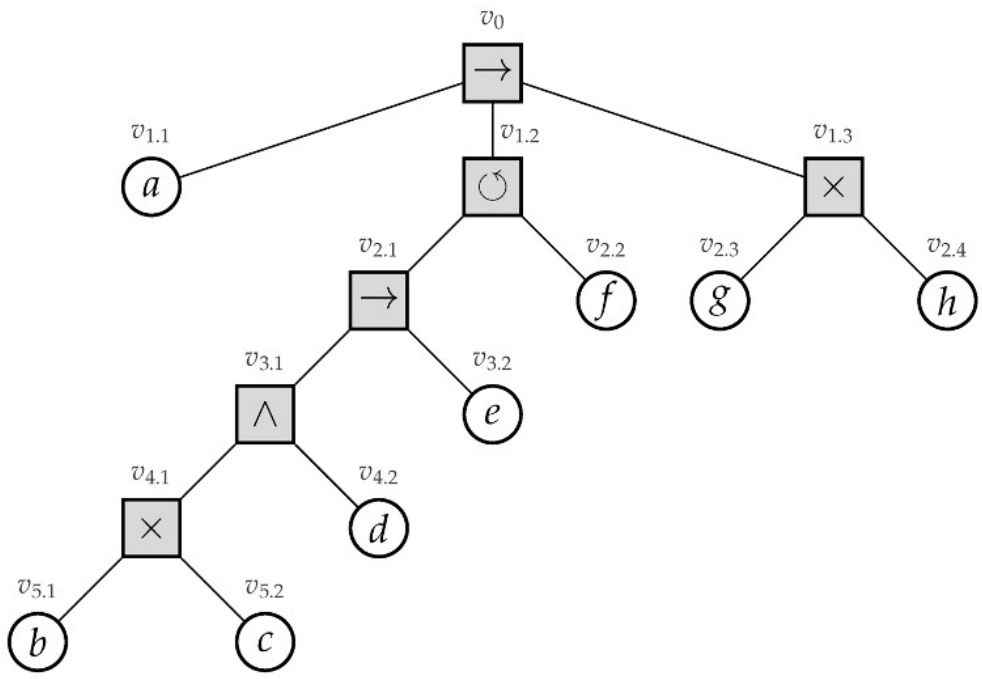
2.2.4 Drzewo procesu

Drzewo procesu jest strukturą danych zawierającą wierzchołki oraz krawędzie, gdzie krawędzie łączą wierzchołki w taki sposób, że zawsze istnieje dokładnie jedna ścieżka pomiędzy dowolnymi dwoma wierzchołkami. Struktura ta odpowiada matematycznemu pojęciu drzewa zakorzenionego, tj. nieskierowanego spójnego grafu acyklicznego [7].

Graf $G = (V, E)$, gdzie V to zbiór wierzchołków oraz E to zbiór krawędzi, jest drzewem pod warunkiem, że:

1. Dla każdego z dwóch wierzchołków w grafie, istnieje dokładnie jedna ścieżka.
2. Graf G jest acykliczny oraz $|V| = |E| + 1$.
3. Graf G jest spójny oraz $|V| = |E| + 1$.

W drzewie procesu wewnętrzne wierzchołki reprezentują relacje, a liście reprezentują czynności procesu. Przedstawia ono hierarchiczną strukturę procesu, w którym zadania i aktywności są pogrupowane według ich zależności między sobą. Drzewo procesu umożliwia wizualizację złożonej struktury procesu biznesowego i tym samym pozwala na łatwiejsze zrozumienie zależności między zdarzeniami.



Rys. 6: Przykładowe drzewo procesu wykorzystywane w eksploracji procesów [7].

Na Rys. 6. przedstawiono wszystkie etykiety występujące w drzewie procesu. Są to:

- sekwencja ($\rightarrow$),
- pętla ($\circlearrowleft$),

- wyłączny wybór ($\times$),
- równoległość ($\wedge$).

Sekwencja określa, że wszystkie podległe czynności wykonywane są od lewej do prawej strony, tzn. w pierwszej kolejności zostanie wykonana czynność znajdująca się na lewej gałęzi, następnie czynności odpowiadające gałęzi środkowej, natomiast na końcu wykonywane są czynności z gałęzi prawej. Pętla określa zachowanie cykliczne, tzn. czynność z lewej strony jest wykonywana zawsze, natomiast czynność z prawej strony wymaga powtórzenia czynności ze strony lewej. Wyłączny wybór reprezentuje bramkę XOR tzn. może być wykonana zarówno czynność pierwsza, jak i druga, jednak nigdy nie obie na raz. Równoległość odnosi się do współbieżności, tzn. czynności posiadające ten sam wierzchołek mogą być wykonywane w tym samym czasie. Dopuszczalne jest także użycie znaku (τ) oznaczające tzw. „cichą aktywność" (ang. Silent activity).

Co więcej, drzewo procesu jest odpowiednikiem poprawnej sieci WF o strukturze blokowej. Sieć WF posiada strukturę blokową wtedy, kiedy zredukowana sieć zawiera tylko jedno przejście [8]. Gdy sieć Petriego jest poprawną siecią WF o strukturze blokowej, możliwe jest przekonwertowanie jej do postaci drzewa. Sposób działania algorytmu konwertującego został opisany w dziale 4. Konwersja w drugą stronę jest zawsze możliwa.

3 Wybrane algorytmy odkrywania procesów

Istnieje wiele algorytmów i technik stosowanych w eksploracji procesów, które mogą być wykorzystywane do analizowania i modelowania procesów biznesowych na podstawie danych z dzienników zdarzeń. Mają one na celu uzyskanie wglądu w przebieg procesu, identyfikację wzorców, zależności oraz optymalizację procesów. Dla celów tej pracy opisano trzy algorytmy wraz z ich modyfikacjami. Algorytmy te nazywane są również Minerami.

3.1 Alpha Miner

Jako jeden z podstawowych algorytmów wykorzystywanych w eksploracji procesów można wskazać Alpha Miner. Na wejściu algorytm ten przyjmuje dziennik zdarzeń, a jego działanie polega identyfikacji sekwencji zdarzeń występujących w procesie. Opiera się na ośmiu krokach, które skutkują otrzymaniem pogrupowanych zdarzeń, co w konsekwencji pozwala na zbudowanie sieci Petriego [8].

W pierwszej kolejności Alpha Miner analizuje dziennik zdarzeń w poszukiwaniu określonych wzorców. Przykładowo, jeśli w logu występuje sekwencja zdarzeń (a, b), ale nigdy nie sekwencja (b, a), wówczas algorytm rozpoznaje zależność przyczynowo-skutkową między a i b. Aby odzwierciedlić tę zależność, odpowiednia sieć Petriego powinna zawierać miejsce, które łączy zdarzenie a z b [8]. Algorytm Alpha Miner wykorzystuje cztery relacje porządkujące, które opierają się na analizie logów, aby wykryć i uwzględnić odpowiednie wzorce występujące w logach. Niech L będzie dziennikiem zdarzeń oraz a i b zdarzeniami występującymi w tym dzienniku. Cztery relacje porządkujące wykorzystywane przez algorytm Alpha Miner to:

- **Relacja bezpośredniego podążania** ($>$):

$$a >_L b \quad \text{bezpośrednio po } a \text{ następuje } b$$

- **Relacja sekwencji** ($\rightarrow$):

$$a \rightarrow_L b \quad (\text{jeśli } a > b \text{ oraz nie } b >_L a)$$

- **Relacja równoległości** ($||$):

$$a||_L b \quad (\text{jeśli zarówno } a >_L b \text{ oraz } b >_L a)$$

- **Relacja braku bezpośredniego podążania** (#):

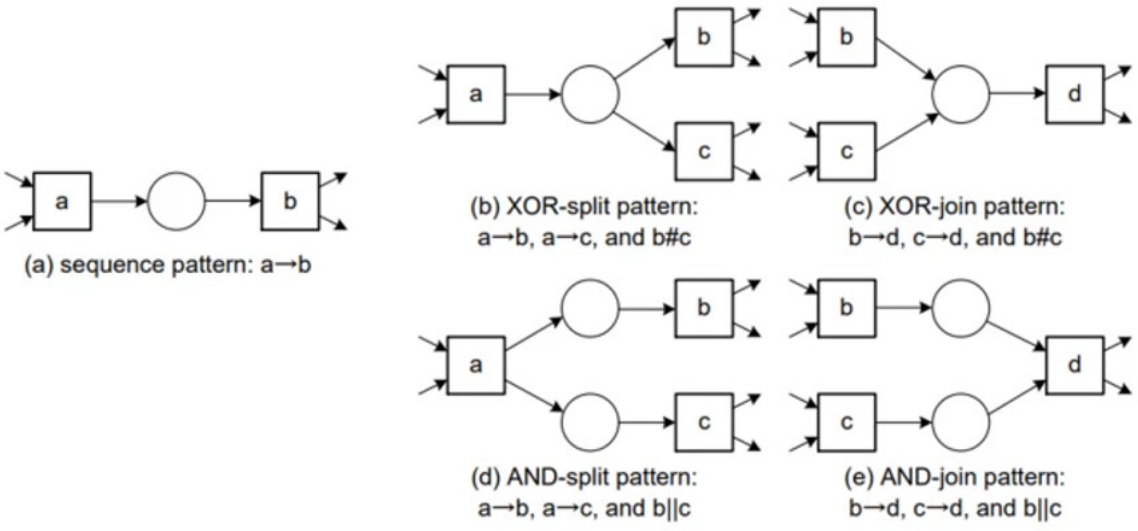
$$a\#_L b \quad (\text{jeśli zarówno nie } a >_L b \text{ ani } b >_L a)$$

Przy czym wzorzec bezpośredniego podążania służy do definiowania reszty wzorców. Dzięki znajomości relacji porządkujących oraz odpowiedniej analizie dziennika zdarzeń możliwe jest stworzenie tzw. Macierzy śladów (ang. Footprint matrix). Formalnie, macierz śladów jest symetryczną macierzą T, gdzie $T[i][j]$ reprezentuje relacje pomiędzy zdarzeniem i, a zdarzeniem j. Tabela 2. przedstawia macierz śladów T dla dziennika zdarzeń L.

Tab. 2: Macierz śladów dla dziennika zdarzeń $L = [(a, b, c, d, e, f, b, d, c, e, g), (a, b, d, c, e, g)^2, (a, b, c, d, e, f, b, c, d, e, f, b, d, c, e, g)]$ [8].

	a	*b*	*c*	*d*	*e*	*f*	*g*
a	#	→	#	#	#	#	#
b	←	#	→	→	#	←	#
c	#	←	#	‖	→	#	#
d	#	←	‖	#	→	#	#
e	#	#	←	←	#	→	→
f	#	→	#	#	←	#	#
g	#	#	#	#	←	#	#

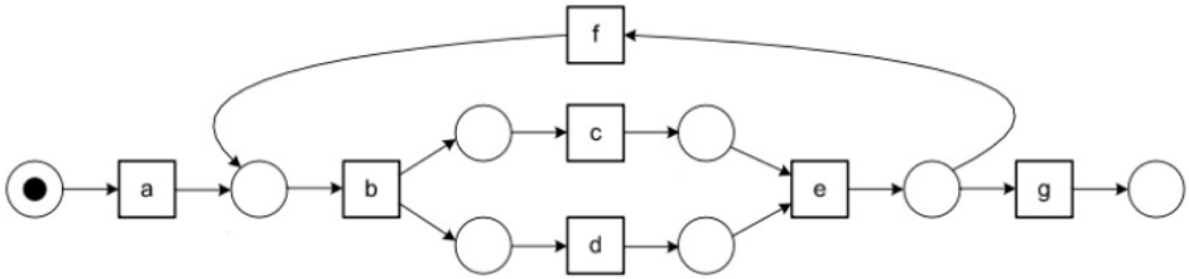
Po przeprowadzeniu analizy logów, kolejną czynnością w algorytmie Alpha Miner jest odpowiednie pogrupowanie zdarzeń z macierzy śladów. Otrzymane grupy opisują sposób następowania po sobie zdarzeń, a więc opisują charakter związku między poszczególnymi elementami procesu.



Rys. 7: Pięć wzorców w notacji sieci Petriego rozpoznawanych przez Alpha Miner [8].

Otrzymywane wzorce to: sekwencja, wzorzec XOR-split, wzorzec XOR-join, wzorzec AND-split oraz wzorzec AND-join, zostały one przedstawione na Rys. 7. Każdy ze wzorców relacji

posiada swój odpowiednik w sieci Petriego, dlatego wynik działania algorytmu zwykle prezentowany jest za pomocą tej notacji. Poniższy rysunek przedstawia sieć Petriego otrzymaną na podstawie pogrupowanych zdarzeń z macierzy śladów T, zaprezentowanej w tabeli 2.



Rys. 8: Sieć Petriego otrzymana dla dziennika zdarzeń $L = [(a, b, c, d, e, f, b, d, c, e, g), (a, b, d, c, e, g)^2, (a, b, c, d, e, f, b, c, d, e, f, b, d, c, e, g)]$ [8].

Główną limitacją algorytmu Alpha Miner jest fakt, że ogranicza się do danych sekwencyjnych, co oznacza, że może mieć trudności z analizą danych niesekwencyjnych lub danych, w których istnieją inne rodzaje zależności między zdarzeniami. Nie bierze pod uwagę częstotliwości zdarzeń, przez co może mieć trudności z dokładnym modelowaniem bardziej złożonych procesów biznesowych. Nadaje się głównie do dzienników zdarzeń bez szumów, a co najważniejsze **nie gwarantuje poprawności** modelu, niemniej jednak stanowi dobre wprowadzenie do tematu eksploracji procesów i jest punktem wyjścia dla wielu innych Minerów. Istnieją również pewne modyfikacje tego algorytmu, takie jak:

- **Alpha+** - wersja algorytmu Alpha Miner obsługująca tzw. „krótkimi pętle", czyli powtórzeniami tego samego zdarzenia w procesie [9],
- **Alpha++** - pozwala wykryć bardziej złożone wzorce, tj. wykrywa dodatkową konstrukcję zwaną niewolnym wyborem ang. (ang. Non-free choice) [10],
- **Alpha#** - poprzez rozszerzenie klasycznej wersji algorytmu umożliwia wykrywanie "niewidzialnych" zdarzeń, czyli takich które istnieją w procesie, ale nie są uwzględnione w dzienniku zdarzeń [1].

3.2 Heuristics Miner

Algorytm ten posiada kilka ulepszeń względem algorytmu Alpha Miner. Po pierwsze, bierze pod uwagę częstotliwości zdarzeń oraz znaczenie, dzięki czemu może odfiltrować rzadkie zachowania, co czyni go mniej wrażliwym na szum i niekompletne dzienniki zdarzeń [12]. Algorytm Heuristics Miner może także wykrywać krótkie pętle. Dodatkowo umożliwia pominięcie pojedynczych czynności. Wciąż nie gwarantuje jednak poprawnych modeli procesu. Podstawową ideą tego algorytmu jest fakt, że rzadkie ścieżki nie powinny być włączane do modelu.

Model procesu jest konstruowany na podstawie dwóch macierzy: macierzy częstości bezpośredniego podążania (ang. Directly-follows frequency matrix) oraz macierzy zależności (ang. Dependency matrix). Pierwsza macierz tworzona jest na podstawie dziennika zdarzeń oraz zawiera informacje o częstotliwości występowania bezpośrednio następujących po sobie zdarzeń w procesie. Każdy element macierzy odpowiada parze zdarzeń (a, b), gdzie a i b to różne zadania lub działania w procesie. Wartość w danym elemencie macierzy określa, ile razy w dzienniku zdarzeń L, zdarzenie b występuje bezpośrednio po zdarzeniu a. Przykładowa macierz bezpośredniego podążania została przedstawiona w Tabeli 3.

Tab. 3: Macierz bezpośredniego podążania dla dziennika zdarzeń $L = [(a, e)^5, (a, b, c, e)^{10}, (a, c, b, e)^{10}, (a, b, e)^1, (a, c, e)^1, (a, d, e)^{10}, (a, d, d, e)^2, (a, d, d, d, e)^1]$ [8].

$\lvert >_L \rvert$	a	b	c	d	e
a	0	11	11	13	5
b	0	0	10	0	11
c	0	10	0	0	11
d	0	0	0	4	13
e	0	0	0	0	0

Kolejna macierz, macierz zależności, reprezentuje relacje między zdarzeniami w procesie. Dla każdej pary zdarzeń (a, b) w macierzy częstości bezpośredniego następowania, macierz zależności określa, czy zdarzenie b jest zależne od zdarzenia a.

$|a \Rightarrow_L b|$ oznacza relację zależności pomiędzy zdarzeniami a oraz b oraz przyjmuje wartości z zakresu od -1 do 1.

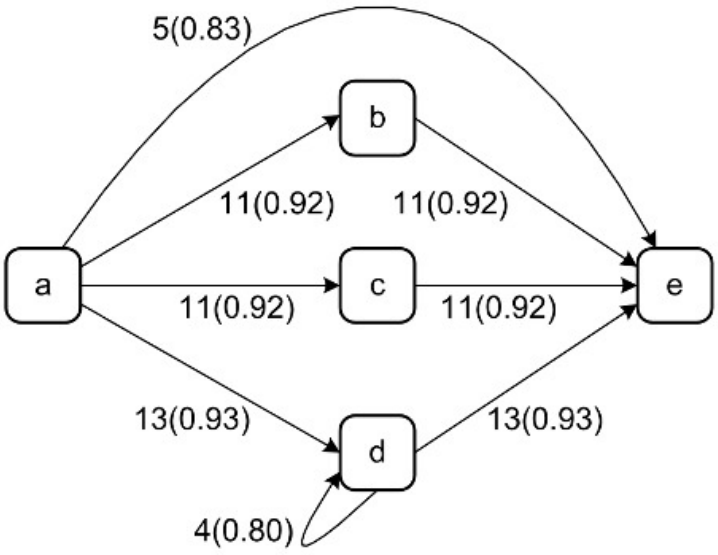
$$|a \Rightarrow_L b| = \begin{cases} \frac{|a>_L b| - |b>_L a|}{|a>_L b| + |b>_L a| + 1}, & \text{jeśli a} \neq \text{b} \\ \frac{a>_L a}{|a>_L a| + 1}, & \text{jeśli a = b} \end{cases}$$

Gdy wartość $|a \Rightarrow_L b|$ jest bliska 1, oznacza to silną pozytywną zależność między zdarzeniami a i b. Wartość ta osiągana jest wtedy, gdy a często bezpośrednio następuje po b, ale b rzadko bezpośrednio następuje po a. Jeżeli wartość $|a \Rightarrow_L b|$ jest bliska -1, oznacza to silną negatywną zależność między a i b, np. b często bezpośrednio następuje po a. Istnieje również specjalny przypadek dla $|a \Rightarrow_L a|$. Jeśli a często następuje po a, sugeruje to pętlę i silną zależność zwrotną. Ponieważ $\frac{|a>_L b|-|b>_L a|}{|a>_L b|+|b>_L a|+1} = 0$, w tym przypadku wykorzystywana jest formuła $|a \Rightarrow_L a| = \frac{a>_L a}{|a>_L a|+1}$. W Tabeli 4. przedstawiono macierz zależności dla zdarzeń z dziennika L.

Tab. 4: Macierz zależności dla dziennika zdarzeń L = [$(a,e)^5$, $(a,b,c,e)^{10}$, $(a,c,b,e)^{10}$, $(a,b,e)^1$, $(a,c,e)^1$, $(a,d,e)^{10}$, $(a,d,d,e)^2$, $(a,d,d,d,e)^1$] [8].

