AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa magisterska

Implementacja algorytmu weryfikacji modelowej własności LTL w środowisku rozproszonym

Implementation of the distributed LTL model checking algorithm

Autor: *Wojciech Kumoń*
Kierunek studiów: *Informatyka*
Opiekun pracy: *prof. dr hab. Marcin Szpyrka*

Kraków, 2019

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Serdecznie dziękuję . . . tu ciąg dalszych podziękowań np. dla promotora, żony, sąsiada itp.

Spis treści

1. Wstęp

Weryfikacja modelowa to dziedzina umożliwiająca sprawdzenie systemu pod kątem specyfikacji. Operacja taka jest zazwyczaj bardzo wymagająca pod kątem obliczeniowym, co skutkuje długimi czasami wykonania. Przeciwdziałać temu można na kilka sposobów, np. stosując uproszczenie modelu, czy wykorzystując wydajniejszy procesor. Kolejna możliwość to stworzenie skalowanego systemu rozproszonego i właśnie ta metoda zostanie rozważona.

Samą specyfikację wyrazić można na wiele sposobów. W pracy wykorzystana zostanie logika LTL (ang. *linear-time temporal logic*).

1.1. Cel pracy

Celem pracy jest implementacja rozproszonego algorytmu weryfikacji modelowej w oparciu o własności logiczne czasu liniowego - LTL dla języka Alvis z wykorzystaniem frameworka Spring.

1.2. Struktura pracy

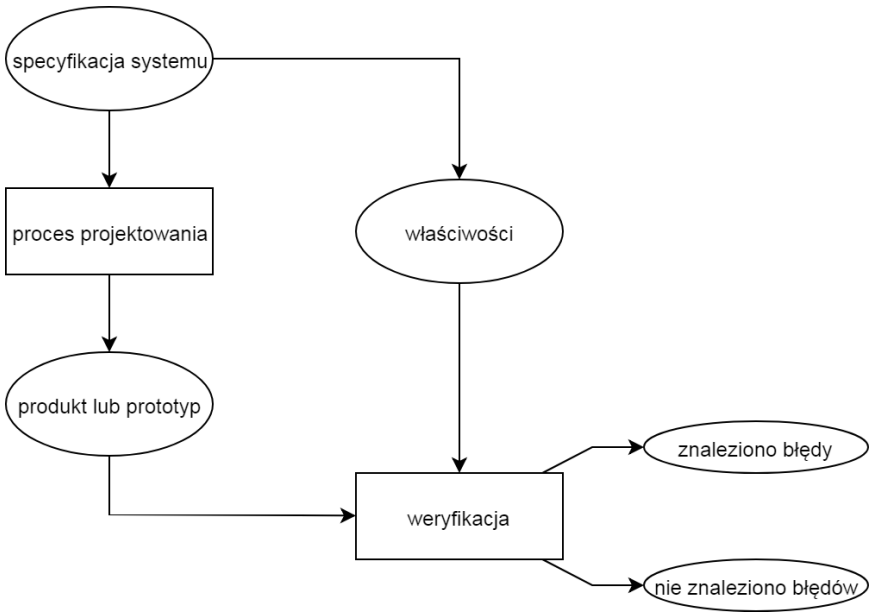
W następnym rozdziale pracy omówione zostanie zagadnienie weryfikacji modelowej w kontekście tematu pracy. Rozdział trzeci zawiera ogólny opis architektury stworzonego systemu. W czwartym rozdziale znajduja się szczegóły dotyczące zastosowanego algorytmu. Piąty rozdział zawiera prezentację rozproszonej implementacji rozwiązania. W ostatnim rozdziale umieszczone jest podsumowanie.

2. Weryfikacja modelowa

Systemy tworzone przez ludzi są coraz bardziej złożone oraz zwiększa się ich rola w życiu każdego z nas. Błędy w oprogramowaniu skutkują stratami finansowymi, wizerunkowymi, opóźnieniami, a także utratą zdrowia i życia ludzi. Dowodzą temu następujące przykłady: nieudany start Ariane-5 (04.06.1996), błąd w procesorach Pentium II Intela, czy źle działająca maszyna do radioterapii, która spowodowała śmierć sześciu pacjentów w latach 1985-1987.

