Generator Formuł Logicznych

Mateusz Ficek, Mateusz Gujda

Studio Projektowe 2

1 Założenia projektu

Głównym założeniem projektu była implementacja algorytmów odpowiedzialnych za generowanie formuł logicznych dla logiki pierwszego rzędu na podstawie artykułu *Pattern-based and composition-driven automatic generation of logical specifications for workflow-oriented software models.*

Dodatkowym założeniem było dodanie obsługi dwóch języków programowania – Java oraz Python.

2 Wymagania

1. Implementacja generatora formuł logicznych w języku Java
2. Implementacja generatora formuł logicznych w języku Python
3. Translacja wzorców z oryginalnej logiki na logikę pierwszego rzędu
4. Testy programu

3 Opis przygotowanego projektu

Cały program składa się z trzech głównych algorytmów:

3.1 Pierwszy algorytm

Pierwszy, odpowiedzialny za nadawanie oznaczeń wzorcom, umożliwia rozpoznawanie, które argumenty należą do poszczególnych wzorców w podanym wyrażeniu.

Przykładowo, po przepuszczeniu przez pierwszy algorytm wyrażenia

$$w = Seq(a, Seq(ConcurRE(b, c, d), ConcurRE(e, f, g)))$$

otrzymamy wyrażenie oznakowane

$$w' = Seq(1]a, Seq(2]ConcurRE(3]b, c, d[3), ConcurRE(3]e, f, g[3)[2)[1)$$

Tak wyrażenie z nadanymi indeksami można poddać następnie dalszej analizie.

3.2 Drugi algorytm

Drugi algorytm odpowiadał za generowanie skonsolidowanego wyrażenia. Jako argumenty algorytm przyjmuje analizowane wyrażenie (w), np. Seq(2]ConcurRE(3]b,c,d[3]), ConcurRE(3]e,f,g[3])[2), typ wyrażenia (t = ini lub fin) oraz predefined pattern property set(Σ), czyli zbiór wzorców, które dopuszczalne są podczas generowania wyrażenia logicznego.

Weźmy podane wyżej wyrażenie

$$Seq(2]Concur(3]b, c, d[3), ConcurRE(3]e, f, g[3)[2)$$

przepuszczając je przez drugi algorytm, wykonane zostaną poniższe czynności:

1. Sprawdzenie jaki typ wyrażenia jest akurat sprawdzany oraz przypisanie do wartości ex odpowiednio wartości ini lub fin, która została odczytana z Pattern Property Set.
2. Ustawienie wartości type zmiennej argType
3. Następnie wykonywana jest pętla po wszystkich argumentach a w wyrażeniu w dla których typ a jest równy wartości zmiennej argType
4. Jeśli a nie jest atomiczna to podmieniane jest a w wyrażeniu ex przez wartość funkcji $ConsEx(a, t, \Sigma)$
5. Po przejściu pętli zwracane jest skonsolidowane wyrażenie ex, które dla podanego wyżej przykładu będzie symbolem g.

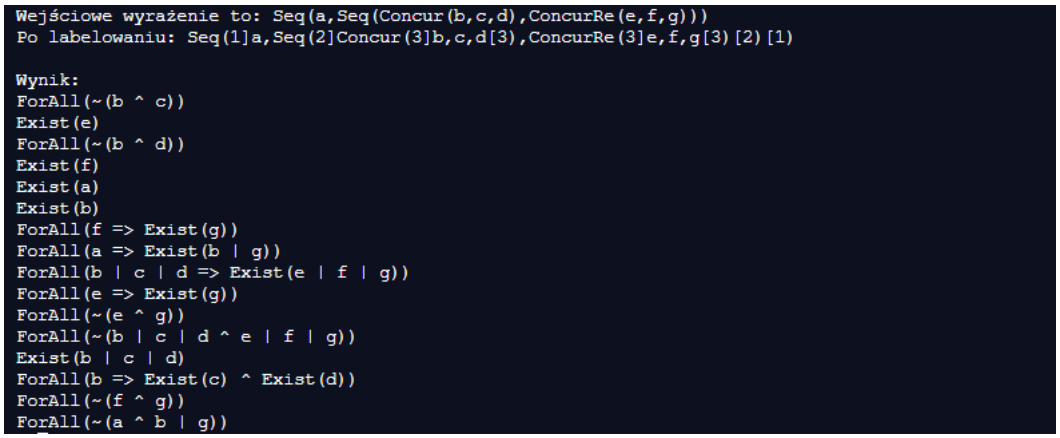
3.3 Trzeci algorytm

Trzeci algorytm jest właściwie główną pętlą odpowiedzialną za tworzenie skonsolidowanych wyrażeń dla poindeksowanego wzorca.