$\|\Rightarrow_L\|$	*a*	*b*	*c*	*d*	*e*
a	$\frac{0}{0+1}=0$	$\frac{11-0}{11+0+1}=0.92$	$\frac{11-0}{11+0+1}=0.92$	$\frac{13-0}{13+0+1}=0.93$	$\frac{5-0}{5+0+1}=0.83$
b	$\frac{0-11}{0+11+1}=-0.92$	$\frac{0}{0+1}=0$	$\frac{10-10}{10+10+1}=0$	$\frac{0-0}{0+0+1}=0$	$\frac{11-0}{11+0+1}=0.92$
c	$\frac{0-11}{0+11+1}=-0.92$	$\frac{10-10}{10+10+1}=0$	$\frac{0}{0+1}=0$	$\frac{0-0}{0+0+1}=0$	$\frac{11-0}{11+0+1}=0.92$
d	$\frac{0-13}{0+13+1}=-0.93$	$\frac{0-0}{0+0+1}=0$	$\frac{0-0}{0+0+1}=0$	$\frac{4}{4+1}=0.80$	$\frac{13-0}{13+0+1}=0.93$
e	$\frac{0-5}{0+5+1}=-0.83$	$\frac{0-11}{0+11+1}=-0.92$	$\frac{0-11}{0+11+1}=-0.92$	$\frac{0-13}{0+13+1}=-0.93$	$\frac{0}{0+1}=0$

Na podstawie Tabeli 3 oraz Tabeli 4. możliwe jest wyprowadzenie tzw. grafu zależności. W grafie tym pokazane są tylko takie łuki, które spełniają określone progi. Dla danego dziennika zdarzeń możliwe jest wygenerowanie różnych modeli w zależności od wyznaczonego progu. Graf zależności przedstawiony na Rys. 9. wykorzystuje próg 2 dla $|>_L|$ oraz 0.7 dla $|\Rightarrow_L|$, czyli łuk np. między a i b jest uwzględniony tylko wtedy, gdy $|a >_L b| \geq 2$ oraz $|a \Rightarrow_L b| \geq 0.7$.



Rys. 9: Macierz zależności dla dziennika zdarzeń L = [(a,b,c,d,e,f,b,d,c,e,g), $(a,b,d,c,e,g)^2$, $(a,b,c,d,e,f,b,c,d,e,f,b,d,c,e,g)$] przy $|>_L| \geq 2$ oraz $|\Rightarrow_L| \geq 0.7$ [8].

3.3 Inductive Miner

Wynikiem algorytmu Inductive Miner jest blokowy model procesu w postaci drzewa. W podstawowej wersji składa się on z trzech rekurencyjnych etapów:

- Budowanie grafu bezpośrednich połączeń,
- Poszukiwania cięć,
- Podział dziennika.

W trakcie działania Inductive Minera tworzony jest pomocniczy model w postaci grafu bezpośrednich podążeń, na podstawie którego przeprowadza się operację cięcia w dzienniku zdarzeń. Jeśli detekcja cięcia zakończy się sukcesem (tj. jeśli zostanie znalezione cięcie), algorytm zwraca operator cięcia oraz jego podział. Na tej podstawie algorytm dzieli dziennik zdarzeń na jeden lub więcej poddzienników. Algorytm jest następnie rekurencyjnie stosowany do każdego poddziennika, tj. dla każdego poddziennika budowany jest graf bezpośrednich podążeń, wyszukiwane jest cięcie, a dziennik zostaje podzielony zgodnie ze znalezionym cięciem. Rekurencja kończy się w momencie wykrycia pojedynczej aktywności. Każde cięcie odbywa się poprzez wykrycie jednego ze wzorców:

- **Sekwencji**, gdzie $(\rightarrow, A_1, \ldots, A_n)$ jest cięciem sekwencji wtedy i tylko wtedy, gdy:

$$\forall i, j \in \{1, .., n\}(\forall a \in A_i, b \in A_j(a \rightarrow_L b \wedge \neg(b \rightarrow_L a)))$$

- **Równoległości**, gdzie $(\wedge, A_1, \ldots, A_n)$ jest cięciem równoległości wtedy i tylko wtedy, gdy:

$$-\forall i \in \{1, .., n\}(A_i \cap A_L^s \neq \emptyset \wedge A_i \cap A_L^e \neq \emptyset)$$

$$-\forall i, j \in \{1, .., n\}(\forall a \in A_i, b \in A_j(i \neq j \Rightarrow a \rightarrow_L b))$$

- **Pętli**, gdzie $(\circlearrowleft, A_1, \ldots, A_n)$ jest cięciem pętli wtedy i tylko wtedy, gdy:

$$-n \geq 2$$

$$-A_1 \supseteq A_L^s \cup A_L^e$$

$$-\forall i, j \in \{2, .., n\}(\forall a \in A_j, (i \neq j \Rightarrow a \rightarrow_L b)$$

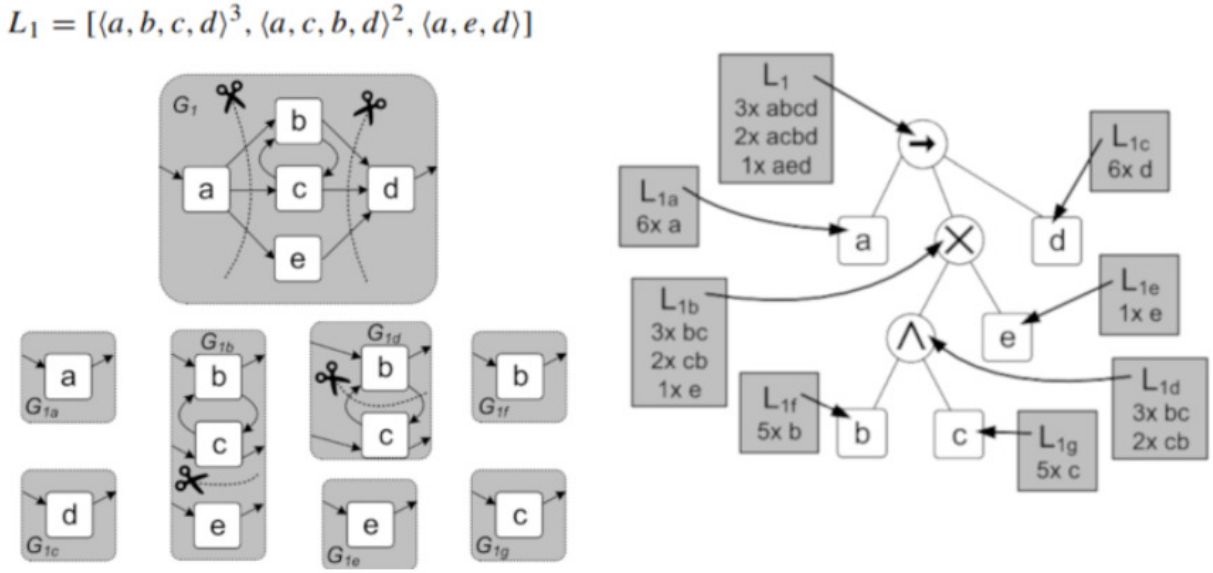
$$-\forall i \in \{2, .., n\}(\forall b \in A_i, (\exists a \in A_L^e(a \rightarrow_L b)) \Rightarrow (\forall a \in A_L^e(a \rightarrow_L b)))$$

$$-\forall i \in \{2, .., n\}(\forall b \in A_i, (\exists a \in A_L^s(b \rightarrow_L a)) \Rightarrow (\forall a \in A_L^s(b \rightarrow_L a)))$$

- **Wyłącznego wyboru**, gdzie $(\times, A_1, \ldots, A_n)$ jest cięciem wyłącznego wyboru wtedy i tylko wtedy, gdy:

$$\forall i, j \in \{1, .., n\}(\forall a \in A_i, b \in A_j(i \neq j \Rightarrow \neg(b \rightarrow_L a)))$$

Po rozpoznaniu wzorca, jego symbol jest dodawany jako wierzchołek drzewa, a poddzienniki powstałe w wyniku cięć stają się bazą dla kolejnych wierzchołków.



Rys. 10: Drzewo procesu uzyskane na podstawie cięć wykonanych na Logu L1 [13].

Niewątpliwą zaletą wykorzystania algorytmu Inductive Miner jest to, że wszystkie odkryte modele odpowiadają poprawnym siecią WF o strukturze blokowej, a model jest zawsze dopasowany, tj. może generować ślady w dzienniku [14]. W związku z tym, zawsze możliwe jest przedstawienie modelu w postaci sieci WF. W praktyce, w dziennikach zdarzeń zwykle występują odstępstwa lub rzadkie zachowania, które nie są istotne dla ostatecznego modelu. W takich przypadkach korzysta się z modyfikacji algorytmu, aby pomóc w odfiltrowaniu tych nieistotnych elementów. Do zmodyfikowanych algorytmów należą:

- **Inductive Miner Infrequent ($\mathbf{IM}_F$)** - bierze pod uwagę częstotliwość zdarzeń oraz radzi sobie z rzadkimi przypadkami poprzez dodanie parametrów odpowiadających za odfiltrowanie anomalii.
- **Inductive Miner Incompleteness ($\mathbf{IM}_C$)** - bierze pod uwagę niekompletne przebiegi procesu poprzez dodanie relacji pomiędzy zdarzeniami.
- **Inductive Miner All operators ($\mathbf{IM}_A$)** - dodaje takie operatory relacji jak: przeplatanie ($\leftrightarrow$) czy włączenie ($\vee$).

Inductive Miner Infrequent

Podobnie jak w przypadku algorytmu Inductive Miner, w zmodyfikowanej wersji tego algorytmu, drzewo procesu jest budowane w sposób rekurencyjny. Działanie wersji Infrequent również rozpoczyna się od odnalezienia grafu bezpośredniego podążenia oraz przeprowadzenia detekcji cięcia. Funkcje używane w tym etapie są takie same jak te używane w podstawowej wersji algorytmu. Jeśli cięcie zostanie znalezione, dziennik zostaje podzielony, a rekurencja stosowana jest w wynikowych poddziennikach. Jeśli cięcie nie zostanie znalezione, graf bezpośrednich podążeń zostaje przefiltrowany na podstawie parametru progowego f. W przeciwieństwie do podstawowej wersji algorytmu, wersja Infrequent nie przyjmuje tylko dziennika zdarzeń jako wejścia, ale również wspomniany parametr f. Po przeprowadzonej filtracji ponownie stosowana jest detekcja cięcia. Jeśli cięcie zostanie znalezione, dziennik zostaje podzielony, a rekurencja kontynuuje się na wynikowych poddziennikach. Jeśli cięcie nie zostanie znalezione, stosowane jest tzw. podejście fall-through.

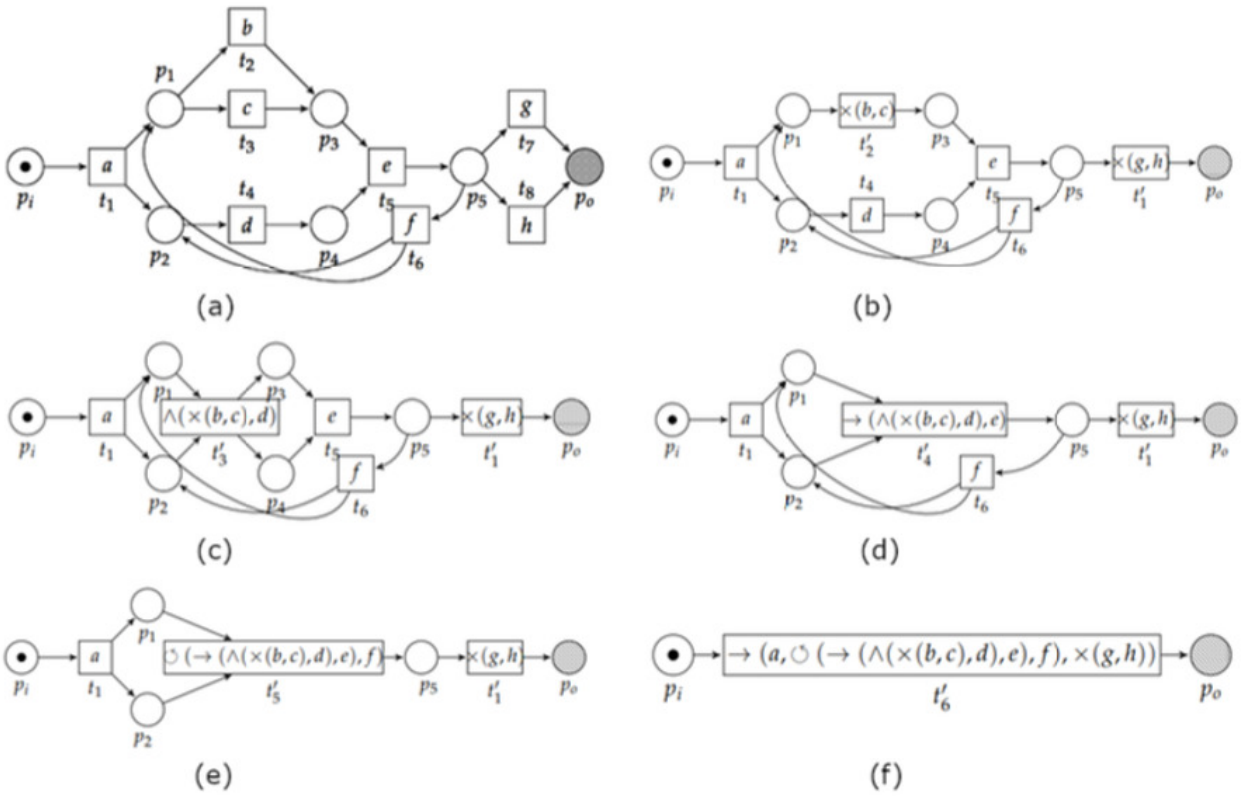
4 Translacja sieci Petriego na drzewo procesu

Aby przekonwertować sieć Petriego analizowanego procesu na drzewo procesu, konieczne jest spełnienie trzech warunków:

1. Podana sieć Petriego musi być siecią Workflow,
2. Sieć musi być poprawna,
3. Sieć musi posiadać strukturę blokową.

W praktyce oznacza to, że nie wszystkie odkryte sieci procesu w postaci sieci Petriego mogą być reprezentowane jako drzewa procesu. Aby osiągnąć zgodność między językiem wyrażanym przez sieć procesu a językiem wyrażanym przez drzewo, konieczne jest dokładne zdefiniowanie wzorców wykrywanych w sieci. W skrócie, zbiory generowanych ścieżek przez oba modele muszą być identyczne. Posiadając model w postaci sieci WF, po dokonaniu redukcji, możliwe jest przedstawienie procesu w formie drzewa wtedy i tylko wtedy, gdy zredukowana sieć zawiera tylko jedno przejście. Jest to równoznaczne ze stwierdzeniem, że podana sieć posiada strukturę blokową. Ponadto sieć, którą chcemy przekonwertować, musi być poprawna. Innymi słowy oznacza to, że zawsze możliwe jest dotarcie ze stanu początkowego do stanu końcowego bez napotkania błędów. Podana sieć nie może zawierać martwych przejść, a więc wszystkie przejścia muszą posiadać możliwość aktywacji. Sieć musi posiadać możliwość ukończenia oraz prawidłowe zakończenie, a więc z każdego miejsca, do którego da się dotrzeć z miejsca początkowego, musi się również dać dotrzeć do miejsca końcowego oraz gdy w miejscu początkowym znajduję się znacznik, w żadnym innym miejscu nie może się już znajdować inny znacznik.

Istnieje algorytm, który tłumaczy sieć WF na drzewo procesów. Algorytm wykrywa, czy sieć ta odpowiada drzewu procesów, a jeśli tak, konstruuje je. Przejście z sieci na drzewo polega na stopniowej redukcji sieci na coraz mniejsze zachowania dające się wyrazić w notacji drzewa. Po odnalezieniu części, która może zostać zredukowana, zostaje ona zamieniona na mniejszy fragment reprezentujący drzewo procesu. Jeżeli przetwarzana w ten sposób sieć sprowadza się na końcu do pojedynczego przejścia, translacja sieci WF na drzewo procesu jest możliwa [7].



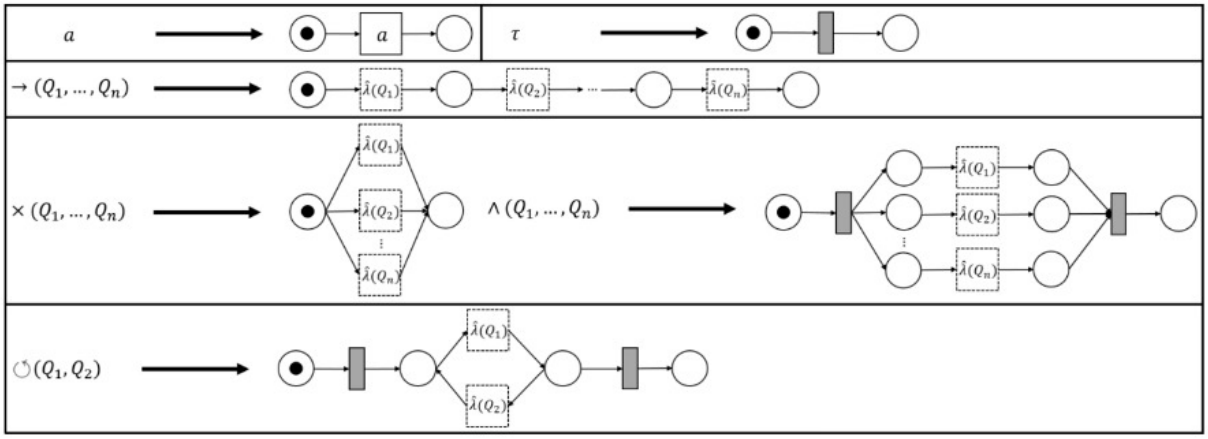
Rys. 11: Zasada działania algorytmu tłumaczącego sieć WF na drzewo procesów [7].

Powyższy rysunek przedstawia zasadę działania algorytmu tłumaczącego sieć WF na drzewo procesów, na jego potrzeby tworzona jest struktura pośrednia, która łączy cechy zarówno sieci WF, jak i drzewa. Struktura ta charakteryzuje się połączeniem kilku aktywności wraz z ich relacjami. Przedstawiona konwersja sieci do drzewa składa się z kilku etapów:

1. Wybrane zostają dwie struktury, dla przykładu: jedna pomiędzy przejściem b i c oraz druga między przejściem g i h.
2. Gdy znaleziony zostaje odpowiedni wierzchołek drzewa, charakteryzujący zależności między czynnościami wewnątrz wybranej struktury, algorytm zastępuje je za pomocą dwóch nowych przejść, na rys. 11b oznoczanych jako $x(b, c)$, $x(g, h)$.
3. Dobierana jest następna para, np. $x(b, c)$ oraz d, która zostaje zastąpiona poprzez nowe przejście na rys. 11c oznaczone jako $\wedge(\times(b, c), d)$.
4. Para $\wedge(\times(b, c), d)$ oraz e zostaje zastąpiona przez przejście $\rightarrow (\wedge(\times(b, c), d), e)$.
5. Para $\rightarrow (\wedge(\times(b, c), d), e)$ oraz f zostaje zastąpiona przez przejście $\circlearrowleft (\rightarrow (\wedge(\times(b, c), d)e), f)$.
6. Rezultatem jest drzewo procesu postaci $\rightarrow (a, \circlearrowleft (\rightarrow (\wedge(\times(b, c), d)e), f) \times (g, h)))$.

W przypadku konwersji odwrotnej, tj. translacji drzewa procesu na sieć Petriego, konieczne jest zdefiniowanie wzorców, które będą odzwierciedleniem wierzchołków drzewa procesu w sieci Petriego. Można bezpośrednio określić pojedyncze działania lub ich brak, natomiast dla innych wzorców, takich jak sekwencja, wybór wyłączny, równoległość czy pętla, konieczne jest zastosowanie rekurencji, aż do rozłożenia poszczególnych wzorców na pojedyncze aktywności. Tabela 5. przedstawia odpowiedniki wzorców z drzewa procesu w sieci Petriego.

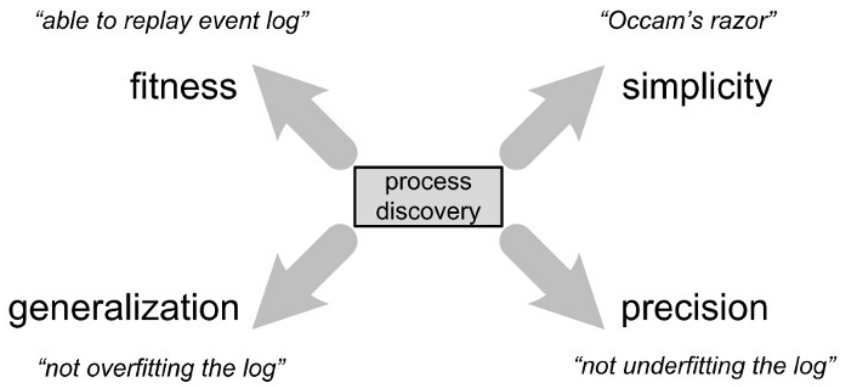
Tab. 5: Odzwierciedlenie wzorców drzewa procesu w sieci Petriego [7].