2.1. Weryfikacja systemu

Weryfikacja systemu ma na celu ustalenie, czy projekt posiada oczekiwane właściwości. Mogą one być dość podstawowe (np. nigdy nie dojdzie do zakleszczenia) lub związane z domeną (np. nie można wypłacić więcej pieniędzy, niż jest na koncie). Specyfikacja dostarcza informacji, jak system może oraz jak nie może się zachowywać. Oprogramowanie uważa się za poprawne, jeśli spełnia wszystkie właściwości. Schemat weryfikacji został przedstawiony na rys. 2.1.



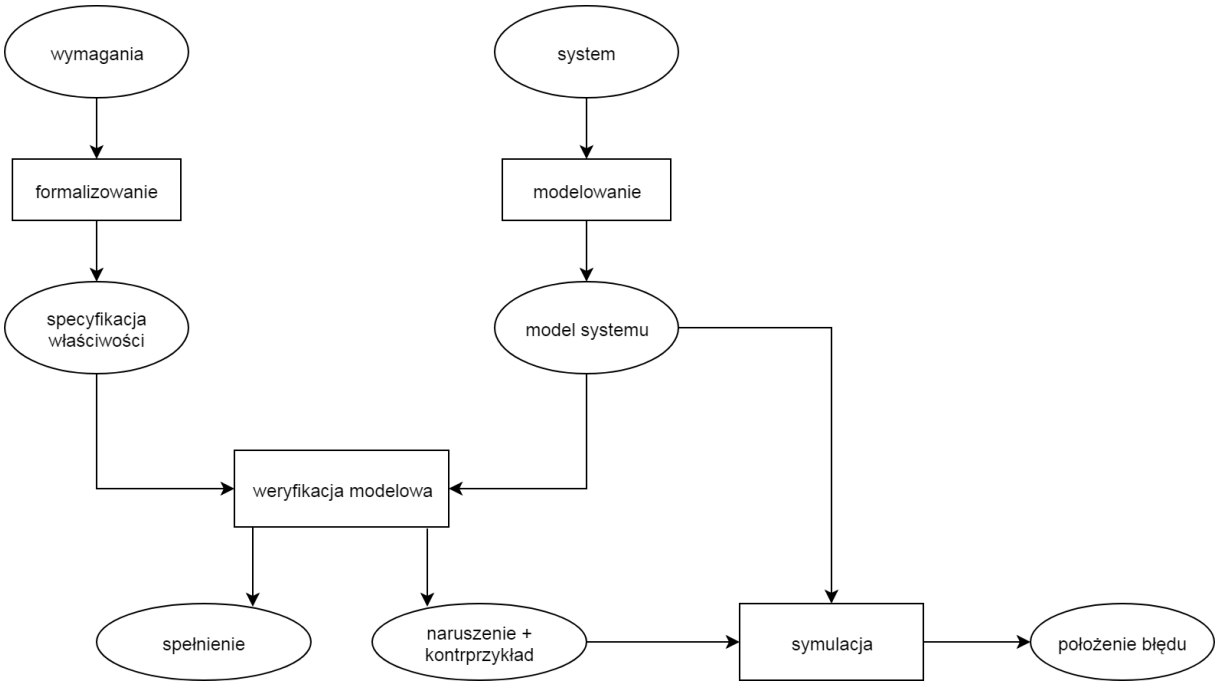
Rys. 2.1. Schemat tworzenia systemu wraz z jego weryfikacją (źródło [1]).

Podstawową formą radzenia sobie z tym problemem jest testowanie oprogramowania (testy jednostkowe, integracyjne, systemowe itp.). Polega ono na uruchamianiu kodu dla różnych ścieżek wykonania, po czym porównuje się ich wyjścia z oczekiwanymi. Niestety, przetestowanie wszystkiego okazuje się praktycznie niemożliwe, zwykle sprawdzane są jedynie warunki brzegowe (stanowi to mały podzbiór wszystkich kombinacji).