Algorytm właśnie zaczyna się od nadania odpowiednich indeksów poprzez wykonanie pierwszego algorytmu. Następnie wykonywana jest pętla, która rozpoczyna się od największego indeksu do wartości 1. Rozpoczynając wybieramy, zaczynając od lewej strony, wzorce od najwyższego indeksu. Iterując po argumentach wzorca sprawdzane jest czy aktualnie sprawdzany argument jest w atomem, jeśli nie jest, wykonywany jest dwa razy trzeci algorytm dla dwóch typów – ini oraz fin, a później oba wyniki łączone są przy pomocy v.

Tak otrzymane skonsolidowane wyrażenia podmieniają wartości w zbiorze $L2$, który zawiera formuły dla aktualnie sprawdzanego patternu.

Po przejściu przez wszystkie wzorce zwracany jest zbiór wszystkich wygenerowanych formuł logicznych.



Rysunek 1: Przykład działania algorytmów

3.4 Sposób przedstawienia wzorców

W celu łatwiejszego aplikowania wartości do wzorców zostały one zaimplementowane jako obiekty zawierające następujące pola:

1. pattern - regex, który umożliwia przeszukiwanie wzorca
2. identifier - identyfikator wzorca (np. Seq, ConcurRE...)
3. argNumber - liczba argumentów wzorca
4. possibleOutcomes - możliwe wyniki dla wzorca
5. passedArguments - argument podane do wzorca

Przykład gotowego wzorca:

```
PredefinedSetEntry(
            "Seq", 2, new String[]{
                    "arg0",
                    "arg1",
                    "Exist(arg0)",
                    "ForAll(arg0 => Exist(arg1))",
                    "ForAll(~(arg0 ^ arg1))"
            })
```

3.5 Pattern property set

Jednym z wymagań projektu była zamiana wzorców na logikę pierwszego rzędu. Poniżej przedstawiono przekonwertowane patterny:

```
Seq(arg0, arg1) = [arg0, arg1,
                   Exist(arg0),
                   ForAll(arg0 => Exist(arg1)),
                   ForAll(~(arg0 ^ arg1))]

Branch(arg0, arg1, arg2) = [arg0, arg1 | arg2,
                   Exist(arg0),
                   ForAll(arg0 => (Exist(arg1) ^ ~Exist(arg2) |
                       (~Exist(arg1) ^ Exist(arg2))),
                   ForAll(arg0 => Exist(arg1)),
                   ForAll(arg0 => Exist(arg2)),
                   ForAll(~(arg0 ^ arg1)),
                   ForAll(~(arg0 ^ arg2))]

BranchRE(arg0, arg1, arg2) = [arg0 | arg1, arg2,
                   (Exist(arg0) ^ ~Exist(arg1)) |
                       (~Exist(arg0) ^ Exist(arg1)),
                   ForAll(arg0 | arg1 => Exist(arg2)),
                   ForAll(~(arg0 ^ arg2)),
                   ForAll(~(arg1 ^ arg2))]

Concur(arg0, arg1, arg2) = [arg0, arg1 | arg2,
                   Exist(arg0),
                   ForAll(arg0 => Exist(arg1) ^ Exist(arg2)),
                   ForAll(~(arg0 ^ arg1)),
                   ForAll(~(arg0 ^ arg2))]

ConcurRE(arg0, arg1, arg2) = [arg0 | arg1, arg2,
                   Exist(arg0),
                   Exist(arg1),
                   ForAll(arg0 => Exist(arg2)),
                   ForAll(arg1 => Exist(arg2)),
                   ForAll(~(arg0 ^ arg2)),
                   ForAll(~(arg1 ^ arg2))]
```

Sposób implementacji wzorców pozwala nam w łatwy sposób aplikować podane argumenty w odpowiednie miejsca.

3.6 Ini / Fin - arguments

Algorytmy wymagały określenia argumentów ini oraz fin. Na podstawie artykułu wywnioskowano iż niniejsze argumenty to odpowiednio pierwsze oraz drugie

wyrażenie poszczególnych patternów. Przykładowo, dla ConcurRE(a, b, c) ini-argumentem będzie $a|b$, natomiast fin-argumentem c.