5 Miary jakości modelu procesu

W procesie oceny jakości wytworzonego modelu, brane pod uwagę są cztery kryteria określające pożądane zachowania modelu. Kryteria te bezpośrednio odnoszą się do sposobu, w jaki model odzwierciedla zachowania obecne w dzienniku zdarzeń. Niestety, kryteria te są ze sobą sprzeczne (Rys. 12), co powoduje konieczność ustalenia między nimi kompromisu. Cztery kryteria jakości modelu to:

- **Trafność** (ang. fitness) - odkryty model powinien uwzględniać zachowanie widoczne w dzienniku zdarzeń.
- **Precyzja** (ang. precision) - odkryty model nie powinien pozwalać na zachowanie zupełnie niezwiązane z tym, co zaobserwowano w dzienniku zdarzeń.
- **Generalizacja** (ang. generalization) - wykryty model powinien uogólnić przykładowe zachowanie widoczne w dzienniku zdarzeń.
- **Prostota** (ang. simplicity) - wykryty model powinien uogólnić przykładowe zachowanie widoczne w dzienniku zdarzeń.



Rys. 12: Balans kryteriów jakości [8].

Model o wysokiej trafności potrafi odzwierciedlić większość śladów obecnych w dzienniku zdarzeń. Precyzja jest związana z pojęciem niedopasowania. Model o niskiej precyzji jest niedokładny i dopuszcza zachowania, które znacznie różnią się od tego, co zostało zaobserwowane w dzienniku zdarzeń. Z kolei generalizacja odnosi się do nadmiernego dopasowania. Model z nadmiernym dopasowaniem nie jest wystarczająco uogólniony, jest zbyt szczegółowy i oparty na konkretnych przykładach z dziennika zdarzeń. Czwarte kryterium jakości to prostota. Sugeruje ona, że w modelu nie powinny występować zbędne elementy, aby wyjaśnić jakiekolwiek zjawiska. Konieczne jest więc znalezienie najprostszego modelu procesu, który potrafi wyjaśnić obserwowane zdarzenia w dzienniku [8].

Znalezienie równowagi między tymi czterema kryteriami jakości może okazać się kłopotliwe. Na przykład, zbyt uproszczony model może być mało użyteczny lub niedokładny. Chociaż wszystkie cztery wymiary są istotne, wymiar trafności wskazujący w jakim stopniu model może odtworzyć ślady w kodzie, jest ważniejszy od pozostałych wymiarów [15]. W eksploracji procesów stosowane są różne algorytmy, jednak większość z nich nie zwraca dobrej jakości metryk we wszystkich przypadkach. Według [5] algorytm Inductive Miner jest szeroko stosowany z bardzo obiecującymi wynikami. Tabela 6. przedstawia porównanie kilku algorytmów w kontekście wymienionych czterech wymiarów jakości. Inductive Miner pozwala na otrzymanie wysokiej trafności w stosunku do algorytmów Alpha Miner i Heuristics Miner. Jest w stanie poradzić sobie z rzadkimi zachowaniami i dużymi dziennikami zdarzeń, zapewniając jednocześnie poprawność [16].

Tab. 6: Porównanie algorytmów eksploracji procesów według kryteriów jakości [17].

Table 5
Comparison of algorithms based on *fitness*, *precision*, *generalization*, *simplicity*, and *overall* in sub-file 4

Algorithm	Cluster	*Fitness*	*Precision*	*Generalization*	*Simplicity*	*Overall*
Alpha Miner	Fail	0.765	0.197	0.422	0.636	0.570
Heuristic Miner	Fail	0.491	0.521	0.487	0.653	0.509
ET Miner	Fail	0.684	0.709	0.873	0.913	0.716
Inductive Miner	Fail	0.96	0.322	0.957	0.882	**0.768**
Alpha Miner	Pass	0.863	0.164	0.464	0.666	0.622
Heuristic Miner	Pass	0.526	0.707	0.603	0.732	0.596
ET Miner	Pass	0.712	0.691	0.841	0.901	0.725
Inductive Miner	Pass	0.959	0.315	0.962	0.882	**0.765**
Alpha Miner	All	0.581	0.198	0.414	0.466	0.452
Heuristic Miner	All	0.472	0.868	0.483	0.611	0.597
ET Miner	All	0.693	0.715	0.719	0.923	0.715
Inductive Miner	All	0.87	0.443	0.867	0.909	**0.747**

W badaniu wykorzystano plik z dziennikiem dostarczony przez Learning Management System (LMS) jako główne źródło danych. Plik ten zawierał zapis aktywności każdego ucznia podczas interakcji z LMS. Przed analizą dane zostały poddane wstępnemu przetwarzaniu i filtrowaniu w celu usunięcia zbędnych informacji oraz zachowania anonimowości uczniów. W kontekście analizy, każdy uczestnik był traktowany jako ”przypadek”, a każdy wiersz w przetworzonym dzienniku zdarzeń reprezentował klasę zdarzeń, które zostały wykonane przez konkretnego ucznia w określonym dniu. Dodatkowo, oceny końcowe uczniów zostały przekształcone na dane kategoryczne (Zdany/Niezdany). Ostatecznie dane dziennika zostały podzielone na trzy pliki: ”All” (zawierający zdarzenia wszystkich uczniów), ”Pass” (zdarzenia studentów, którzy zaliczyli kurs) oraz ”Fail” (zdarzenia studentów, którzy nie zaliczyli kursu). Taki podział okazał się przydatny do porównywania wydajności różnych algorytmów oraz do oceny praktycznego zastosowania i wartości teoretycznej wynikowych modeli [17].

6 Specyfikacja logiczna

Jedną z postaci formalnej specyfikacji systemu informatycznego może być jego specyfikacja logiczna. W eksploracji procesów, wykorzystanie specyfikacji logicznej może mieć wiele zalet oraz przyczynić się do usprawnienia procesów biznesowych. Jedną z głównych zalet specyfikacji logicznej jest jej jednoznaczność dzięki czemu analiza uzyskanych danych może okazać się bardziej wiarygodna i dokładna. Specyfikacja logiczna pozwala na zdefiniowanie reguł, jakie muszą być spełnione przez proces, aby był on uznany za poprawny. Tworzony system zakłada otrzymywanie specyfikacji logicznej w postaci logiki temporalnej, która w kontekście zastosowanego algorytmu generującego specyfikację logiczną, może zostać w pewnym uproszczeniu sprowadzona do rachunku pierwszego rzędu. Model procesu w postaci drzewa, dzięki swojej strukturze hierarchicznej i dobrze zdefiniowanym wzorcom, stanowi dobre wejście dla algorytmu generującego specyfikację. W tej części pracy omówione zostaną zagadnienia dotyczące specyfikacji logicznej oraz systemów dowodzenia twierdzeń.

6.1 Logika pierwszego rzędu

Logika pierwszego rzędu to taki system logiczny, który zajmuje się opisem relacji między obiektami oraz operacjami na tych relacjach. Podstawowe elementy logiki pierwszego rzędu to:

- Predykaty, przy czym predykat zero argumentowy nazywany jest zdaniem elementarnym. Predykaty są wyrażeniami zawierającymi zmienne oraz opisujące pewne relacje lub cechy obiektów.
- Kwantyfikatory, takie jak kwantyfikator ogólny ($\forall$ - ”dla wszystkich”) oraz kwantyfikator egzystencjalny ($\exists$ - ”istnieje”), które pozwalają na wyrażanie twierdzeń o wszystkich elementach zbioru lub o istnieniu przynajmniej jednego elementu spełniającego pewne warunki.
- Operatory logiczne, takie jak: implikacja ($\rightarrow$), równoważność ($\Leftrightarrow$), koniunkcja ($\wedge$), alternatywa ($\vee$) czy negacja ($\neg$).
- Aksjomaty i dowody pozwalające na tworzenie formalnych aksjomatów oraz dowodów matematycznych.

Atom to podstawowa jednostka logiczna, reprezentująca zdanie logiczne. W niniejszej pracy zero bądź jednoargumentowy predykat. Literał to atom bądź jego negacja, a klauzula to alternatywa literałów, zawierająca co najmniej jeden kwantyfikator. Formuła to koniunkcja dowolnej liczby klauzul, natomiast specyfikacją nazwiemy zbiór formuł, charakteryzujący dany system.

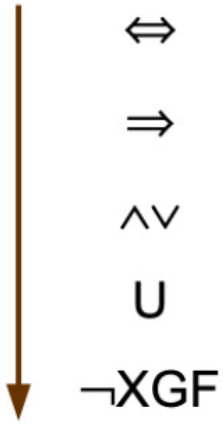
6.2 Logika temporalna

Logika temporalna to rodzaj logiki, który pozwala na formalny opis oraz analizę czasowo-zależnych systemów bez wprowadzania czasu wprost. W tym celu wykorzystuje się symbole i reguły, które pozwalają na opisywanie czasowych relacji pomiędzy różnymi zdarzeniami w systemie. Występuje więc tutaj rozszerzenie logiki pierwszego rzędu o symbole określające upływ czasu. W większości przypadków, logiki temporalne zakładają czas składający się z dyskretnych wydarzeń. Istnieją jednak również takie, które korzystają z ciągłego czasu. Do najprostszego rodzaju logik temporalnych należy logika LTL (ang. Linear Temporal Logic) operująca na dyskretnym liniowym modelu czasu. Logika CTL (ang. Computation Tree Logic) rozszerza logikę LTL o rozgałęziające się modele czasu, dodając kolejne operatory [18].

6.2.1 Składnia logiki temporalnej

Logika LTL jest systemem logiki używanym do opisu i rozumienia właściwości czasowych i sekwencyjnych systemów. W logice LTL, formuły są budowane za pomocą atomów, operatorów logicznych oraz operatorów temporalnych [19]. Atomami są stwierdzenia, które mogą być prawdziwe lub fałszywe w danym stanie systemu. Operatory logiczne, takie jak implikacja ($\rightarrow$), równoważność ($\Leftrightarrow$), koniunkcja ($\wedge$), alternatywa ($\vee$) i negacja ($\neg$), pozwalają na konstruowanie złożonych formuł. Natomiast operatory temporalne, takie jak ”X” (następnie), ”G” (zawsze), ”F” (kiedyś) i ”U” (dopóki), pozwalają na wyrażanie zależności czasowych oraz przypisują wyrażeniom wartość prawda/fałsz w strukturze czasowej.

W logice LTL zamiast X, G i F zwykle korzysta się z symboli: $\circ$, $\Box$, $\Diamond$. Kolejność operatorów i spójników określona jest wg. siły sklejania.



Rys. 13: Kolejność operatorów i spójników w logice LTL (siła sklejania wzrasta od góry do dołu) [19].

Przykładowa formuła w logice LTL może wyglądać następująco:

G(Fa): ”a jest prawdziwe dla każdego możliwego momentu w przyszłości.”

Logika LTL znajduje zastosowanie w wielu dziedzinach, takich jak weryfikacja formalna systemów, modelowanie i weryfikacja protokołów komunikacyjnych, systemy wbudowane, itp. Za pomocą tej logiki można wyrażać i analizować różne aspekty zachowania systemów, takie jak bezpieczeństwo, żywotność czy wymagania czasowe.

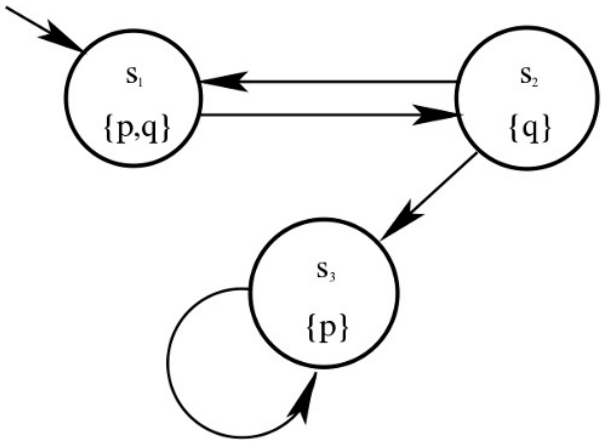
6.2.2 Struktura Kripkiego

Struktura Kripkiego jest stosowana podczas oceny modelu w celu reprezentacji zachowania systemu. Jest to graf, w którym węzły reprezentują dostępne stany systemu, a krawędzie symbolizują zmiany między tymi stanami. Kolejnym elementem jest funkcja etykietująca, która przypisuje każdemu węzłowi zestaw cech, które są obecne w odpowiadającym stanie. Jest również nazywana modelem relacyjnym i może być zdefiniowana jako struktura postaci [20]:

$$M = (S, s_0, R, L) \tag{3}$$

gdzie:

- S – to niepusty zbiór stanów,
- s_0 – jest stanem początkowym i $s_0 \in S$,
- $R : S \rightarrow S$ jest deterministycznym przejściem dla stanów, tj. $R \subseteq S \times S$ i pozwala wyznaczyć następcę danego stanu,
- $L : S \rightarrow 2^{WA}$ to funkcja etykietująca pozwalająca na przypisanie wyrażeń atomowych WA każdemu ze stanów, które są prawdziwe w tym stanie.



Rys. 14: Przykładowa struktura Kripkiego, gdzie $S = \{s1, s2, s3\}$, $S_0 = \{s1\}$, $R = \{(s1, s2), (s2, s1), (s2, s3), (s3, s3)\}$, $L = \{(s1, \{p, q\}), (s2, \{q\}), (s3, \{p\})\}$ [21].

6.3 Przemienność operatorów i kwantyfikatorów

Na potrzeby tej pracy wystarczającym było ograniczenie jedynie do dwóch symboli logiki temporalnej $\Box$ oraz $\Diamond$. Jak już wspomniano, symbole te są używane do wyrażania pewnych właściwości czasowych zdania logicznego. $\Box p$ oznacza, że zdanie p jest prawdziwe przez cały czas, natomiast $\Diamond p$ oznacza, że zdanie p jest prawdziwe w pewnym momencie w przyszłości. Choć są one powiązane z kwestią czasu i przedstawianiem warunków w różnych punktach czasowych, można zastąpić je odpowiednio kwantyfikatorem ogólnym i egzystencjalnym oraz poprzez użycie atomów jednoargumentowych zamiast zero argumentowych.

- Rozważając zdanie logiczne $\Box p$, można je zinterpretować za pomocą kwantyfikatora ogólnego „$\forall$” w taki sposób, że:

  $$\forall t : p(t)$$

  Innymi słowy, zdanie p jest prawdziwe dla każdego punktu czasowego t.

- Zdanie logiczne $\Diamond p$ można zinterpretować za pomocą kwantyfikatora egzystencjalnego ($\exists$) w taki sposób, że:

  $$\exists t : p(t)$$

  Innymi słowy, istnieje punkt czasowy t, w którym zdanie p jest prawdziwe.

Jak pokazano, logikę temporalną można w pewnym stopniu zinterpretować przy użyciu kwantyfikatorów ogólnych i egzystencjalnych, co umożliwia użycie klasycznych metod logiki do analizy aspektów temporalnych. Podejście to zostało wykorzystane w tej pracy.

7 Systemy dowodzenia twierdzeń

Systemy dowodzenia twierdzeń (ang. Automated theorem provers), bądź w skrócie Provery, to systemy tworzone w celu automatyzacji procesu dowodzenia twierdzeń logicznych. Twierdzenia te są zazwyczaj wyrażane w formie logicznej i opisują relacje między różnymi zdaniami czy elementami w danej teorii. Głównym zadaniem tych systemów jest potwierdzanie lub obalanie zadanych twierdzeń, zazwyczaj za pomocą dedukcji lub dowodzenia formalnego. Dzięki automatyzacji tego procesu, nawet przy skomplikowanych twierdzeniach, dowód staje się bardziej efektywny. Ponieważ zaprzeczenie twierdzenia jest spełnialne wtedy i tylko wtedy, gdy twierdzenie to nie jest tautologią. Provery starają się dowieść poprawność danej formuły poprzez wykazanie sprzeczności jej negacji. Zawsze istnieje możliwość rozstrzygnięcia wszystkich twierdzeń rachunku zdań, chociażby poprzez metodę polegającą na sprawdzeniu wszystkich kombinacji prawda/fałsz dla n zmiennych zdaniowych zawartych w twierdzeniu. W praktyce jednak stosowane są bardziej optymalne metody.

7.1 Problem spełnialności

Problem spełnialności jest jednym z kluczowych problemów w dziedzinie teorii obliczeń ze względu na swoje znaczenie w kontekście rozwiązywania innych problemów obliczeniowych. Należy on do klasy problemów NP-zupełnych, czyli takich, dla których znalezienie rozwiązania nie jest możliwe w czasie wielomianowym. Niech F będzie formułą logiczną w postaci koniunkcji klauzul:

$$F = C_1 \wedge C_2 \wedge ... \wedge C_k \tag{4}$$

gdzie każda klauzula C_i to alternatywa literałów, a każdy literał to zmienna logiczna lub jej negacja. Problem ten polega na ustaleniu, czy istnieje taki wektor przypisań wartości zmiennym logicznym, który sprawi, że formuła stanie się prawdziwa.

7.2 Prover Vampire

Prover Vampire [22] to zaawansowany system automatycznego dowodzenia twierdzeń, oparty na logice pierwszego rzędu. Został opracowany na Wydziale Informatyki Uniwersytetu w Manchesterze i od tego czasu jest rozwijany przez międzynarodową społeczność. Jądro systemu implementuje obliczenia w oparciu o rezolucję binarną i superpozycje. Do realizacji wszystkich głównych operacji wykorzystuje szereg wydajnych technik indeksowania. Chociaż jądro systemu operuje tylko na postaci Clause Normal Form (CNF), Vampire akceptuje również problemy w pełnej składni logiki pierwszego rzędu, konwertując je na formę klauzul i przeprowadzając kilka przekształceń, zanim przekaże wynik do jądra. Gdy twierdzenie zostanie udowodnione, system generuje dowód, który potwierdza etap konwersji na klauzulę. Vampire obsługuje popularny format TPTP (Thousands of Problems for Theorem Provers) oparty na tekstowym formacie plików oraz składający się z różnych sekcji i dyrektyw, które pomagają opisać problem. Typowe

pliki TPTP zawierają sekcje takie jak cnf dla formuł logicznych w postaci Klauzulowej Formy Normalnej, fof dla formuł logiki pierwszego rzędu, tff dla formuł z uwzględnieniem typów czy thf dla formuł wyższego rzędu. Dalsze rozważania w niniejszej pracy będą opierały się na formie fof.

7.3 Prover E

Rozwój Provera E [23] zapoczątkowany został w ramach projektu E-SETHEO w TUM. Pierwsza publikacja na jego temat ukazała się w 1998 roku i od tamtej pory system jest stale rozszerzany i udoskonalany. Jego działanie polega na analizie struktur danych matematycznych, takich jak: równania, nierówności, wyrażenia logiczne oraz dowody matematyczne zapisane w określonym formacie. Wykorzystuje on strategie dedukcyjne, w tym metodę rezolucji, przekształceń algebraicznych oraz heurystyki, aby sprawdzić poprawność postawionych twierdzeń. Głównym celem Provera E jest wykrywanie błędów w dowodach matematycznych oraz wspieranie procesu weryfikacji formalnych dowodów. Prover E podobnie jak Prover Vampire akceptuje problemy w pełnej składni logiki pierwszego rzędu. Jeżeli dowód zostanie znaleziony (problem zostanie obalony), system dostarcza listę kroków dowodzenia, w celu indywidualnej weryfikacji. Akceptuje on również popularny format TPTP.

8 Analiza dziennika zdarzeń

Aby poprawnie wczytać dziennik zdarzeń, konieczna jest znajomość nazwy kolumny odpowiadającej identyfikatorowi przypadku, śladu czasu oraz nazwy samej czynności, bądź też zdefiniowanie reguł pozwalających na automatyczne wykrywanie odpowiednich kolumn. Standard XES ma na celu zapewnienie spójnej i rozszerzalnej metody przechowywania logów w dziennikach zdarzeń. Standard ten obejmuje schemat XML, który opisuje strukturę dziennika zdarzeń XES, oraz schemat XML, który opisuje strukturę rozszerzeń takiego dziennika. Dodatkowo, standard zawiera podstawowy zbiór prototypów rozszerzeń XES, które dostarczają semantykę dla pewnych atrybutów rejestrowanych w dzienniku zdarzeń [24]. W kwestii identyfikatora przypadku, w standardzie XES spotkamy się z oznaczeniem “case:concept:name”, natomiast nazwa zdarzenia w standardzie XES będzie oznaczona jako “concept:name”.