2.2. Weryfikacja modelowa

Podczas tworzenia skomplikowanych systemów, kładzie się coraz większy nacisk na testowanie poprawności oprogramowania. Metody formalne mają duży potencjał na tym polu. Ich wczesna integracja (podczas procesu projektowania) dostarcza efektywnych technik weryfikacji. Intuicyjnie, metody formalne można rozważać jako matematykę stosowaną dla modelowania i analizy systemów informatycznych. Zapewniają one poprawność z matematyczną dokładnością.

Techniki weryfikacji bazujące na modelu opisują zachowanie systemu deterministycznie i kompletnie. Samo tworzenie pełnego modelu może wykryć luki lub niespójności. Po jego stworzeniu, wraz z otaczającymi algorytmami, następuje eksploracja stanów systemu. Dzieje się to w podejściu "brute-force" - przejrzane zostają wszystkie możliwe scenariusze. W ten sposób udowadnia się spełnialność właściwości. Weryfikacji dotyczy również kolejność zdarzeń w czasie, np. czy system zawsze odpowie na żądania klienta w tym samym porządku, w jakim je otrzymał.



Rys. 2.2. Schemat podejścia weryfikacji modelowej (źródło [1]).

Podejście to świetnie sprawdza się w wykrywaniu (częstych) błędów związanych z wielowątkowością. Typowe oczekiwane właściwości:

– osiągalność (niemożliwe jest zakleszczenie)

– bezpieczeństwo (coś niepożądanego nigdy nie wystąpi [2])

– żywotność (coś "dobrego"w końcu nastąpi [3])

– uczciwość (czy przy odpowiednich warunkach zdarzenie występuje powtarzalnie)

– właściwości czasu rzeczywistego

Model systemu zazwyczaj generuje się automatycznie z opisu w odpowiednim języku lub dialekcie wspieranym przez narzędzie. Weryfikator modelowy przeszukuje kolejne stany. Następnie sprawdzane są pod kątem właściwości. Po znalezieniu naruszenia, prezentowany zostaje kontrprzykład wraz z całą ścieżką wykonania, która do niego prowadzi. Pozwala to łatwo odtworzyć całą ścieżkę, a także znaleźć niespójność, która wymaga zmiany modelu (lub właściwości) - schemat na rys. 2.2.

To nie jedyne zalety, kolejnymi mocnymi stronami weryfikacji modelowej są:

– To ogólna metoda weryfikacji, która sprawdza się zarówno przy tworzeniu oprogramowania, jak i projektowaniu sprzętu (np. procesorów).

– Wspiera częściową weryfikację - możemy sprawdzać poszczególne właściwości niezależnie, nawet gdy pełna specyfikacja nie jest gotowa.

– Wszystkie możliwości zostają sprawdzone.

– Dostarcza informacji diagnostycznych po wykryciu niespełnionej właściwości.

– Uruchomienie weryfikatora nie wymaga ekspertyzy na tym polu.

– Łatwo zintegrować to rozwiązanie z cyklem wytwarzania oprogramowania.

– Obecnie wzrasta zainteresowania tym podejściem.

Oczywiście nie brak także wad:

– Weryfikowany jest model systemu, a nie sam system. Brak tu gwarancji, że implementacja poprawnie go odtwarza.

– Sprawdzane są tylko wykorzystane wymagania. Nie można wnioskować o poprawności właściwości, które nie zostały sprawdzone (błąd niedokładnej specyfikacji [4]).

– Aplikacje oparte na dużej ilości danych mogą posiadać zbyt wiele stanów, aby je wszystkie wygenerować, a nawet ich liczba może być nieskończona.

– Podatność na problem eksplozji przestrzeni stanów [5].

2.3. Logika LTL

Logika LTL to jednak z logik temporalnych. Opiera się na liniowej strukturze czasu. Jej składnia i semantyka pozwala precyzyjnie opisywać bogate spektrum właściwości systemu, włączając w to m.in bezpieczeństwo czy żywotność [6]. "Temporalna" oznacza umiejscawianie zdarzeń relatywnie względem innych, to pewna abstrakcja ponad czasem. Z tego powodu niewykonalne jest sprawdzenie, czy maksymalne opóźnienie między zdarzeniami wynosi $500ms$.