3.7 Oznaczenia

Jedyną akcją, którą musi wykonać użytkownik, jest podanie wyrażenia na podstawie którego mają zostać wygenerowane formuły logiczne. Implementacja jest odporna na ilość białych znaków, a prawidłowy format wygląda następująco: $Seq(a, ConcurRE(b, c, d))$. Podawane argumenty powinny być pojedynczymi małymi literami.

Należy również zwrócić uwagę na sposób wpisywania patternów, ponieważ powinny odpowiadać tym podanym w pliku PatternPropertySet.

1. Symbol ~ należy rozumieć jako negację wyrażenia
2. Symbol | oznacza logiczny symbol $\vee$
3. Symbol oznacza logiczne $\wedge$
4. Formuła żywotnościowa ma postać:

$$ForAll(t_1 \geqslant t_0)Exists(t \geqslant t_1) : P(t_1) \Rightarrow Q(t)$$

5. Formuła bezpieczeństwa ma postać:

$$ForAll(t \geqslant t_0) : -B(t)$$

4 Przeprowadzone testy

Algorytmy przetestowano na poniższych przykładach:

1. Łatwy przykład:

$$Seq(a, b)$$

Wynikiem działania programu było:

```
Wejściowe wyrażenie to: Seq(a,b)
Po labelowaniu: Seq(1]a,b[1)

Wynik:
ForAll(~(a ^ b))
ForAll(a => Exist(b))
Exist(a)
```

2. Przykład z trzema zagnieżdżonymi wartościami:

$$ConcurRE(Seq(a,b),Concur(c,d,e),Branch(f,g,h))$$

Wynik:

```
Wejściowe wyrażenie to: ConcurRE(Seq(a,b),Concur(c,d,e),Branch(f,g,h))
Po labelowaniu: ConcurRE(1]Seq(2]a,b[2),Concur(2]c,d,e[2),Branch(2]f,g,h[2)[1)

Wynik:
ForAll(~(a ^ b))
Exist(c)
ForAll(f => (Exist(g) ^ ~Exist(h) | (~Exist(g) ^ Exist(h)))
ForAll(~(a | b ^ f | g | h))
Exist(f)
ForAll(f => Exist(h))
Exist(a)
ForAll(f => Exist(g))
ForAll(c => Exist(d) ^ Exist(e))
Exist(a | b)
Exist(c | d | e)
ForAll(a => Exist(b))
ForAll(~(f ^ h))
ForAll(c | d | e => Exist(f | g | h))
ForAll(~(f ^ g))
ForAll(~(c | d | e ^ f | g | h))
ForAll(~(c ^ d))
ForAll(~(c ^ e))
ForAll(a | b => Exist(f | g | h))
```

3. Przykład z wieloma poziomami zagnieżdżeń:

$$Seq(a, ConcurRE(Branch(b, c, d), BranchRE(Seq(e, f), Seq(g, h), i),$$
$$Seq(Seq(j, k), Seq(ConcurRE(c, g, f), Branch(Branch(Branch(a, b, c), b, c), b, c)))))$$

```
Wynik:
ForAll(j => Exist(k))
Exist(e)
ForAll(g => Exist(h))
Exist(a)
ForAll(~(c | g | f ^ a | b | c))
ForAll(e => Exist(f))
ForAll(~(a ^ c))
ForAll(~(g | h ^ i))
ForAll(c => Exist(f))
ForAll(b => Exist(c))
ForAll(a => Exist(b))
ForAll(~(e | f ^ i))
Exist(a | b | c)
ForAll(b => (Exist(c) ^ ~Exist(d) | (~Exist(c) ^ Exist(d)))
ForAll(~(c ^ f))
ForAll(~(e | g | i ^ j | b | c))
Exist(j)
ForAll(~(b ^ d))
Exist(b)
ForAll(j | k => Exist(c | g | b | c))
Exist(b | c | d)
ForAll(~(b | c | d ^ j | b | c))
ForAll(e | g | i => Exist(j | b | c))
Exist(c | g | f)
ForAll(~(a | b | c ^ b))
ForAll(a | b | c => Exist(c))
Exist(g)
ForAll(~(a ^ b))
ForAll(c | g | f => Exist(a | b | c))
ForAll(b | c | d => Exist(j | b | c))
Exist(c)
ForAll(~(j ^ k))
ForAll(~(e ^ f))
ForAll(b => Exist(d))
ForAll(a | b | c => (Exist(b) ^ ~Exist(c) | (~Exist(b) ^ Exist(c)))
Exist(e | g | i)
ForAll(a => Exist(c))
ForAll(~(j | k ^ c | g | b | c))
ForAll(~(b ^ c))
(Exist(e | f) ^ ~Exist(g | h)) | (~Exist(e | f) ^ Exist(g | h))
ForAll(a => (Exist(b) ^ ~Exist(c) | (~Exist(b) ^ Exist(c)))
ForAll(a => Exist(b | e | g | b | c))
ForAll(e | f | g | h => Exist(i))
ForAll(~(a ^ b | e | g | b | c))
ForAll(g => Exist(f))
Exist(j | k)
ForAll(~(a | b | c ^ c))
ForAll(~(g ^ f))
ForAll(a | b | c => Exist(b))
ForAll(~(g ^ h))
```