W celu wstępnej analizy badanego procesu, zdarzenia występujące w dzienniku mogą zostać pogrupowane w obrębie jednego przypadku, a następnie na podstawie czasu początkowego możliwe jest stworzenie tzw. śladów. Ślady te mogą zostać zliczone, co pozwoli na identyfikację najpopularniejszych sekwencji czynności występujących w procesie. Takie działanie może posłużyć jako punkt wyjścia do tworzenia grafu bezpośrednich połączeń.

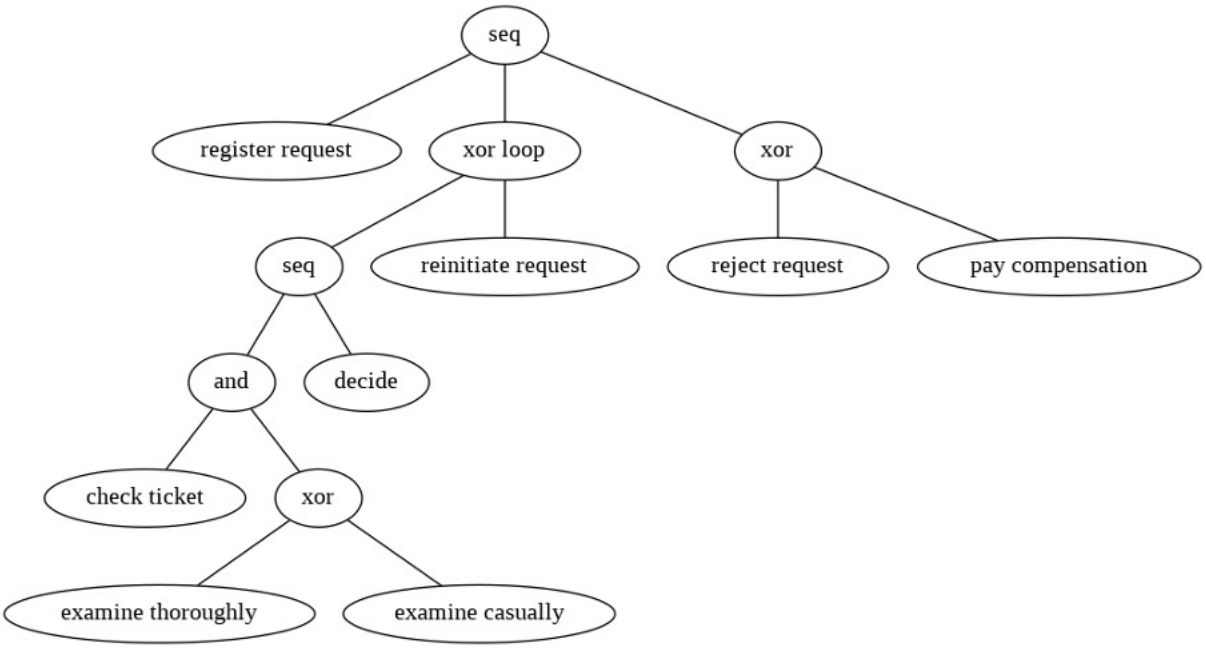
Tab. 7: Zliczone zdarzenia pogrupowane w obrębie każdego przypadku.

	Activity	count
0	Create Purchase Requisition;Analyze Purchase R...	88
1	Create Purchase Requisition;Create Request for...	77
2	Create Purchase Requisition;Analyze Purchase R...	63
3	Create Purchase Requisition;Analyze Purchase R...	48
4	Create Purchase Requisition;Create Request for...	32

8.1 Biblioteka PM4Py

PM4Py (Process Mining for Python) to biblioteka typu open-source służąca do analizy procesów biznesowych. Jest to narzędzie rozwijane przez społeczność akademicką, którego celem jest umożliwienie analizy, eksploracji i wizualizacji procesów biznesowych przy użyciu języka Python. Biblioteka PM4Py umożliwia import i eksport danych w formatach, takich jak CSV czy XES, a także pozwala na filtrację tych danych. Jednym z głównych obszarów funkcjonalnych biblioteki jest zapewnienie implementacji powszechnie stosowanych algorytmów odkrywania procesów oraz ewaluacji wygenerowanych modeli. Biblioteka oferuje również narzędzia do wizualizacji, które umożliwiają tworzenie graficznych reprezentacji procesów, takich jak sieci Pe-

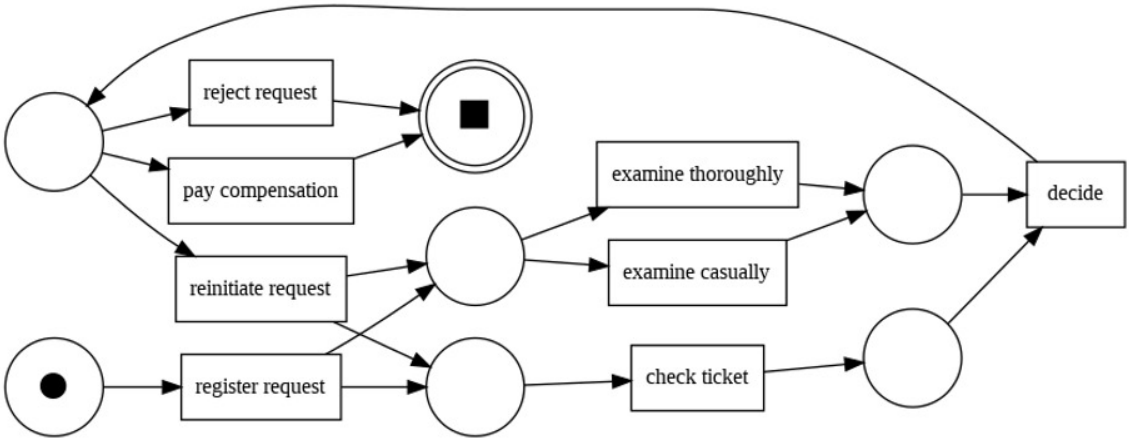
triego czy drzewa procesów. Rys. 15. prezentuje przykładowe drzewo procesu otrzymane przy użyciu tej biblioteki. W drzewie tym możemy wyróżnic następujące wierzchołki: ”seq”, ”xor”, ”$xor \quad loop$” oraz ”and”. Odpowiadają one kolejno wzorcom sekwencji, wyłącznego wyboru, pętli oraz równoległości, które zostały omówione w dziale 2.2.4, przy okazji definicji drzewa procesu.



Rys. 15: Przykładowe drzewo procesu wygenerowane przez bibliotekę PM4Py.

Alpha Miner

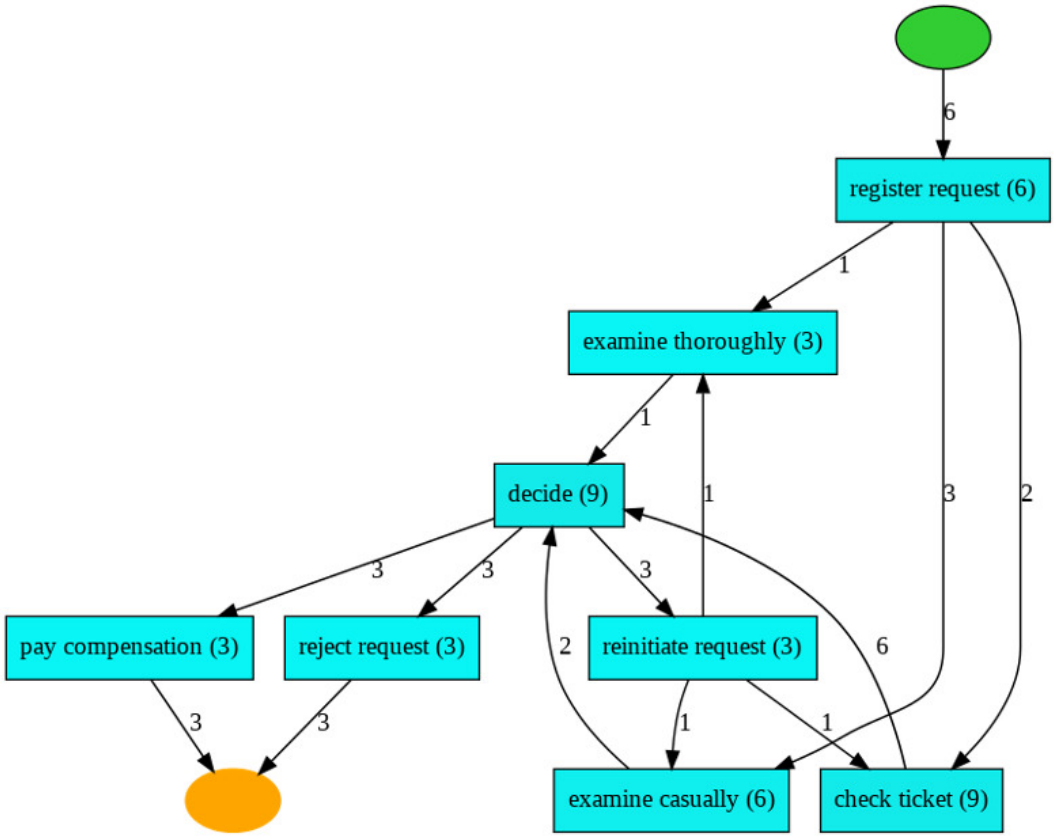
Biblioteka PM4Py udostępnia implementację algorytmu Alpha Miner oraz pozwala na wizualizację uzyskanego wyniku za pomocą sieci Petriego. Jeśli uzyskana sieć jest poprawną siecią WF, możliwe jest także przekonwertowanie jej do postaci drzewa procesu, zgodnie z algorytmem zaprezentowanym w rozdziale 4. W praktyce, jeśli konwersja sieci Petriego, otrzymanej za pomocą algorytmu Alpha Miner na drzewo procesu jest możliwa, to otrzymany wynik będzie taki sam jak ten przy użyciu algorytmu Inductive Miner.



Rys. 16: Sieć Petriego otrzymana za pomocą algorytmu Alpha Miner z wykorzystaniem biblioteki PM4Py.

Heuristics Miner

Po zaimportowaniu dziennika zdarzeń biblioteka PM4Py pozwala na zastosowanie algorytmu Heuristics Miner w celu odkrycia sieci Petriego czy też sieci heurystycznej. Sieć heurystyczna to obiekt grafowy, składający się z wierzchołków (czynności) oraz krawędzi. Dodatkowo sieć taka zawiera etykiety w zależności od tego, jak często w logu występowało dane zdarzenie albo przepływ. Możliwa jest również konwersja otrzymanej sieci Petriego na drzewo procesu. W praktyce jednak, w większości przypadków konwersja ta nie zakończy się pomyślnie. Sieć Petriego otrzymana poprzez algorytm Heuristics Miner jest znacznie bardziej skomplikowana od tej, otrzymanej za pomocą algorytmu Alpha Miner bądź Inductive Miner, a tym samym ciężej o spełnienie warunku poprawności.



Rys. 17: Przedstawienie modelu w postaci sieci heurystycznej otrzymana za pomocą algorytmu Heuristics Miner z wykorzystaniem biblioteki PM4Py.

Inductive Miner

PM4Py udostępnia implementację algorytmu Inductive Miner, a zwrócony wynik można zwizualizować zarówno w postaci drzewa procesu, jak i sieci Petriego, która jest poprawną siecią WF. Biblioteka posiada także implementację modyfikacji tego algorytmu - Inductive Miner Infrequent. Aby skorzystać z filtracji trzeba zmienić parametr wejściowy $noise\_threshold$ z wartości domyślnej wynoszącej 0 na dowolną wartość z zakresu od 0 do 1. Parametr ten dotyczy stopnia ignorowania zakłóceń w dzienniku zdarzeń. Warto zauważyć, że gdy $noise\_threshold$ ustawiony jest na wartość 0 to wytworzony model ma stuprocentową trafność, natomiast nie jest do końca precyzyjny. Zastosowanie filtracji poprzez użycie algorytmu Inductive Miner Infrequent pozwala na poprawę precyzji, zachowując przy tym dość dobrą trafność. Dzieje się tak, ponieważ zasada Pareto może mieć zastosowanie do dzienników zdarzeń. Zazwyczaj 80% obserwowanego zachowania może być wyjaśnione przez model, który stanowi jedynie 20% modelu wymaganego do opisania całego zachowania [5].

9 Generowanie specyfikacji logicznej

Założeniem projektu było zaimplementowanie rozwiązania umożliwiającego automatyczne generowanie specyfikacji logicznych w oparciu o wybrane zatwierdzone wzorce przepływu oraz bezpośrednio z nimi powiązane, predefiniowane wzorce logiczne. Jako pośredni model procesu wybrano drzewo procesu wytworzone na podstawie dziennika zdarzeń, a etykiety drzewa posłużyły do definicji zatwierdzonych wzorców przepływu. Uzyskaną specyfikację logiczną użyto do automatycznej weryfikacji poprawności działania systemu poprzez zastosowanie wybranych, dostępnych systemów automatycznego dowodzenia twierdzeń. Takie podejście umożliwiło efektywną weryfikację zachowań analizowanego systemu.

Stworzony system składa się z 4 elementów:

1. Generator drzewa procesu,
2. Konwerter wzorców z drzewa procesu do zatwierdzonych wzorców przepływu,
3. Predefiniowane wzorce logiczne,
4. Algorytm konwertujący drzewo procesu do postaci specyfikacji logicznej.

9.1 Generowanie drzewa procesu

Proponowane rozwiązanie opiera się na wykorzystaniu drzewa procesu, które zostało otrzymane za pomocą algorytmu Inductive Miner (alternatywnie algorytmu Alpha Miner, jeśli konwersja do drzewa jest możliwa). Drzewo procesu, ze względu na swoją strukturę hierarchiczną oraz klarownie zdefiniowane wzorce przepływu, stanowi dobry punkt wejścia dla algorytmu generującego specyfikację logiczną, jaki zaprezentowano w artykule [20]. Warto zaznaczyć, że algorytm Inductive Miner cechuje się wysoką trafnością (w wersji podstawowej) co oznacza, że jest w stanie dokładnie odtworzyć większość ścieżek z dziennika zdarzeń, w porównaniu do innych algorytmów eksploracji procesów. W ramach testów, sprawdzono specyfikację logiczną otrzymaną dla dwóch różnych dzienników zdarzeń, przy czym jeden z dzienników oceniono przy różnych wartościach parametru zaszumienia, a tym samym różnej trafności. Do wygenerowania drzewa procesu użyto bibliotekę PM4Py opisaną w rozdziale 8.1. Akceptowanym formatem dziennika zdarzeń jest zarówno plik o formacie CSV, jak i XES.

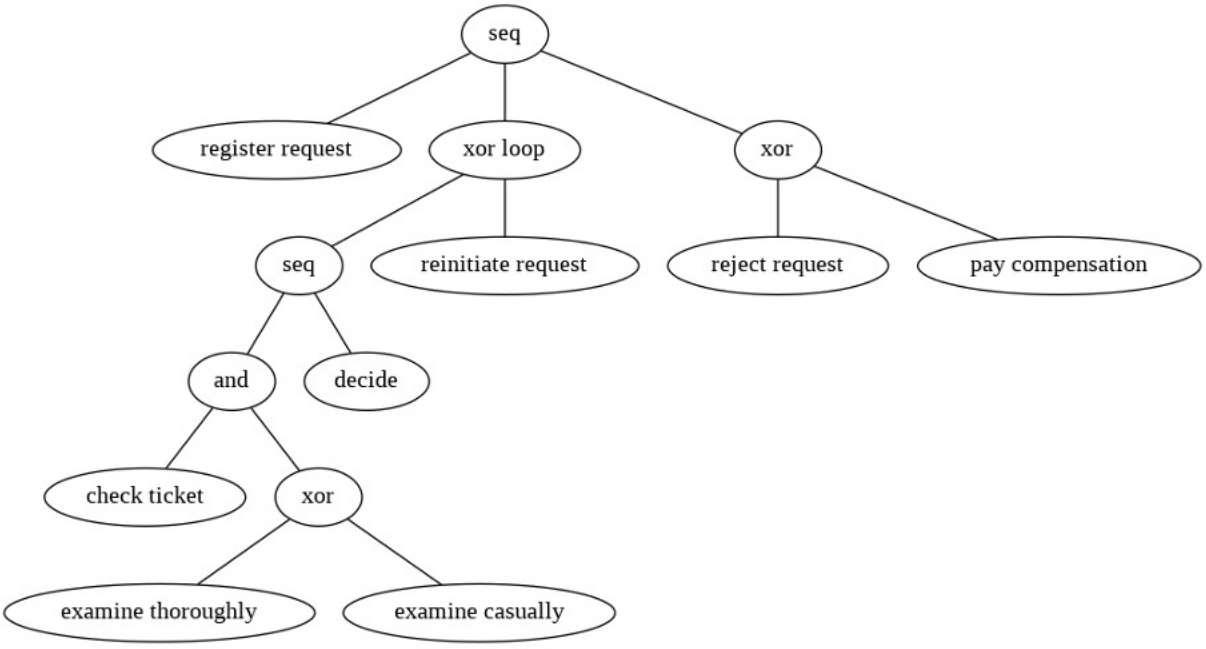
Analizowane dzienniki zdarzeń to:

- **`running-example.xes` [25]**

Tab. 8: Pięć pierwszych logów z dziennika zdarzeń `running-example.xes`.

	Activity	Costs	Resource	case:concept:name	case:creator	org:resource	time:timestam
0	register request	50	Pete	1	Fluxicon Nitro	Pete	2010-12-3 10:02:00+00:0
1	examine thoroughly	400	Sue	1	Fluxicon Nitro	Sue	2010-12-3 09:06:00+00:0
2	check ticket	100	Mike	1	Fluxicon Nitro	Mike	2011-01-0 14:12:00+00:0
3	decide	200	Sara	1	Fluxicon Nitro	Sara	2011-01-0 10:18:00+00:0
4	reject request	200	Pete	1	Fluxicon Nitro	Pete	2011-01-0 13:24:00+00:0

W przypadku dziennika zdarzeń `running-example.xes` możliwe jest otrzymanie drzewa procesu zarówno poprzez użycie algorytmu Inductive Miner jak i poprzez translacja sieci Petriego otrzymanej przy użyciu algorytmu Alpha Miner. Wynik końcowy obu tych operacji jest identyczny i przyjmuje formę przedstawioną na rys. 18.



Rys. 18: Drzewo procesu dla dziennika zdarzeń `running-example.xes`.