4.1 Testy czasu wykonania od złożoności wyrażenia

Algorytmy były testowane na komputerze o poniższej specyfikacji:
4-rdzeniowy procesor i7-7700HQ
16 GB pamięci RAM
Karta graficzna Nvidia GeForce MX150
Programy wykonywane na dysku SSD.

Ilość wyrażeń	Czas wykonania Java (sek)	Czas wykonania Python (sek)
1	0.017	0.002
2	0.03	0.003
3	0.052	0.005
4	0.063	0.006
5	0.055	0.007
10	0.106	0.033

Numer indeksu	Czas wykonania Java (sek)	Czas wykonania Python (sek)
1	0.007	0.002
2	0.01	0.002
3	0.012	0.002
4	0.02	0.003
5	0.026	0.003
6	0.026	0.003
7	0.03	0.004
8	0.037	0.004
9	0.04	0.005

W pierwszej tabeli przedstawiono czasy wykonania progrmau w językach Java oraz Python, gdzie zwiększana była ilość wyrażeń. Natomiast w drugim przypadku zwiększano poziom zagnieżdżenia wyrażeń.

Zagnieżdżenia polegały na dodawaniu nieatomicznych wyrażeń jako argumentów wzorców. Jako przykład podam jedno z wykorzystanych wyrażeń:

$$Seq(a, Seq(a, Seq(a, Seq(a, Seq(a, b)))))$$

Przy takim wyrażeniu poziom zagłębienia to 5. Przetestowaliśmy czas wykonania algorytmów dla zadanych wyrażeń z zagłębieniem od 1 do 9. Wyniki tego badania zostały przedstawione na na rysunku 3..

Dodatkowo przetestowaliśmy wyrażenia, które zwiększały ilość elementów "wszerz". W tym przypadku, nie było to zagłębianie tak jak w wyrażeniu wcześniejszym, lecz dodawaliśmy nowe elementy przy nieznacznym poziomie zagłębienia. A więc maksymalny numer indeksu nie zwiększał aż tyle co w pierwszym przypadku.

Tutaj jako przykład podam wyrażenie:

$$Concur(Seq(a, b), Branch(c, d, e), ConcurRE(f, g, h))$$

gdzie maksymalny indeks ma wartość 2, a liczba wyrażeń nieatomicznych wynosi 4. Wyniki tego badania zostały przedstawione na na rysunku 2..

Pierwszy test polegał na wykonaniu programu dla poniżych wyrażeń:

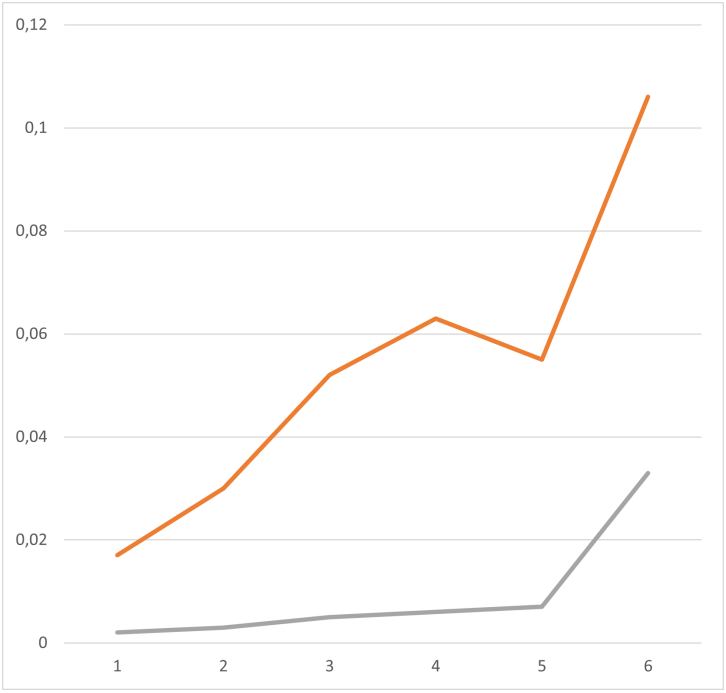
```
Seq(a,b)

Seq(Seq(b,c),d)

Branch(Seq(a,b), Concur(c,d,e), b)

Concur(Seq(a,b), Branch(c,d,e), ConcurRE(f,g,h))

BranchRE(ConcurRE(a,b,Seq(a,b), c, Seq(a,Seq(f,d))))

Branch(Seq(Seq(a,b),b), Seq(c,d), BranchRE(a, Seq(c,f),
                ConcurRE(Seq(a,b),Concur(d,g,e),Seq(c,e))))
```

Natomiast drugi test polegał na wykonaniu wyrażenia Seq(a, Seq(a, ...)) zagnieżdżonego n-razy.

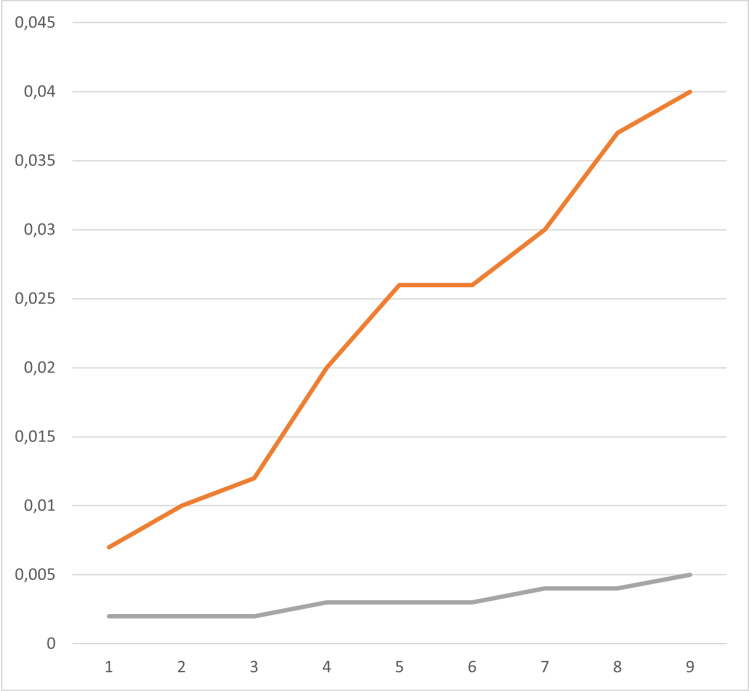
Jak możemy zauważyć, większy wpływ na czas wykonania poszczególnych instrukcji miała ilość wykonanych wzorców oraz ilość ich argumentów niż poziom zagłębienia.

Z niniejszych wykresów możemy wywnioskować, że zwiększanie poziomu zagłębienia oraz zwiększanie ilości wzorców wpływa liniowo na czas generowania formuł logicznych.

Zauważyć również można znaczną różnicę pomiędzy czasem wykonania algorytmów w zależności od języka programowania w którym zaimplementowany został program. Java wykonywała te same przykłady od 4 do nawet 8 razy wolniej niż Python. Różnica w czasie wykonania zapewne wynika z ilości operacji, które wykonywane są w poszczególnych językach. W przypadku Pythona użyliśmy więcej wbudowanych funkcji, podczas gdy w Javie musieliśmy sami dopisać niektóre wspomagające metody. Czas wykonania algorytmów był krótszy w przypadku języka Python, ponieważ lepiej on wspiera operacje wykonywane na listach oraz zmiennych typu String. Właśnie na tych dwóch typach oparta była implementacja algorytmów.



Rysunek 2: Zależność czasu od liczby wyrażeń (kolor pomarańczowy: Java, szary: Python)



Rysunek 3: Zależność czasu od poziomu zagłębienia (kolor pomarańczowy: Java, szary: Python)