Równoważnym zapisem drzewa z Rys. 18. jest wyrażenie wzorcowe W następującej postaci:

$W = \rightarrow$ ('register request', $*(\rightarrow$ ($+$('check ticket', X('examine thoroughly', 'examine casually')),

'decide'), 'reinitiate request'), X('reject request', 'pay compensation'))

Gdzie:

$\rightarrow$ - sekwencja (seq)

$*$ - pętla ($xor \quad loop$)

$+$ - równoległość (and)

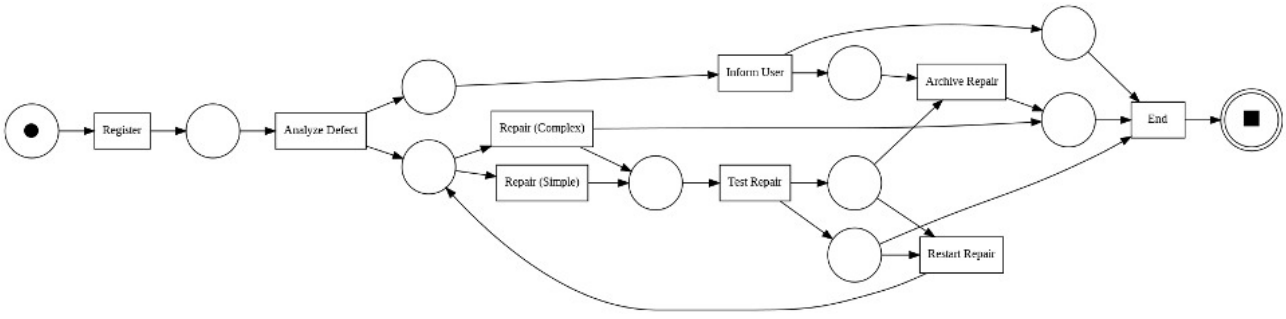
X - wyłączny wybór (xor)

- **`repairexample.xes` [26]**

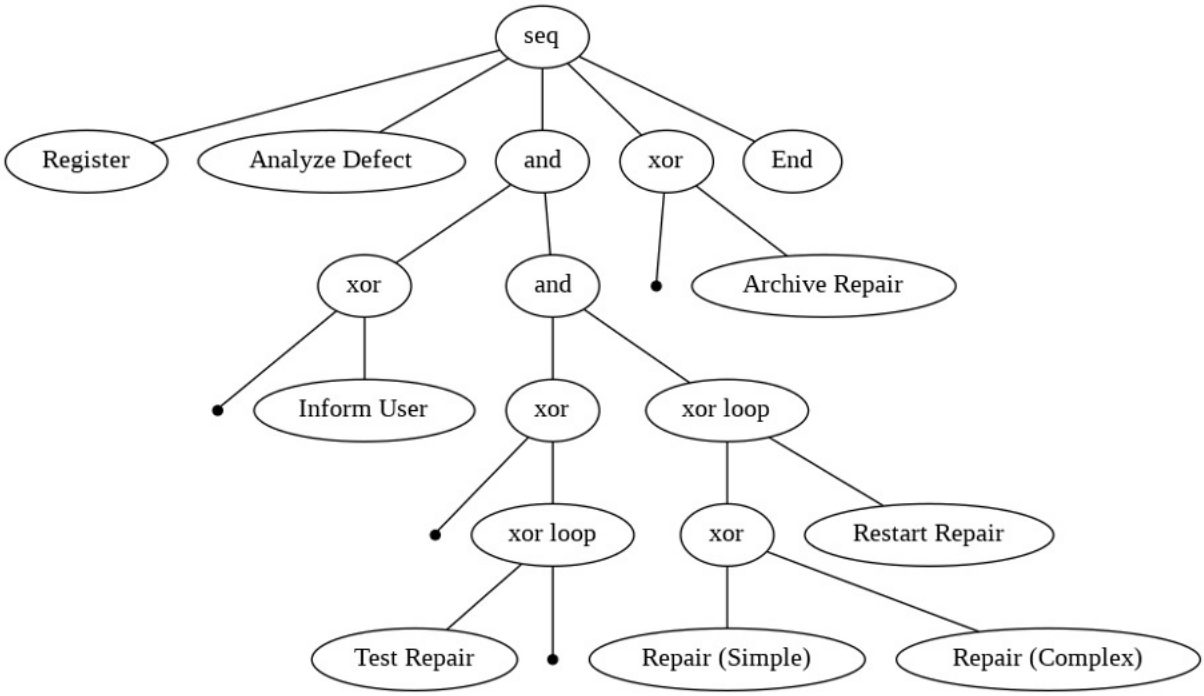
Tab. 9: Pięć pierwszych logów z dziennika zdarzeń `repairexample.xes`.

	Case ID	Activity	Resource	Start Timestamp	Complete Timestamp	Variant	Variant index	(case) creator	(cas varia
0	1	Register	System	1970-01-02 12:23:00+00:00	1970-01-02 12:23:00+00:00	Variant 2	2	Fluxicon Disco	Varia
1	1	Analyze Defect	Tester3	1970-01-02 12:23:00+00:00	1970-01-02 12:30:00+00:00	Variant 2	2	Fluxicon Disco	Varia
2	1	Repair (Complex)	SolverC1	1970-01-02 12:31:00+00:00	1970-01-02 12:49:00+00:00	Variant 2	2	Fluxicon Disco	Varia
3	1	Test Repair	Tester3	1970-01-02 12:49:00+00:00	1970-01-02 12:55:00+00:00	Variant 2	2	Fluxicon Disco	Varia
4	1	Inform User	System	1970-01-02 13:10:00+00:00	1970-01-02 13:10:00+00:00	Variant 2	2	Fluxicon Disco	Varia

Dziennik zdarzeń `repairexample.xes` umożliwia otrzymanie drzewa procesu jedynie poprzez użycie algorytmu Inductive Miner. Sieć Petriego otrzymana poprzez algorytm Alpha Miner nie posiada struktury blokowej, w związku z tym nie jest możliwe sprowadzenie jej do postaci drzewa.



Rys. 19: Sieć Petriego dla dziennika zdarzeń `repairexample.xes`.



Rys. 20: Drzewo procesu dla dziennika zdarzeń `repairexample.xes` orzymana przy użyciu algorytmu Inductive Miner.

Równoważnym zapisem powyższego drzewa procesu jest wyrażenie wzorcowe W następującej postaci:

$W = \rightarrow$ ('Register', 'Analyze Defect', $+(X(tau$, 'Inform User'), $+(X(tau, *$('Test Repair', $tau))$),

$*(X$('Repair-Simple', 'Repair-Complex'), 'Restart Repair'))), $X(tau$, 'Archive Repair'), 'End')

Przy czym tau oznacza cichą aktywność.

9.2 Konwersja wzorców z drzewa procesu do zatwierdzonych wzorców przepływu

Wspomniane w części 8.1. modyfikacje wyrażeń wzorcowych W odbywają się w konwerterze wzorców i są to kolejno:

1. numerowanie zagnieżdżeń,
2. zmiana etykiet drzewa zgodnie z zestawem zatwierdzonych wzorców przepływu,
3. dodanie pozornych zadań (jeśli występują w zatwierdzonych wzorcach przepływu).

Numerowanie zagnieżdżeń

Numerowane wyrażenie wzorcowe W' jest otrzymywane z wyrażenia wzorcowego W zgodnie z regułami podanymi w[20], tj.:

- Każdy otwierający nawias okrągły jest zamykany za pomocą odpowiadającego mu nawiasu kwadratowego, a po jego prawej stronie znajduje się etykieta numeryczna.
- Każdy zamykający nawias okrągły jest otwierany za pomocą odpowiadającego mu nawiasu kwadratowego, a po jego lewej stronie znajduje się etykieta numeryczna.
- Wszystkie sparowane nawiasy okrągłe posiadają tę samą etykietę numeryczną.
- Każde wyrażenie wzorcowe jest przetwarzane w kolejności od lewej do prawej, przy czym najmniejsza dostępna liczba to 1.
- Liczba oznaczająca etykietę zawsze jest wprowadzana jako najmniejsza możliwa wartość, a dopiero później jest wykorzystywana w prawym nawiasie okrągłym.

Dodatkowo, jeśli w nazwie czynności występuje spacja, zostaje zamieniona na symbol podkreślenia „_". Taka operacja ma na celu ułatwienie dalszych modyfikacji wyrażeń wzorcowych.

Wyrażenie W' dla dziennika zdarzeń `running-example.xes`:

$$W' =\rightarrow (1)\text{'register\_request'}, *(2] \rightarrow (3] + (4]\text{'check\_ticket'}, X(5]\text{'examine\_thoroughly'},$$

$$\text{'examine\_casually'}[5)[4), \text{'decide'}[3), \text{'reinitiate\_request'}[2), X(2]\text{'reject\_request'},$$

$$\text{'pay\_compensation'}[2)[1)$$

Wyrażenie W' dla dziennika zdarzeń `repairexample.xes`:

$$W' =\rightarrow (1]\text{Register}, \text{Analyze\_Defect}, +(2]X(3]\text{tau}, \text{Inform\_User}[3), +(3]X(4]\text{tau},$$

$$*(5]\text{Test\_Repair}, \text{tau}[5)[4), *(4]X(5]\text{Repair-Complex}, \text{Repair-Simple}[5),$$

$$\text{Restart\_Repair}[4)[3)[2), X(2]\text{tau}, \text{Archive\_Repair}[2), \text{End}[1)$$

Zamiana etykiet drzewa zgodnie z zestawem zatwierdzonych wzorców przepływu

W oparciu o etykiety występujące w drzewie procesu (czy też wyrażeniu W') oraz ilości czynności podlegających pod każdą z tych etykiet, zdefiniowano następujący zestaw zatwierdzonych wzorców przepływu:

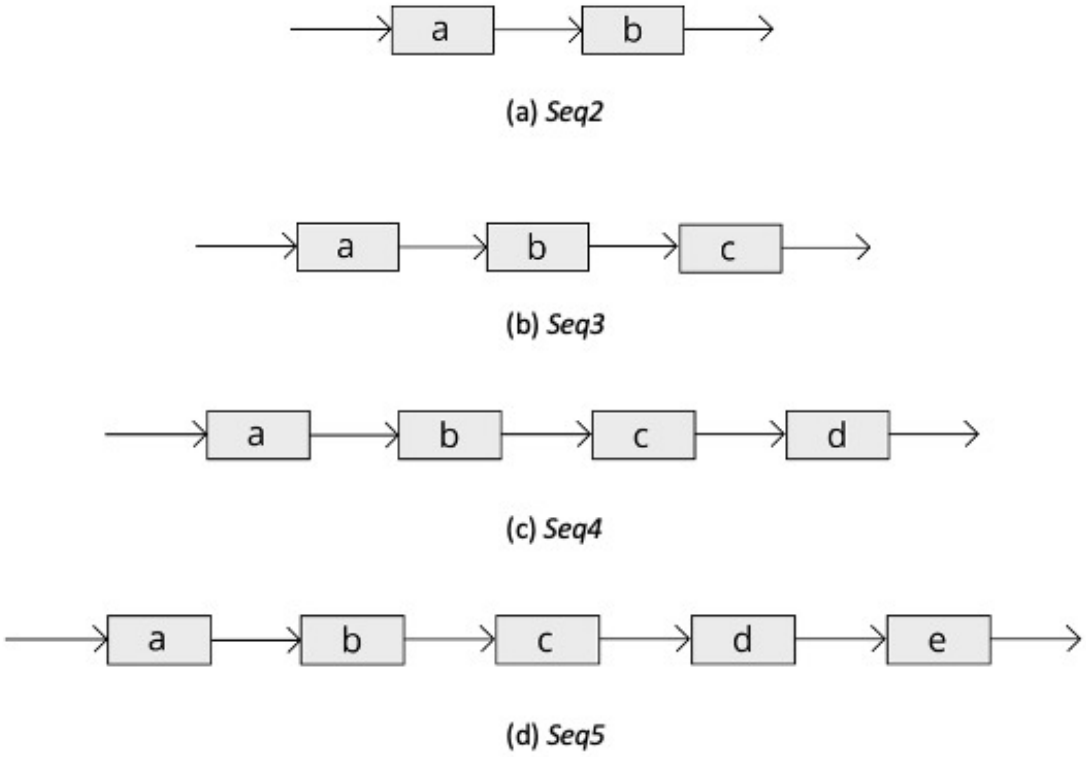
$$\sum = \{Seq2, Seq3, Seq4, Seq5, Xor2, Xor3, And2, And3, Loop\}$$

Warto zaznaczyć, że przypadku każdego z wymienionych wzorców czynnością podległą pod dany wzorzec może być zarówno pojedyncza aktywność, jak i kolejny zagnieżdżony wzorzec.

Każda z etykiet występujących w wyrażeniu W' tj. $\rightarrow, X, +, *$ zostaje zamieniona na nazwę odpowiadającego jej zatwierdzonego wzorca przepływu. Przy czym zatwierdzone wzorce przepływu są konstruowane w następujący sposób:

- **Sekwencja**

W znacznej większości spotykanych przykładów drzew procesu, wzorzec sekwencji składa się z od dwóch do pięciu czynności. W związku z tym zdefiniowano cztery zatwierdzone wzorce przepływu ($Seq2, Seq3, Seq4$ oraz $Seq5$), w celu rozróżnienia ilości czynności z jakiej składa się dany wzorzec oraz umożliwienia wyznaczenia predefiniowanych wzorców logicznych dla każdego z nich. Rys. 21. przedstawia zatwierdzone wzorce przepływu (21.a dla $Seq2$, 21.b dla $Seq3$, 21.c dla $Seq4$ oraz 21.d dla $Seq5$) dla wzorca sekwencji.

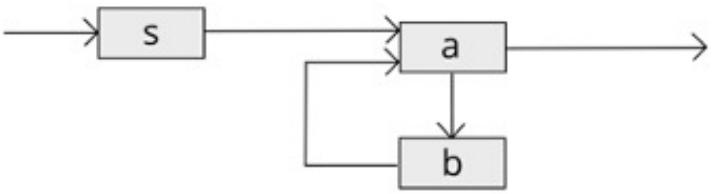


Rys. 21: Zatwierdzone wzorce przepływu dla wzorca sekwencji.

- **Pętla**

W drzewie procesu pętla najczęściej składa się z dwóch naprzemiennych zadań. Po wykonaniu zadania pierwszego (lewej gałęzi etykiety pętli) możliwe jest wykonanie zadania drugiego (prawej gałęzi etykiety pętli) bądź opuszczenie wzorca. Jeśli wykonane zostanie zadanie drugie, skutkuje to ponownym wykonaniem zadania pierwszego. Podobnie jak wcześniej, dopiero po ponownym wykonaniu zadania z lewej gałęzi możliwe jest opuszczenie wzorca.

Przyjęta metoda generowania specyfikacji logicznej zakłada, że każdy zatwierdzony wzorzec przepływu ma jedno zadanie początkowe oraz jedno zadanie końcowe. Zaistniała więc konieczność zdefiniowania dodatkowego zadania s, które jest zadaniem pozornym i symbolizuje rozpoczęcie tego wzorca. Rys. 22. przedstawia zatwierdzony wzorzec przepływu ($Loop$) dla wzorca Pętli.

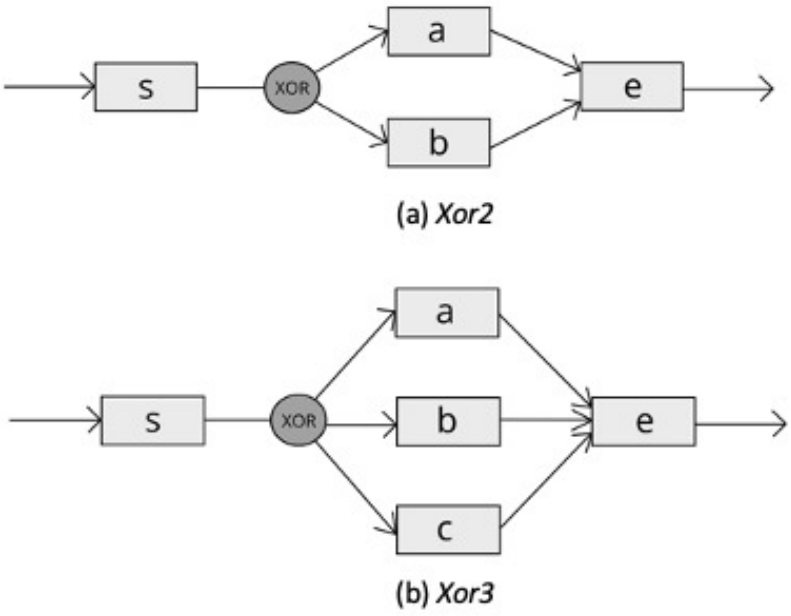


Rys. 22: Zatwierdzony wzorzec przepływu dla wzorca pętli.

- **Wyłączny wybór**

W drzewie procesu wyłączny wybór reprezentuje bramkę XOR. Oznacza to, że może być wykonana zarówno czynność pierwsza jak i druga, jednak nigdy nie obie na raz. Choć możliwe jest występowanie dowolnej ilości czynności we wspomnianym wzorcu, to na potrzeby tej pracy dokonano ograniczenia do trzech czynności.

Ze względu na założenie metody dotyczące obecności jednego zadania początkowego i jednego zadania końcowego w zatwierdzonym wzorcu przepływu, konieczne było wprowadzenie dwóch pozornych zadań. Zadanie s symbolizuje rozpoczęcie wzorca, natomiast zadanie e, jego opuszczenie. Rys. 23. przedstawia zatwierdzone wzorce przepływu (23.a dla $Xor2$ oraz 23.b dla $Xor3$) dla wzorca wyłącznego wyboru.

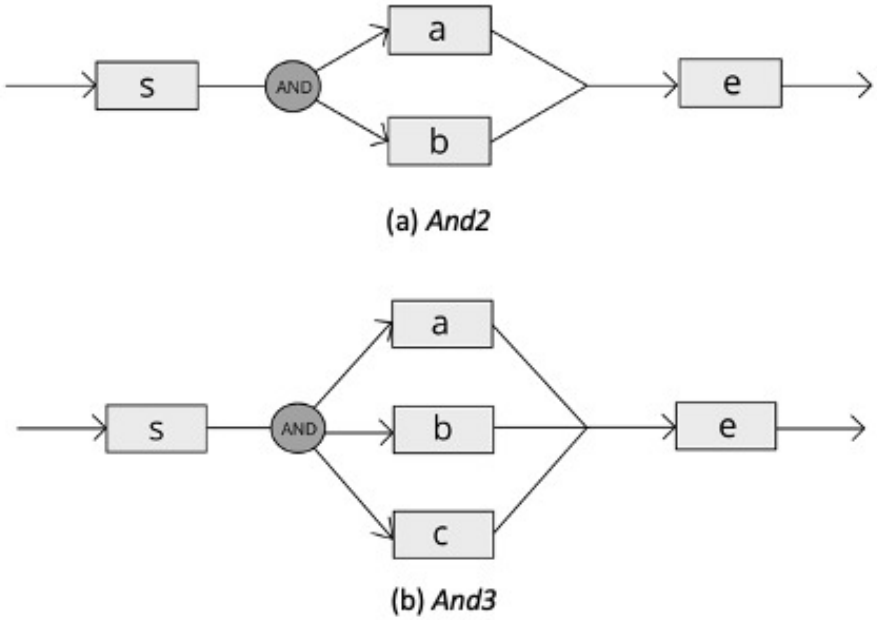


Rys. 23: Zatwierdzone wzorce przepływu dla wzorca wyłącznego wyboru.

- **Równoległość**

Równoległość odnosi się do współbieżności tzn. czynności posiadające ten sam wierzchołek mogą być wykonywane w tym samym czasie. W przypadku wzorca równoległości najczęściej występuje ograniczenie do trzech współbieżnych zadań.

Podobnie jak w przypadku wzorca wyłącznego wyboru, dodano dwa pozorne zadania. Zadanie s symbolizuje rozpoczęcie wzorca, natomiast zadanie e, jego opuszczenie. Rys. 24. przedstawia zatwierdzone wzorce przepływu (24.a dla $And2$ oraz 24.b $And3$) dla wzorca równoległości.



Rys. 24: Zatwierdzone wzorce przepływu dla wzorca równoległości.

Dodanie pozornych zadań

Wraz ze zmianą etykiet drzewa, do wzorców przepływu występujących w wyrażeniu W' dodawane są zadania pozorne, pod warunkiem ich występowania w zatwierdzonym wzorcu przepływu. W ten sposób konstruowane są wyrażenia ZW.

W celu rozróżnienia zadań pozornych s i e w różnych wzorcach (z wyjątkiem wzorców sekwencji, gdzie zadania pozorne nie występują), nadano im nazwy zgodnie z tabelą 10.

Tab. 10: Nazwy pozornych zadań w zależności od zatwierdzonego wzorca przepływu.

Zatwierdzony wzorzec przepływu	Pozorne zadanie s	Pozorne zadanie e
Loop	l_s	-
Xor2	x2_s	x2_e
Xor3	x3_s	x3_e
And2	a2_s	a2_e
And3	a3_s	a3_e

Wyrażenie ZW dla dziennika zdarzeń `running-example.xes`:

$$ZW = Seq3(1)register\_request, Loop(2)l\_s, Seq2(3)And2(4)a2\_s, check\_ticket,$$

$$Xor2(5)x2\_s, examine_thoroughly, examine_casually, x2\_e[5), a2\_e[4), decide[3),$$

$$reinitiate_request[2), Xor2(2]x2\_s, reject\_request, pay_compensation, x2\_e[2)[1)$$

Wyrażenie ZW dla dziennika zdarzeń `repairexample.xes`:

$$ZW = Seq5(1]Register, Analyze\_Defect, And2(2]a2\_s, Xor2(3]x2\_s, tau,$$

$$Inform\_User, x2\_e[3), And2(3]a2\_s, Xor2(4]x2\_s, tau, Loop(5]l\_s, Test\_Repair,$$

$$tau[5), x2\_e[4), Loop(4]l\_s, Xor2(5]x2\_s, Repair - Complex, Repair - Simple, x2\_e[5),$$

$$Restart\_Repair[4), a2\_e[3), a2\_e[2), Xor2(2]x2\_s, tau, Archive\_Repair, x2_e[2), End[1)$$

9.3 Predefiniowane wzorce logiczne

Dla wszystkich zatwierdzonych wzorców przepływu wyznaczono bezpośrednio z nimi powiązane, predefiniowane wzorce logiczne $\sum$. Zostały one przedstawione w Tabeli 11.

Tab. 11: Zbiór predefiniowanych wzorców logicznych.

$\Sigma = \{$ $\mathbf{Seq2}(a,b) = \langle a, b, \Diamond a, \Box(a \Rightarrow \Diamond b), \Box\neg(a \wedge b)\rangle$,
$\mathbf{Seq3}(a,b,c) = \langle a, c, \Diamond a, \Box(a \Rightarrow \Diamond b), \Box(b \Rightarrow \Diamond c), \Box\neg(a \wedge b), \Box\neg(a \wedge c), \Box\neg(b \wedge c)\rangle$,
$\mathbf{Seq4}(a,b,c,d) = \langle a, d, \Diamond a, \Box(a \Rightarrow \Diamond b), \Box(b \Rightarrow \Diamond c), \Box(c \Rightarrow \Diamond d), \Box\neg(a \wedge b), \Box\neg(a \wedge c), \Box\neg(a \wedge d), \Box\neg(b \wedge c), \Box\neg(b \wedge d), \Box\neg(c \wedge d)\rangle$,
$\mathbf{Seq5}(a,b,c,s,e) = \langle a, e, \Diamond a, \Box(a \Rightarrow \Diamond b), \Box(b \Rightarrow \Diamond c), \Box(c \Rightarrow \Diamond d), \Box(d \Rightarrow \Diamond e), \Box\neg(a \wedge b), \Box\neg(a \wedge c), \Box\neg(a \wedge d), \Box\neg(a \wedge e), \Box\neg(b \wedge c), \Box\neg(b \wedge d), \Box\neg(b \wedge e), \Box\neg(c \wedge d), \Box\neg(c \wedge e), \Box\neg(d \wedge e)\rangle$,
$\mathbf{Xor2}(s,a,b,e) = \langle s, e, \Diamond s, \Box(s \Rightarrow (\Diamond a \wedge \neg\Diamond b) \vee (\neg\Diamond a \wedge \Diamond b)), \Box((a \vee b) \Rightarrow \Diamond e), \Box\neg(s \wedge a), \Box\neg(s \wedge b), \Box\neg(s \wedge e), \Box\neg(a \wedge b), \Box\neg(a \wedge e), \Box\neg(b \wedge e)\rangle$,
$\mathbf{Xor3}(s,a,b,c,e) = \langle s, e, \Diamond s, \Box(s \Rightarrow (\Diamond a \wedge \neg\Diamond b \wedge \neg\Diamond c) \vee (\neg\Diamond a \wedge \Diamond b \wedge \neg\Diamond c) \vee (\neg\Diamond a \wedge \neg\Diamond b \wedge \Diamond c)), \Box((a \vee b \vee c) \Rightarrow \Diamond e), \Box\neg(s \wedge a), \Box\neg(s \wedge b), \Box\neg(s \wedge c), \Box\neg(s \wedge e), \Box\neg(a \wedge b), \Box\neg(a \wedge c), \Box\neg(a \wedge e), \Box\neg(b \wedge c), \Box\neg(b \wedge e), \Box\neg(c \wedge e)\rangle$,
$\mathbf{And2}(s,a,b,e) = \langle s, e, \Diamond s, \Box(s \Rightarrow \Diamond a \wedge \Diamond b), \Box(a \Rightarrow \Diamond e), \Box(b \Rightarrow \Diamond e), \Box\neg(s \wedge (a \vee b)), \Box\neg((a \vee b) \wedge e)\rangle$,
$\mathbf{And3}(s,a,b,c,e) = \langle s, e, \Diamond s, \Box(s \Rightarrow \Diamond a \wedge \Diamond b \wedge \Diamond c), \Box(a \Rightarrow \Diamond e), \Box(b \Rightarrow \Diamond e), \Box(c \Rightarrow \Diamond e), \Box\neg(s \wedge (a \vee b \vee c)), \Box\neg((a \vee b \vee c) \wedge e)\rangle$,
$\mathbf{Loop}(s,a,b) = \langle s, a, \Diamond s, \Box(s \Rightarrow \Diamond a), \Box(a \Rightarrow (\Diamond b \wedge \Diamond a) \vee \neg\Diamond b), \Box(b \Rightarrow \Diamond a), \Box\neg(s \wedge a), \Box\neg(s \wedge b), \Box\neg(a \wedge b)\rangle$ $\}$

Przy czym, zgodnie z [20], predefiniowany wzorzec logiczny P jest to struktura postaci:

$$P = (ini, fin, \text{pat}(a_1, a_2, \ldots))$$

Gdzie:

ini, fin - to wyrażenia logiczne opisujące okoliczności odpowiednio rozpoczęcia i zakończenia całego zatwierdzonego wzorca przepływu (tj. zadanie początkowe oraz końcowe),

pat() - to zbiór formuł logicznych logiki temporalnej, opisujący zachowanie zatwierdzonego wzorca przepływu, czyli wszystkich wymagań, które muszą zostać spełnione.

Zasada spełnialności dotyczy stwierdzenia, czy istnieje takie przypisanie wartości logicznych (prawda lub fałsz) zmiennym w formule logicznej (lub zestawie formuł), które sprawi, że cała ta formuła (lub formuły) stanie się prawdziwa. Jeśli takie przypisanie istnieje, formuła (lub zestaw formuł) jest uważana za spełnialną. Zasada ta ma również zastosowanie do predefiniowanych wzorców logicznych. W oparciu o [20], „wzorzec logiczny P jest poprawnie skomponowany wtedy, gdy wszystkie formuły P.pat() są spełnialne i P.pat() $\models$ P.Tr, gdzie P.Tr są formułami przejścia dla P oraz P.Tr $\equiv$ Tr $\equiv \{\Box\neg(P.\text{ini} \wedge P.\text{fin}), \Diamond P.\text{ini}, \Box(P.\text{ini} \Rightarrow \Diamond P.\text{fin})\}$. Formuły przejścia opisują zarówno aspekty związane z bezpieczeństwem, jak i żywotnością wzorca logicznego. Z jednej strony zapewniają, że wykonanie wzorca zostanie zakończone w określonych sytuacjach. Z drugiej strony, pozwalają na pominięcie szczegółów wewnętrznego zachowania wzorca i analizowanie go jako całość.”

Właściwości bezpieczeństwa i żywotności w logice temporalnej pozwalają na odzwierciedlenie wymagań stawianych rzeczywistym systemom oraz odnoszą się kolejno do:

Właściwość bezpieczeństwa służy do opisu takiego zdarzenia, które nigdy nie wystąpi. (gdzie a to opisywane zdanie logiczne).

$$\Box\neg(a)$$

Właściwość żywotności wskazuje, że dane zdarzenie na pewno się wydarzy (gdzie a to zdanie logiczne będące warunkiem, natomiast b to opisywane zdanie logiczne).

$$\Box(a \Rightarrow \Diamond b)$$

Zbiór predefiniowanych wzorców logicznych Σ został zawarty w pliku konfiguracyjnym o nazwie `approved_patterns.json` oraz jest bezpośrednio wykorzystywany przez tworzony system do generowania specyfikacji logicznej dla analizowanego dziennika zdarzeń. Wzorce logiczne zawarte w tym pliku zostały zapisane w postaci logiki pierwszego rzędu poprzez zastąpienie operatorów $\Box$ oraz $\Diamond$ odpowiednimi kwantyfikatorami tak, jak zostało to przedstawione w dziale 6.3.

9.4 Algorytm konwertujący drzewo procesu do postaci specyfikacji logicznej

Oprócz tworzenia numerowanych wyrażeń wzorcowych W', na metodę generującą specyfikację logiczną zgodnie z artykułem [20] składają się dwa algorytmy:

1. Obliczanie skonsolidowanych wyrażeń

Wyrażenie W^i (lub W^f) dla wyrażenia wzorcowego W to skonsolidowane wyrażenie ini (lub odpowiednio fin) tworzone według następujących reguł:

- Jeśli w miejscu dowolnego argumentu atomowego wyrażenia W nie ma kolejnego wzorca, który syntaktycznie należy do wyrażenia ini (lub odpowiednio wyrażenia fin), to W^i jest równe $W.ini$ (lub W^f jest równe $W.fin$),
- Jeśli w miejscu dowolnego argumentu wyrażenia W zamiast dowolnego argumentu atomowego występuje kolejny wzorzec, np. $A()$, który syntaktycznie należy do wyrażenia ini (lub odpowiednio wyrażenia fin), wówczas argument atomowy zostaje zastąpiony przez A^i (lub odpowiednio A^f).

2. Generowanie specyfikacji logicznych

Algorytm do generowania specyfikacji logicznej posiada dwa wejścia: wyrażenie W oraz zbiór predefiniowanych wzorców logicznych $\sum$, natomiast jego wynik to specyfikacja logiczna L. W przypadku tworzonego systemu algorytm zawiera drobną modyfikację, a mianowicie wejściowe wyrażenie W jest tak naprawdę wyrażeniem ZW, w związku z tym linia 2 algorytmu zostaje pominięta. W dalszych krokach, zamiast na zmiennej lab, operowano na wejściowej zmiennej ZW.

Algorithm 1: Algorytm generujący specyfikację logiczną [20].

Algorithm 3 Generating logical specifications ($\mathcal{A}3$).

Input: pattern expression w
Input: predefined pattern property set Σ (non-empty)
Output: logical specification L

```
1:  L:= ∅;                                              ▷ initiating specification
2:  lab:= Labelling(w);                                 ▷ Algorithm 1
3:  for l:= max(lab) downto 1 do
4:      c:=1;                                           ▷ current pattern for label l to get
5:      p:= getPat(lab, l, c);                          ▷ first/leftmost pattern for label l
6:      repeat
7:          L2:= Σ.p.pat();                             ▷ take PLTL formulas for p from Σ
8:          for j:=1 to |p| do                          ▷ for every argument in pattern p
9:              if p ↑ j is non-atomic then             ▷ then a_j is a pattern itself
10:                 cons:= ConsEx(p ↑ j, ini, Σ) + "∨" +   ▷ Algorithm 2 (x 2)
11:                        ConsEx(p ↑ j, fin, Σ);       ▷ to consolidate
12:             replace in L2 every p ↑ j by cons
13:         end if
14:     end for
15:     L := L ∪ L2;
16:     c++;
17:     p:= getPat(lab, l, c)                           ▷ next pattern for label l
18:     until p = ε                                     ▷ no next pattern
19: end for
20: return L
```

1

Funkcja `max()` zwraca największą etykietę przekazanego jako argument wyrażenia. Symbol $\uparrow$ jest selektorem argumentu. W $\uparrow i$ oznacza i-ty argument wyrażenia W. Natomiast funkcja `getPat()` zwraca kolejny zagnieżdżony wzorzec z ZW, z etykietą l na c-tej pozycji od lewej oraz usuwa wprowadzoną etykietę. $L2$ to zmienna, która jest używana do przechowywania szczegółów specyfikacji dla danego wzorca. Jeśli wszystkie elementy tego wzorca są atomowe, to $L2$ jest dodawane do wynikowej specyfikacji bez wprowadzania żadnych zmian [20].

10 Analiza własności logicznych

Zarówno Vampire Prover, jak i E Prover, przekształcają wprowadzone twierdzenia do postaci Klauzulowej Formy Normalnej (CNF), która stanowi standardowy format w większości systemów automatycznego dowodzenia twierdzeń opartych na logice pierwszego rzędu. CNF jest reprezentacją, gdzie każde twierdzenie jest przedstawione jako koniunkcja klauzul. Każda klauzula stanowi alternatywę literałów, przy czym każdy literał może być zmienną lub jej negacją. Proces przekształcania formuł do takiej postaci jest realizowany automatycznie przez oba systemy, w związku z czym formuły składające się na specyfikację systemu mogą obejmować format FOF. Format ten stanowi bardziej ogólną reprezentację, pozwala na reprezentację różnorodnych rodzajów formuł logicznych, w tym aksjomatów, przypuszczeń (dowodów), a także bardziej ogólnych zdań logicznych zawierających operatory logiczne, takie jak: koniunkcja, alternatywa, implikacja, a także kwantyfikatory.

10.1 Translacja formuł do formatu TPTP

W formacie TPTP formuły logiczne są konstruowane przy użyciu składni logiki pierwszego rzędu oraz uwzględniają symbole, takie jak:

- & – koniunkcja,
- | – alternatywa,
- $\Leftrightarrow$ – równoważnoś,
- $\Rightarrow$ – implikacja,
- ! – kwantyfikator ogólny,
- ? – kwantyfikator egzystencjalny,
- $\sim$ – zaprzeczenie.

Formuły są zapisywane w plikach z rozszerzeniem $.p$, przy czym dla formatu fof zazwyczaj przyjmują one formę składającą się z aksjomatów (`axiom`) bądź dowodów (`conjecture`).

Przykład aksjomatu w TPTP:

```
fof(formula, axiom, ?[X]: (a[X]))
```

Przykład dowodu w TPTP:

```
fof(thesis, conjecture, ![X]: ((a(X)) => ?[X1]: (b(X1) | c(X1)))).
```

Każda formuła otrzymana w wygenerowanej specyfikacji logicznej jest konwertowana według następującego schematu:

- Na początku każdej formuły dodawany jest następujący ciąg znaków:

  $$\text{fof(formula}\{\text{i}\}\text{, axiom,}$$

  gdzie {i} wskazuje na numer formuły począwszy od 1.

- Na końcu każdej formuły dodawany jest nawias zamykający oraz kropka.

- Wszystkie wystąpienia kwantyfikatora uniwersalnego w formule zamieniane są na następujący ciąg znaków:

  $$![X] :$$

- Wszystkie wystąpienia kwantyfikatora egzystencjalnego w formule zamieniane są na następujący ciąg znaków:

  $$?[X\{j\}] :$$

  Przy czym {j} nie występuje, gdy kwantyfikator ten znajduje się na początku formuły. W pozostałych przypadkach {j} wskazuje na ilość wystąpień tego kwantyfikatora w formule począwszy od 1.

- Każdy symbol koniunkcji do tej pory występujący jako symbol ^ zamieniany jest na &.

- Jeśli nazwa zmiennej rozpoczyna się od wielkiej litery, musi zostać zmieniona na małą literę (wymagane do prawidłowego działania Provera Vampire, natomiast nie ma znaczenia dla Provera E).

Tak przetworzone formuły przyjmują postać aksjomatów, czyli fundamentalnych twierdzeń, które przyjęte są jako prawdziwe, aby zbudować daną teorię. Zbiór wszystkich formuł wygenerowanych dla danego dziennika zdarzeń stanowi specyfikację logiczną analizowanego systemu.

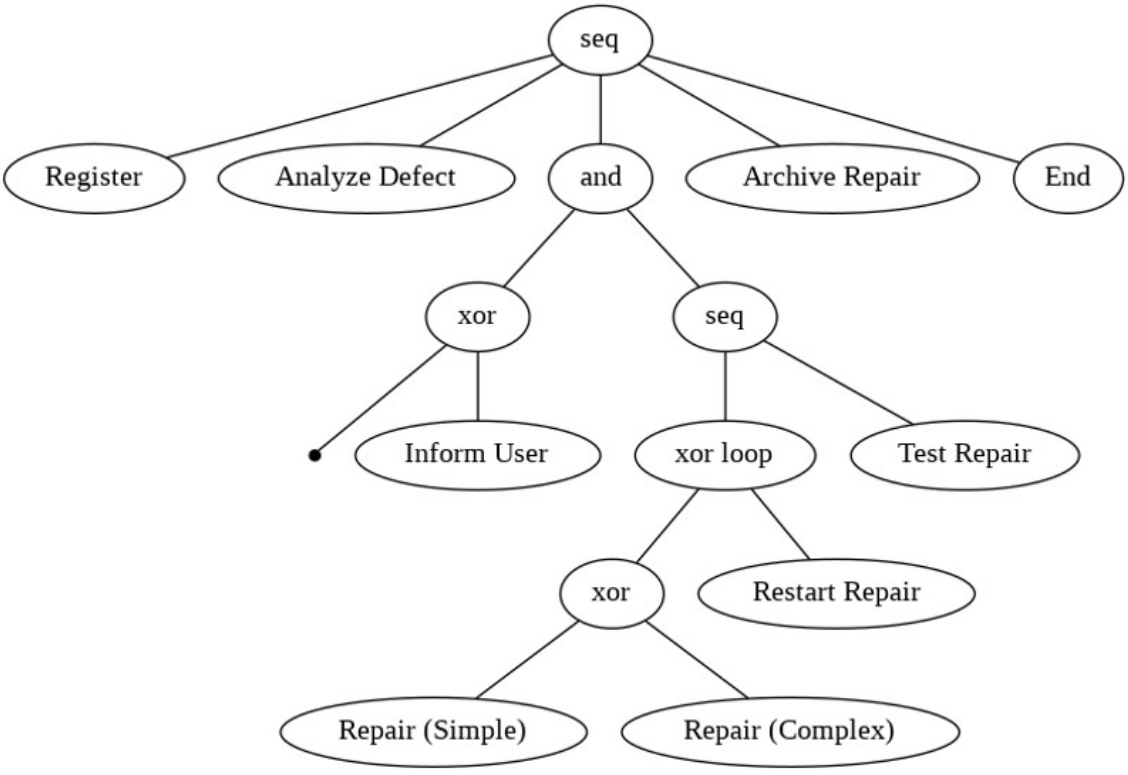
Specyfikację logiczne wytworzone dla dzienników zdarzeń `running-example.xes` oraz `repairexample.xes` zawarte zostały w Dodatku A. Przy czym dla drugiego dziennika zdarzeń wygenerowano specyfikację logiczne dla drzew procesu otrzymanych przy ustawieniu parametru progu zaszumienia na kolejno 0 (wartość domyślna), 0.5 oraz 1. Drzewa te zaprezentowano na poniższych rysunkach.

Wyrażenie ZW dla dziennika zdarzeń `repairexample.xes` przy wartości parametru zaszumienia wynoszącej 0.5:

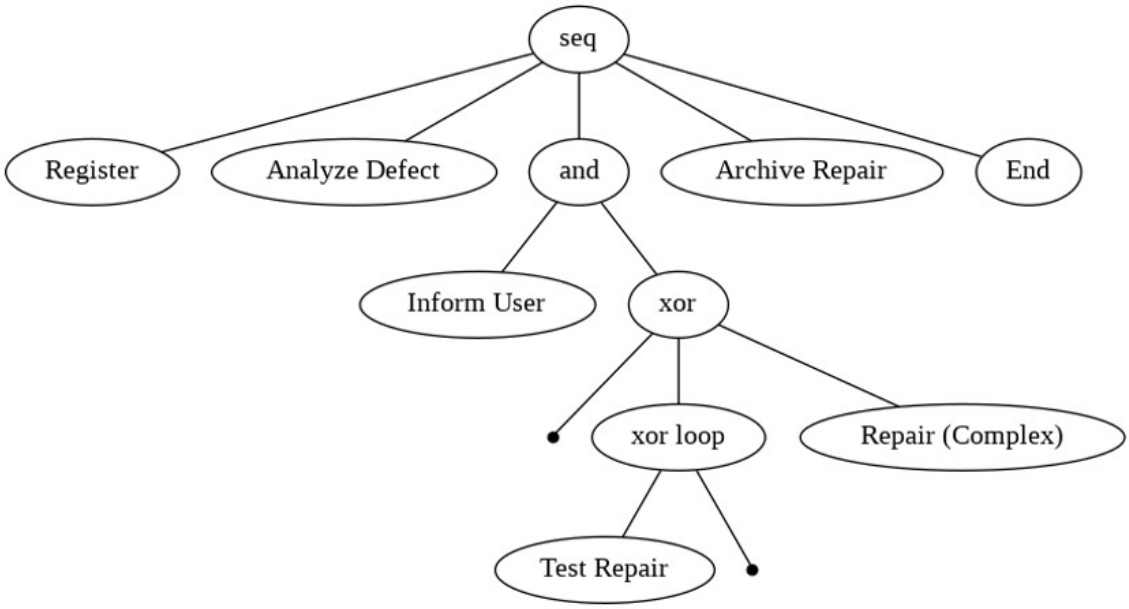
$$ZW = Seq5(1]Register, Analyze\_Defect, And2(2]a2\_s, Xor2(3]x2\_s, tau,$$
$$Inform\_User, x2_e[3), Seq2(3]Loop(4]l\_s, Xor2(5]x2\_s, Repair\_Simple$$
$$Repair\_Complex, x2\_e[5), Restart\_Repair[4), Test\_Repair[3), a2\_e[2),$$
$$Archive\_Repair, End[1)$$

Wyrażenie ZW dla dziennika zdarzeń `repairexample.xes` przy wartości parametru zaszumienia wynoszącej 1:

$$ZW = Seq5(1]Register, Analyze\_Defect, And2(2]a2\_s, Inform\_User, Xor3(3]x3\_s, tau,$$
$$Repair\_Complex, Loop(4]l\_s, Test\_Repair, tau[4), x3\_e[3), a2\_e[2), Archive\_Repair,$$
$$End[1)$$



Rys. 25: Drzewo procesu dla dziennika zdarzeń `repairexample.xes` przy wartości parametru zaszumienia wynoszącej 0.5.



Rys. 26: Drzewo procesu dla dziennika zdarzeń `repairexample.xes` przy wartości parametru zaszumienia wynoszącej 1.

10.2 Wyniki

Na podstawie wygenerowanych specyfikacji, w osobnych plikach zdefiniowano problemy, które posłużyły do weryfikacji ich poprawności. Wytworzone pliki to:

- **`problem1.p`** - problem wygenerowany na podstawie dziennika zdarzeń `running-example.xes` przy domyślnej wartości progu zaszumienia,
- **`problem2.p`** - problem wygenerowany na podstawie dziennika zdarzeń `repairexample.xes` przy domyślnej wartości progu zaszumienia,
- **`problem3.p`** - problem wygenerowany na podstawie dziennika zdarzeń `repairexample.xes` przy progu zaszumienia ustawionym na 0.5,
- **`problem4.p`** - problem wygenerowany na podstawie dziennika zdarzeń `repairexample.xes` przy progu zaszumienia ustawionym na 1.

Weryfikację wygenerowanych specyfikacji logicznych wykonano poprzez analizę spełnialności dla każdego z problemów przy użyciu dwóch systemów automatycznego dowodzenia twierdzeń, Provera Vampire oraz Provera E. Na podstawie doboru problemów można wyróżnić następujące przypadki testowe:

1. Sprawdzenie spełnialności każdej ze specyfikacji.
2. Porównanie spełnialności dwóch specyfikacji wygenerowanych przez algorytm Inductive Miner, przy tych samych parametrach, dla dzienników zdarzeń o różnej złożoności.

3. Sprawdzenie czy redukcja złożoności dziennika zdarzeń, poprzez zastosowanie filtracji (Inductive Miner Infrequent), wpłynie na spełnialność specyfikacji.

W poniższej tabeli przedstawiono porównanie oczekiwanych fragmentów wyjścia każdego z Proverów dla wszystkich z problemów.

Tab. 12: Wyniki spełnialności dla specyfikacji 1, 2, 3 oraz 4.

Nr problemu	Plik	Wyjście Provera Vampire	Wyjście Provera E
1	problem1.p	% SZS status Satisfiable for problem1 % Termination reason: Satisfiable	# No proof found! # SZS status Satisfiable
		Specyfikacja spełnialna	Specyfikacja spełnialna
2	problem2.p	% Refutation found % Termination reason: Refutation	# Proof found! # SZS status Unsatisfiable
		Specyfikacja niespełnialna	Specyfikacja niespełnialna
3	problem3.p	% SZS status Satisfiable for problem3 % Termination reason: Satisfiable	# No proof found! # SZS status Satisfiable
		Specyfikacja spełnialna	Specyfikacja spełnialna
4	problem4.p	% SZS status Satisfiable for problem4 % Termination reason: Satisfiable	# No proof found! # SZS status Satisfiable
		Specyfikacja spełnialna	Specyfikacja spełnialna

W kontekście automatycznego dowodzenia twierdzeń, `SZS status Satisfiable` oznacza, że analizowana specyfikacja może być spełniona, czyli istnieje co najmniej jedno przypisanie wartości, które spełnia określone warunki. `SZS status Unsatisfiable` wskazuje na sprzeczność, uniemożliwiającą istnienie spełniającego przypisania wartości dla wszystkich założeń jednocześnie.

W oparciu o zawartość wygenerowanych plików oraz uzyskane wyniki, postanowiono przeanalizować dwa kolejne przypadki testowe:

1. Sprawdzenie czy dwie specyfikacje są równoważne.
2. Sprawdzenie czy jedna specyfikacja implikuje inną specyfikacje.

Z uwagi na ograniczenia wybranych systemów, analiza takich przypadków dla specyfikacji jako całości nie jest wykonalna. Dlatego konieczne było zapisanie obu specyfikacji w jednym pliku oraz przedstawienie ich jako koniunkcji poszczególnych formuł wewnątrz zdefiniowanego przypuszczenia. Zawartość przykładowego pliku, sprawdzającego równoważność specyfikacji, została zaprezentowana na Rys. 27.

```
% Definicje formul dla Specyfikacji 1
fof(formula1_spec1, axiom, (...)).
fof(formula2_spec1, axiom, (...)).
% ... kolejne formuly dla Specyfikacji 1 ...

% Definicje formul dla Specyfikacji 2
fof(formula1_spec2, axiom, (...)).
fof(formula2_spec2, axiom, (...)).
% ... kolejne formuly dla Specyfikacji 2 ...

% Teza do udowodnienia
fof(thesis, conjecture,
    ![X]: : ((formula1_spec1(X) & formula2_spec1(X)) <=> (formula1_spec2(X) & formula2_spec2(X)))).
```

Rys. 27: Przykładowa zawartość pliku sprawdzającego równoważność specyfikacji.

- **`problem5.p`** - problem sprawdzający czy specyfikacja 2 implikuje specyfikację 3,
- **`problem6.p`** - problem sprawdzający czy specyfikacja 3 implikuje specyfikację 4,
- **`problem7.p`** - problem sprawdzający czy specyfikacja 1 implikuje specyfikację 4,
- **`problem8.p`** - problem sprawdzający czy specyfikacja 2 jest równoważna specyfikacji 3,
- **`problem9.p`** - problem sprawdzający czy specyfikacja 3 jest równoważna specyfikacji 4,
- **`problem10.p`** - problem sprawdzający czy specyfikacja 1 jest równoważna specyfikacji 4.

W tabeli 13. przedstawiono porównanie oczekiwanych fragmentów wyjścia, każdego z Proverów, dla podanych problemów.

Tab. 13: Wyniki dla dowodów problemów 5, 6, 7 oraz 8, 9, 10.

Nr problemu	Plik	Wyjście Provera Vampire	Wyjście Provera E
5	problem5.p	`# SZS status Refutation` `% Termination reason: Refutation`	`# Proof found!` `# SZS status ContradictoryAxioms`
		Niespełnialność koniunkcji aksjomatów oraz zaprzeczenie dowodu	Sprzeczność między aksjomatami
6	problem6.p	`% SZS status CounterSatisfiable for problem6` `% Termination reason: Satisfiable`	`# No proof found!` `# SZS status CounterSatisfiable`
		Spełnialność koniunkcji aksjomatów, ale zaprzeczenie dowodu	Spełnialność koniunkcji aksjomatów, ale zaprzeczenie dowodu
7	problem7.p	`% SZS status CounterSatisfiable for problem7` `% Termination reason: Satisfiable`	`# No proof found!` `# SZS status CounterSatisfiable`
		Spełnialność koniunkcji aksjomatów, ale zaprzeczenie dowodu	Spełnialność koniunkcji aksjomatów, ale zaprzeczenie dowodu
8	problem8.p	`# SZS status Refutation` `% Termination reason: Refutation`	`# Proof found!` `# SZS status ContradictoryAxioms`
		Niespełnialność koniunkcji aksjomatów i zaprzeczenie dowodu	Sprzeczność między aksjomatami
9	problem9.p	`% SZS status CounterSatisfiable for problem9` `% Termination reason: Satisfiable`	`# No proof found!` `# SZS status CounterSatisfiable`
		Spełnialność koniunkcji aksjomatów, ale zaprzeczenie dowodu	Spełnialność koniunkcji aksjomatów, ale zaprzeczenie dowodu
10	problem10.p	`% SZS status CounterSatisfiable for problem10` `% Termination reason: Satisfiable`	`# No proof found!` `# SZS status CounterSatisfiable`
		Spełnialność koniunkcji aksjomatów, ale zaprzeczenie dowodu	Spełnialność koniunkcji aksjomatów, ale zaprzeczenie dowodu

`SZS status Refutation` występuje, gdy system znalazł sprzeczność lub konflikt w logicznej strukturze twierdzenia. Ma to miejsce podczas sytuacji, w której wykazano, że koniunkcja aksjomatów jest niespełnialna, a także udowodniono negacje zdefiniowanego przypuszczenia (dowodu). `SZS status CounterSatisfiable` odnosi się do sytuacji, w której określone przypuszczenie (dowód) jest błędne, co oznacza, że istnieje kontradykcja w możliwości spełnienia tego założenia [27]. O ile `Refutation` oznacza, że zarówno dowód jak i specyfikacja (koniunkcja aksjomatów) jest niespełnialna, to `CounterSatisfiable` występuje w sytuacji, gdy

istnieje model spełniający koniunkcje aksjomatów oraz zaprzeczenie przypuszczenia. Termin `ContradictoryAxioms` oznacza sprzeczne aksjomaty. W przypadku wyników Provera taki komunikat wskazuje, że narzędzie wykryło sprzeczność w zdefiniowanych aksjomatach. `SZS status Theorem` będzie występował w sytuacji, gdy zarówno specyfikacja jak i przypuszczenie jest spełnialne w ramach przyjętej logiki.

Kolejnym zagadnieniem wartym poruszenia, jest badanie implikacji wygenerowanej specyfikacji na uprzednio zdefiniowane klauzule bezpieczeństwa i żywotności w kontekście konkretnego, analizowanego dziennika zdarzeń. Ponieważ specyfikacja 2 jest niespełnialna, w analizie ograniczono się jedynie do specyfikacji 1, 3 i 4.

Dla specyfikacji 1 zdefiniowano dwa, osobne przypuszczenia(dowody) sprawdzające jej implikacje na następującą klauzulę żywotności:

$$\Box(\text{register\_request} \Rightarrow \Diamond(\text{reject\_request} \vee \text{pay\_compensation})) \tag{5}$$

Oraz bezpieczeństwa:

$$\Box\neg(\text{reject\_request} \wedge \text{pay\_compensation}) \tag{6}$$

Ponieważ specyfikacje 3 oraz 4 tyczą się tego samego dziennika zdarzeń, przy różnej wartości parametru filtracji postanowiono sprawdzić, jak klauzule zdefiniowane dla specyfikacji 3 będą się miały do specyfikacji 4. Klauzule wybrano tak, że powinny być one konsekwencją modelu użytego do wygenerowania specyfikacji 3, jednak nie powinny one występować dla specyfikacji 4.

Przypuszczenie zawierające klauzule żywotnością dla specyfikacji 3 i 4:

$$\Box(\text{register} \Rightarrow \Diamond(\text{repair\_Simple} \vee \text{repair\_Complex})) \tag{7}$$

Przypuszczenie zawierające klauzule bezpieczeństwa dla specyfikacji 3 i 4:

$$\Box\neg(\text{inform\_User} \wedge tau) \tag{8}$$

Dowody zdefiniowane na podstawie zaprezentowanych klauzul zostały dodane na koniec każdego pliku zawierającego specyfikacje (oprócz `problem2.p`) tj. 1,3 oraz 4. Na tej podstawie udało się zdefiniować kolejne problemy, w taki sposób, że:

- **`problem11.p`** - Problem sprawdzający czy specyfikacja 1 implikuje klauzule (5),
- **`problem12.p`** - Problem sprawdzający czy specyfikacja 1 implikuje klauzule (6),
- **`problem13.p`** - Problem sprawdzający czy specyfikacja 3 implikuje klauzule (7),
- **`problem14.p`** - problem sprawdzający czy specyfikacja 3 implikuje klauzule (8),
- **`problem15.p`** - problem sprawdzający czy specyfikacja 4 implikuje klauzule (7),
- **`problem16.p`** - problem sprawdzający czy specyfikacja 4 implikuje klauzule (8).

Poniższa tabela przedstawia porównanie wyjść Proverów dla poszczególnych problemów.

Tab. 14: Wyniki dla dowodów implikacji specyfikacji 1, 3 i 4 na wybrane klauzule bezpieczeństwa i żywotności.

Nr problemu	Nazwa pliku	Klauzule	Wyjście Provera Vampire	Wyjście Provera E
11	problem11.p	Klauzula żywotności (5)	% SZS status Theorem for problem11	# Proof found! # SZS status Theorem
			Dowód potwierdzony	Dowód potwierdzony
12	Problem12.p	Klauzula bezpieczeństwa (6)	% SZS status Theorem for problem12	# Proof found! # SZS status Theorem
			Dowód potwierdzony	Dowód potwierdzony
13	Problem13.p	Klauzula żywotności (7)	% SZS status Theorem for problem13	# Proof found! # SZS status Theorem
			Dowód potwierdzony	Dowód potwierdzony
14	Problem14.p	Klauzula bezpieczeństwa (8)	% SZS status Theorem for problem14	# Proof found! # SZS status Theorem
			Dowód potwierdzony	Dowód potwierdzony
15	Problem15.p	Klauzula żywotności (7)	% SZS status CounterSatisfiable for problem15	# No proof found! # SZS status CounterSatisfiable
			Dowód obalony	Dowód obalony
16	Problem16.p	Klauzula bezpieczeństwa (8)	% SZS status Theorem for problem16	# Proof found! # SZS status Theorem
			Dowód potwierdzony	Dowód potwierdzony

10.3 Wnioski

Zarówno program Vampire Prover, jak i E Prover wskazują na spełnialność wygenerowanych specyfikacji 1, 3 oraz 4 oraz niespełnialność specyfikacji 2. Warto zaznaczyć, że specyfikacja 2 wyróżnia się wśród pozostałych pod względem złożoności, wynikającej z licznych pozornych zadań, dodanych w celu wsparcia algorytmu generowania drzewa i spełnienia wymogu osiągnięcia stuprocentowej trafności modelu. Rezultatem tej większej złożoności jest także większa liczba wygenerowanych formuł logicznych w specyfikacji 2, przekraczająca liczbę 60, przy czym pozostałe specyfikacje składają się z około 40 formuł. Istotne jest również spostrzeżenie, że specyfikacje 3 i 4 opierają się na tym samym dzienniku zdarzeń co specyfikacja 2. Wnioskuje się zatem, że zastosowanie filtracji nie tylko prowadzi do redukcji złożoności modelu oraz poprawy parametrów prostoty i precyzji, ale także ma wpływ na spełnialność równoważnej formalnej specyfikacji. Innym istotnym spostrzeżeniem jest to, że dla mniej złożonych dzienników zdarzeń, jak w przypadku `running-example.xes`, nie jest konieczne stosowanie filtracji, szczególnie gdy istotne jest zachowanie wysokiej trafności modelu.

Wyniki obu Proverów dla dowodów zawartych w problemach 6, 7 oraz 9, 10 wskazują na sprzeczne dowody, natomiast w przypadku problemów 5 i 8, podobnie jak przy problemie 2, na sprzeczne aksjomaty. Wynik te nie wzbudzają jednak wątpliwości. Badanie wpływu operacji algebraicznej, takiej jak filtracja dziennika zdarzeń na wygenerowaną specyfikację logiczną, a także na implikacje i równoważność między różnymi specyfikacjami, stanowi zagadnienie o dużej złożoności, które wymaga dalszych badań. Dodatkowo wykazano, że na podstawie wygenerowanej specyfikacji możliwa jest analiza zdefiniowanych klauzul żywotności i bezpieczeństwa. W przypadku problemów 11, 12, 13 i 14, Provery wykazały spełnialność przypuszczenia dla zdefiniowanych klauzul, natomiast przypadku problemu 15 wykazały, że przypuszczenie jest błędne dla zdefiniowanej klauzuli żywotności. Warto jednak zauważyć, że otrzymano potwierdzenie dowodu dla problemu 16, a tym samym dla zaproponowanej klauzuli bezpieczeństwa. Ponieważ klauzula bezpieczeństwa dotyczy sytuacji, że pewnie niepożądane zdarzenie nigdy nie nastąpi, potwierdzenie dowodu zawierającego taką klauzule dla specyfikacji 4 zdaje się sytuacją oczekiwaną. Wynik ten umożliwił weryfikacje zastosowanego podejścia, w kontekście analizowania dowolnie zdefiniowanych właściwości procesu, przy użyciu metod formalnych przedstawionych w niniejszej pracy.

11 Podsumowanie

Założeniem projektu dyplomowego było stworzenie systemu do automatycznego wydobywania wzorców zachowań z logów systemowych oraz generowania na ich podstawie równoważnej specyfikacji logicznej. W tym celu przybliżono zagadnienia związane z logiką temporalną, logiką pierwszego rzędu, systemami dowodzenia twierdzeń oraz tematem eksploracji procesów. Porównano istniejące algorytmy służące do wydobywania informacji z dzienników zdarzeń takie jak Alpha Miner, Heuristics Miner, Inductive Miner oraz jego modyfikacje. Przedstawiono różne notacje, stosowane do wizualnej reprezentacji modelu. Omówiono metody do konwertowania modeli z jednej notacji do drugiej oraz nakreślono warunki, przy jakich jest to możliwe. Ze względu na swoją strukturę hierarchiczną jako pośredni model procesu, wybrano drzewo procesu. Etykiety drzewa posłużyły do definicji zatwierdzonych wzorców przepływu oraz predefiniowanych wzorców logicznych. Na podstawie [20] opisano algorytm pozwalający na generowanie równoważnej specyfikacji logicznej. W końcu przedstawiono sposób modyfikacji drzewa procesu do postaci wyrażeń ZW, odpowiadających wejściu omawianego algorytmu oraz zaprezentowano wygenerowane specyfikacje.

Tak zrealizowane cele projektu umożliwiły wyznaczenie kolejnych, dotyczących szerokiej analizy wygenerowanych specyfikacji. W ramach analizy zdefiniowano 16 problemów badających własności wygenerowanych specyfikacji oraz skorzystano z dwóch systemów automatycznego dowodzenia twierdeń dla logiki pierwszego rzędu. Zarówno Vampire Prover, jak i E Prover nie dostarczają dowodu dla problemów 1, 3 i 4, w przeciwieństwie do problemu 2, sugerując ich spełnialność. Problem 2 wyróżnia się bardziej złożoną specyfikacją zawierającą liczne pozorne zadania, co skutkuje większą liczbą formuł logicznych. Problemy 3 i 4 bazują na tym samym dzienniku zdarzeń co problem 2, jednak zostały uprzednio poddane filtracji. Może to wskazać na istotność upraszczania modeli procesu. Ponadto można wysunąć wniosek, że filtracja może być zbędna w przypadku prostych dzienników zdarzeń, takich jak running-example.xes, jeśli konieczne jest zachowanie wysokiej trafności modelu. Wyniki obu Proverów dla problemów 6, 7, 9 i 10 wskazują na sprzeczne dowody, podczas gdy dla problemów 5 i 8 (podobnie jak w problemie 2), wykazano występowanie sprzecznych aksjomatów. Dalsze badania dotyczące wpływu operacji algebraicznej, takiej jak filtracja dziennika zdarzeń na specyfikację logiczną oraz implikacje i równoważność między różnymi specyfikacjami są jednak niezbędne, ze względu na złożoność tego zagadnienia.

Warto zauważyć, że wybór najlepszego modelu procesu jest kwestią sporną, gdyż każdy model jest jedynie pewnym odzwierciedleniem procesu. Proponowane rozwiązanie może być dodatkowym narzędziem służącym do weryfikacji oraz porównywania modeli procesu, jak i również algorytmów je generujących. W ramach pracy udało się wykazać możliwość analizy wygenerowanej specyfikacji w kontekście jej implikacji na zdefiniowane klauzule żywotności i bezpieczeństwa. Dzięki temu możliwe jest nie tylko upewnienie się, czy specyfikacja wygene-

rowana dla badanego procesu jest poprawna, ale także możliwe jest sprawdzenie, czy spełnione są pewne istotne warunki dotyczące funkcjonowania procesu.

12 Bibliografia

[1] Klimek, R. *Wprowadzenie do logiki temporalnej*, Kraków: AGH. Uczelniane Wydawnictwa Naukowo-Dydaktyczne (1999r.).

[2] Rafiei, M., Van der Aalst, W. *Mining Roles From Event Logs While Preserving Privacy*, Lecture Notes in Business Information Processing book series (LNBIP,volume 362) (2020r.).

[3] Szmuc, W. *Modelowanie wybranych diagramów języka UML 2.0 z zastosowaniem kolorowanych sieci Petriego*, Kraków: Akademia Górniczo-Hutnicza, Rozprawa doktorska (2014r.), s. 11-13.

[4] Van der Aalst, W. Workflow Nets and Soundness, *Process Mining: Data Science in Action*, Coursera (ostatni dostęp 16.08.2023r.)

[5] Leemans S., Fahland D., van der Aalst W. *Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour*, Conference: International Conference on Business Process Management, Eindhoven University of Technology, the Netherlands (2015r.).

[6] van der Aalst, W. *A practitioner's guide to process mining: Limitations of the directly-follows graph*, CENTERIS - International Conference on ENTERprise Information System, Science Direct (2019r.), s. 321-328.

[7] van Zelst, S. J., and Leemans, S. J. *Translating workflow nets to process trees: An algorithmic approach*, Algorithms 13, 11 (2020r.).

[8] Van der Aalst, W. *Process Mining. Discovery, Conformance and Enhancement of Business Processes*, Springer (2011r.), s. 129-167.

[9] de Medeiros, A., van Dongen, B., van der Aalst, W., Weijters, A. *Process Mining for Ubiquitous Mobile Systems: An Overview and a Concrete Algorithm*, Ubiquitous Mobile Information and Collaboration Systems UMICS (2004r.) s. 156-170.

[10] Wen, L., van der Aalst, W., Wang, J., Sun., J. *Mining process models with non-free-choice constructs. Data Mining and Knowledge Discovery*, 15(2):145-180, (2007r).

[11] Wen, L., Wang, J., van der Aalst W., B. Huang, J. Sun. *Mining Process Models with Prime Invisible Tasks. Data and Knowledge Engineering*, 69(10):999-1021, (2010r).

[12] Bogarín, A., Romero, C., Cerezo, R., Sánchez-Santillán, M. *Clustering for improving educational process mining. Proceedings of the Fourth International Conference on Learning Analytics and Knowledge* (pp. 170-181). Indianapolis, USA (2014r.).

[13] van der Aalst, W., *Process Mining: Data Science in Action. Springer Berlin Heidelberg*, (2016r.) s. 222-233.

[14] Ghawi, R., *Process Discovery using Inductive Miner and Decomposition. A Submission to the Process Discovery Contest*, Technical Report (2016r.).

[15] J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst, *On the Role of Fitness, Precision, Generalization and Simplicity in Process Discovery*, OTM Confederated International Conferences, On the Move to Meaningful Internet Systems, (2012r.).

[16] Leemans, S. J., Fahland, D., van der Aalst, W. M., *Process and Deviation Exploration with Inductive Visual Miner*, BPM (Demos), (2014r.)

[17] Bogarín A., Cerezo R., Romero C., *Discovering learning processes using Inductive Miner: A case study with Learning Management Systems (LMSs)*, Psicotherma, (2018r.)

[18] Goranko V., Galton A., *Temporal Logic, Stanford Encyclopedia of Philosophy [online]*, CSLI, Stanford University (2020r.).

[19] Głuchowski P., *Logika Temporalna i Automaty Czasowe (2) Logika LTL*, Politechnika Wrocławska, wersja 2.1 (ostatni dostęp 12.09.2023 r.)

[20] Klimek. R., *Pattern-based and composition-driven automatic generation of logical specifications for workflow-oriented software models*, Journal of Logical and Algebraic Methods in Programming 104 (2019r.), s. 201–226.

[21] https://en.wikipedia.org/wiki/Kripke_structure_(model_checking) (ostatni dostęp 12.09.2023 r.)

[22] Vampire (theorem prover) https://en.wikipedia.org/wiki/Vampire_(theorem_prover) (ostatni dostęp 17.09.2023 r.)

[23] The E Theorem Prover, https://wwwlehre.dhbw-stuttgart.de/ sschulz/E/E.html (ostatni dostęp 17.09.2023 r.)

[24] EEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams https://xes-standard.org/ (ostatni dostęp 18.09.2023 r.)

[25] Dziennik zdarzeń running-example https://pm4py.fit.fraunhofer.de/getting-started-page (ostatni dostęp 18.09.2023 r.)

[26] Dziennik zdarzeń repairexample https://ai.ia.agh.edu.pl/pl:dydaktyka:dss:lab02 (ostatni dostęp 18.09.2023 r.)

[27] Schreiner, W. *First-order logic: software for proving, Course "Computational logic "*, Research Institute for Symbolic Computation (RISC) (ostatni dostęp 19.11.2023 r.)

Dodatek A

Specyfikacja 1

Otrzymana specyfikacja logiczna dla dziennika zdarzeń running-example.xes, przy domyślnych parametrach algorytmu Inductive Miner:

```
ForAll((register_request) => Exist(l_s | decide))
ForAll((check_ticket) => Exist(a2_e))
ForAll(~((examine_thoroughly) ^ (x2_e)))
ForAll((a2_s | a2_e) => Exist(decide))
ForAll(~((x2_s) ^ (x2_e)))
ForAll(((pay_compensation) | (reject_request)) => Exist(x2_e))
ForAll(~((l_s) ^ (a2_s | decide)))
ForAll((x2_s | x2_e) => Exist(a2_e))
ForAll((reinitiate_request) => Exist(a2_s | decide))
ForAll(~((pay_compensation) ^ (reject_request)))
ForAll((a2_s) => (Exist(check_ticket) ^ Exist(x2_s | x2_e)))
ForAll(~((a2_s | a2_e) ^ (decide)))
ForAll((l_s) => Exist(a2_s | decide))
ForAll(~((register_request) ^ (l_s | decide)))
ForAll(~((a2_s) ^ ((check_ticket) | (x2_s | x2_e))))
ForAll(~((x2_s) ^ (reject_request)))
ForAll(~((reject_request) ^ (x2_e)))
ForAll(~((l_s | decide) ^ (x2_s | x2_e)))
Exist(register_request)
ForAll((x2_s) => ((Exist(examine_casually) ^
~(Exist(examine_thoroughly))) | (~(Exist(examine_casually)) ^
Exist(examine_thoroughly))))
ForAll(~((pay_compensation) ^ (x2_e)))
ForAll((examine_casually) | (examine_thoroughly)) => Exist(x2_e))
Exist(l_s)
ForAll(~(((check_ticket) | (x2_s | x2_e)) ^ (a2_e)))
ForAll(~((x2_s) ^ (pay_compensation)))
ForAll((x2_s) => ((Exist(pay_compensation) ^
~(Exist(reject_request))) | (~(Exist(pay_compensation)) ^
Exist(reject_request))))
ForAll(~((examine_casually) ^ (x2_e)))
ForAll((l_s | decide) => Exist(x2_s | x2_e))
ForAll(~((register_request) ^ (x2_s | x2_e)))
Exist(a2_s)
ForAll(~((x2_s) ^ (examine_thoroughly)))
ForAll(~((examine_casually) ^ (examine_thoroughly)))
Exist(a2_s | a2_e)
ForAll(~((l_s) ^ (reinitiate_request)))
ForAll(~((a2_s | decide) ^ (reinitiate_request)))
ForAll(~((x2_s) ^ (examine_casually)))
Exist(x2_s)
ForAll((a2_s | decide) => ((Exist(reinitiate_request) ^ Exist(a2_s |
decide)) | (~(Exist(reinitiate_request)))))
```

- **Specyfikacja 2**

Otrzymana specyfikacja logiczna dla dziennika zdarzeń repairExample.xes, przy domyślnych parametrach algorytmu Inductive Miner:

```
ForAll(~((tau) ^ (x2_e)))
ForAll((tau) => Exist(Test_Repair))
ForAll((a2_s) => (Exist(x2_s | x2_e) ^ Exist(a2_s | a2_e)))
ForAll((x2_s) => ((Exist(tau) ^ ~(Exist(Archive_Repair))) |
(~(Exist(tau)) ^ Exist(Archive_Repair))))
ForAll(~((x2_s | x2_e) ^ (End)))
ForAll(~((a2_s) ^ ((x2_s | x2_e) | (l_s | x2_e))))
ForAll((Restart_Repair) => Exist(x2_s | x2_e))
ForAll(~((x2_s) ^ (Inform_User)))
ForAll(~((x2_s) ^ (x2_e)))
ForAll(~((Register) ^ (a2_s | a2_e)))
ForAll(~((l_s | Test_Repair) ^ (x2_e)))
ForAll((x2_s | x2_e) => ((Exist(Restart_Repair) ^ Exist(x2_s |
x2_e)) | (~(Exist(Restart_Repair)))))
ForAll(((tau) | (Inform_User)) => Exist(x2_e))
ForAll((x2_s | x2_e) => Exist(a2_e))
ForAll(~((a2_s | a2_e) ^ (x2_s | x2_e)))
Exist(Register)
ForAll(~((Repair_Simple) ^ (x2_e)))
ForAll(~((x2_s) ^ (l_s | Test_Repair)))
ForAll(~((l_s) ^ (Test_Repair)))
ForAll(~((Repair_Complex) ^ (x2_e)))
ForAll(((Repair_Simple) | (Repair_Complex)) => Exist(x2_e))
Exist(x2_s)
ForAll(~((l_s) ^ (Restart_Repair)))
ForAll(~((Inform_User) ^ (x2_e)))
ForAll((l_s) => Exist(x2_s | x2_e))
ForAll(~((tau) ^ (Archive_Repair)))
ForAll(~((Register) ^ (Analyze_Defect)))
ForAll(~((tau) ^ (l_s | Test_Repair)))
ForAll(~((Repair_Simple) ^ (Repair_Complex)))
ForAll(~((Analyze_Defect) ^ (a2_s | a2_e)))
ForAll((a2_s | a2_e) => Exist(a2_e))
ForAll((l_s) => Exist(Test_Repair))
ForAll((l_s | x2_e) => Exist(a2_e))
ForAll(~((l_s) ^ (tau)))
ForAll((Test_Repair) => ((Exist(tau) ^ Exist(Test_Repair)) |
(~(Exist(tau)))))
ForAll(~((Test_Repair) ^ (tau)))
ForAll(~((tau) ^ (Inform_User)))
Exist(l_s)
ForAll(~((x2_s) ^ (tau)))
ForAll((Analyze_Defect) => Exist(a2_s | a2_e))
ForAll(~((Register) ^ (x2_s | x2_e)))
ForAll((x2_s) => ((Exist(Repair_Simple) ^ ~(Exist(Repair_Complex)))
| (~(Exist(Repair_Simple)) ^ Exist(Repair_Complex))))
ForAll(~(((x2_s | x2_e) | (a2_s | a2_e)) ^ (a2_e)))
ForAll((x2_s | x2_e) => Exist(End))
ForAll(~((l_s) ^ (x2_s | x2_e)))
```

```
ForAll((x2_s) => ((Exist(tau) ^ ~(Exist(l_s | Test_Repair))) |
(~(Exist(tau)) ^ Exist(l_s | Test_Repair))))
ForAll((a2_s | a2_e) => Exist(x2_s | x2_e))
ForAll((Register) => Exist(Analyze_Defect))
Exist(a2_s)
ForAll(~(((x2_s | x2_e) | (l_s | x2_e)) ^ (a2_e)))
ForAll(~((x2_s | x2_e) ^ (Restart_Repair)))
ForAll(~((x2_s) ^ (Repair_Complex)))
ForAll(~((a2_s) ^ ((x2_s | x2_e) | (a2_s | a2_e))))
ForAll((x2_s) => ((Exist(tau) ^ ~(Exist(Inform_User))) |
(~(Exist(tau)) ^ Exist(Inform_User))))
ForAll(~((Register) ^ (End)))
ForAll(~((Analyze_Defect) ^ (x2_s | x2_e)))
ForAll(((tau) | (Archive_Repair)) => Exist(x2_e))
ForAll(~((Analyze_Defect) ^ (End)))
ForAll((a2_s) => (Exist(x2_s | x2_e) ^ Exist(l_s | x2_e)))
ForAll(~((a2_s | a2_e) ^ (End)))
ForAll(~((x2_s) ^ (Repair_Simple)))
ForAll(((tau) | (l_s | Test_Repair)) => Exist(x2_e))
ForAll(~((Archive_Repair) ^ (x2_e)))
ForAll(~((x2_s) ^ (Archive_Repair)))
```

- **Specyfikacja 2a**

Otrzymana specyfikacja logiczna dla dziennika zdarzeń repairExample.xes, przy parametrze noise_threshold algorytmu Inductive Miner ustawionym na 0.5:

```
ForAll(~((tau) ^ (x2_e)))
ForAll(~(((x2_s | x2_e) | (l_s | Test_Repair)) ^ (a2_e)))
ForAll((Restart_Repair) => Exist(x2_s | x2_e))
ForAll(~((x2_s) ^ (Inform_User)))
ForAll((Archive_Repair) => Exist(End))
ForAll(~((x2_s) ^ (x2_e)))
ForAll(~((Register) ^ (a2_s | a2_e)))
ForAll((l_s | x2_e) => Exist(Test_Repair))
ForAll(~((Archive_Repair) ^ (End)))
ForAll((x2_s | x2_e) => ((Exist(Restart_Repair) ^ Exist(x2_s |
x2_e)) | (~(Exist(Restart_Repair)))))
ForAll(((tau) | (Inform_User)) => Exist(x2_e))
ForAll((x2_s | x2_e) => Exist(a2_e))
Exist(Register)
ForAll(~((Analyze_Defect) ^ (Archive_Repair)))
ForAll(~((Repair_Simple) ^ (x2_e)))
Exist(l_s | x2_e)
ForAll(~((l_s | x2_e) ^ (Test_Repair)))
ForAll(~((Repair_Complex) ^ (x2_e)))
ForAll(((Repair_Simple) | (Repair_Complex)) => Exist(x2_e))
Exist(x2_s)
ForAll(~((l_s) ^ (Restart_Repair)))
ForAll(~((Inform_User) ^ (x2_e)))
ForAll((l_s) => Exist(x2_s | x2_e))
ForAll((a2_s) => (Exist(x2_s | x2_e) ^ Exist(l_s | Test_Repair)))
ForAll(~((Register) ^ (Analyze_Defect)))
ForAll(~((Repair_Simple) ^ (Repair_Complex)))
ForAll(~((Analyze_Defect) ^ (a2_s | a2_e)))
ForAll(~((Register) ^ (Archive_Repair)))
ForAll(~((tau) ^ (Inform_User)))
ForAll(~((a2_s) ^ ((x2_s | x2_e) | (l_s | Test_Repair))))
Exist(l_s)
ForAll(~((x2_s) ^ (tau)))
ForAll((Analyze_Defect) => Exist(a2_s | a2_e))
ForAll((x2_s) => ((Exist(Repair_Simple) ^ ~(Exist(Repair_Complex)))
| (~(Exist(Repair_Simple)) ^ Exist(Repair_Complex))))
ForAll(~((l_s) ^ (x2_s | x2_e)))
ForAll((Register) => Exist(Analyze_Defect))
Exist(a2_s)
ForAll(~((x2_s | x2_e) ^ (Restart_Repair)))
ForAll(~((x2_s) ^ (Repair_Complex)))
ForAll((l_s | Test_Repair) => Exist(a2_e))
ForAll((x2_s) => ((Exist(tau) ^ ~(Exist(Inform_User))) |
(~(Exist(tau)) ^ Exist(Inform_User))))
ForAll((a2_s | a2_e) => Exist(Archive_Repair))
ForAll(~((Register) ^ (End)))
ForAll(~((a2_s | a2_e) ^ (Archive_Repair)))
ForAll(~((Analyze_Defect) ^ (End)))
```

```
ForAll(~((a2_s | a2_e) ^ (End)))
ForAll(~((x2_s) ^ (Repair_Simple)))
```

- **Specyfikacja 2b**

Otrzymana specyfikacja logiczna dla dziennika zdarzeń repairExample.xes, przy parametrze noise_threshold algorytmu Inductive Miner ustawionym na 1:

```
ForAll((tau) => Exist(Test_Repair))
ForAll((Archive_Repair) => Exist(End))
ForAll(~((Register) ^ (a2_s | a2_e)))
ForAll(~((x3_s) ^ (x3_e)))
ForAll(~((Archive_Repair) ^ (End)))
Exist(Register)
ForAll(~((Analyze_Defect) ^ (Archive_Repair)))
ForAll(~((l_s) ^ (Test_Repair)))
ForAll(~((tau) ^ (x3_e)))
ForAll((Inform_User) => Exist(a2_e))
ForAll(~((Register) ^ (Analyze_Defect)))
ForAll(~((tau) ^ (l_s | Test_Repair)))
ForAll(~((Analyze_Defect) ^ (a2_s | a2_e)))
ForAll((a2_s) => (Exist(Inform_User) ^ Exist(x3_s | x3_e)))
ForAll(~((x3_s) ^ (Repair_Complex)))
ForAll(((tau) | (Repair_Complex) | (l_s | Test_Repair)) =>
Exist(x3_e))
ForAll(~((x3_s) ^ (tau)))
ForAll(~((Register) ^ (Archive_Repair)))
ForAll((x3_s | x3_e) => Exist(a2_e))
ForAll((l_s) => Exist(Test_Repair))
ForAll(~((l_s) ^ (tau)))
ForAll((Test_Repair) => ((Exist(tau) ^ Exist(Test_Repair)) |
(~(Exist(tau)))))
ForAll(~((Test_Repair) ^ (tau)))
ForAll(~((a2_s) ^ ((Inform_User) | (x3_s | x3_e))))
Exist(l_s)
ForAll((Analyze_Defect) => Exist(a2_s | a2_e))
ForAll((x3_s) => ((Exist(tau) ^ ~(Exist(Repair_Complex)) ^
~(Exist(l_s | Test_Repair))) | (~(Exist(tau)) ^
Exist(Repair_Complex) ^ ~(Exist(l_s | Test_Repair)) | (~(Exist(tau))
^ ~(Exist(Repair_Complex)) ^ Exist(l_s | Test_Repair))))
ForAll(~((Repair_Complex) ^ (x3_e)))
ForAll((Register) => Exist(Analyze_Defect))
ForAll(~((tau) ^ (Repair_Complex)))
Exist(a2_s)
Exist(x3_s)
ForAll((a2_s | a2_e) => Exist(Archive_Repair))
ForAll(~((Register) ^ (End)))
ForAll(~((Repair_Complex) ^ (l_s | Test_Repair)))
ForAll(~((x3_s) ^ (l_s | Test_Repair)))
ForAll(~((a2_s | a2_e) ^ (Archive_Repair)))
ForAll(~((Analyze_Defect) ^ (End)))
ForAll(~((a2_s | a2_e) ^ (End)))
ForAll(~(((Inform_User) | (x3_s | x3_e)) ^ (a2_e)))
ForAll(~((l_s | Test_Repair) ^ (x3_e)))
```

Dodatek B

Lista załączników:

- `logi` - folder zawierający dziennik `repairExample.xes` oraz `running-example.xes`.
- `program` - folder zawierający kod źródłowy `projekt_magisterski.ipynb` oraz plik konfiguracyjny `approved_patterns.json`.
- `problemy` - folder zawierający wszystkie problemy zdefiniowane w ramach pracy.
- `konfiguracja.txt` - plik instruujący sposób uruchomienia kodu oraz Proverów.