Formuły LTL ponad zbiorem AP wyrażeń atomowych przedstawia poniższa gramatyka:

$$\varphi ::= true \mid false \mid a \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \mid \varphi \Leftrightarrow \psi \mid$$
$$\mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \varphi\mathbf{U}\psi \mid (\varphi)$$

gdzie $a \in AP$
Wyjaśnienie symboli (intuicyjna semantyka przedstawiona jest na rys. 2.3):
X - $ne\mathbf{X}t$ - w następnym stanie
U - $\mathbf{U}ntil$ - aż do pewnego momentu w przyszłości
F - $\mathbf{F}inally$ - teraz lub w przyszłości
G - $\mathbf{G}lobally$ - teraz i w każdym momencie w przyszłości

Przykładowe formuły LTL:

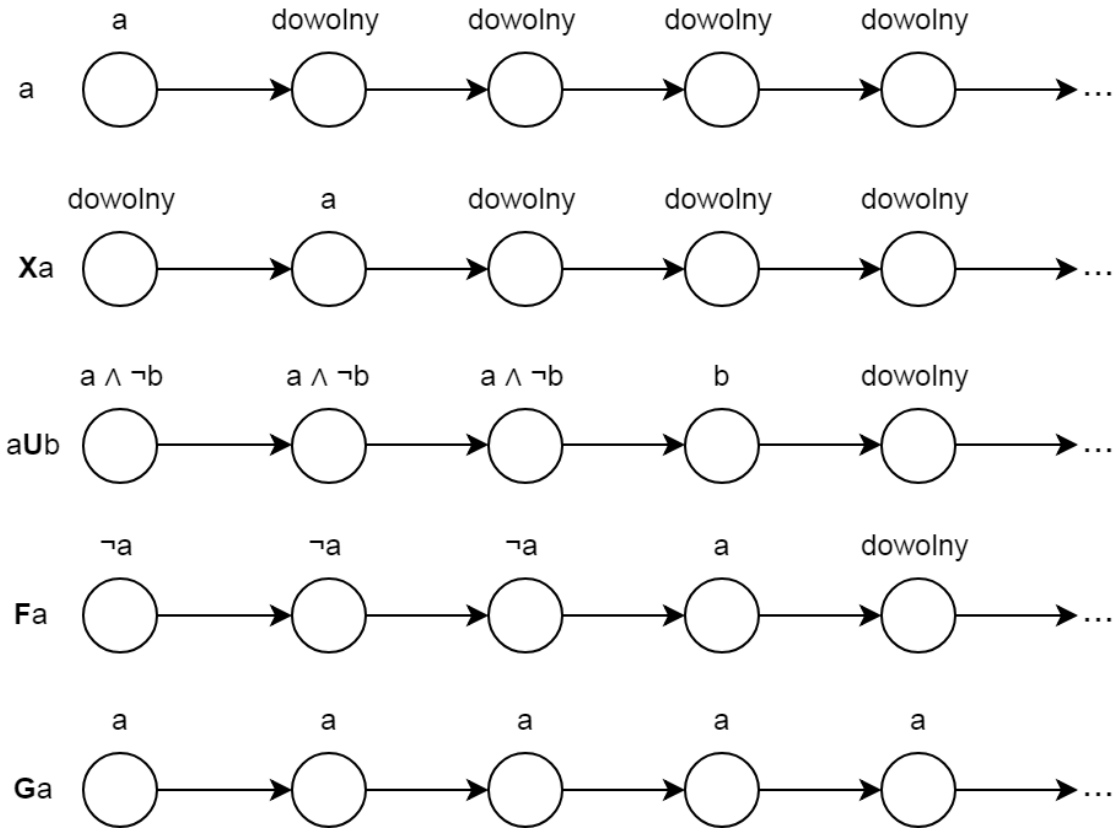
– $\mathbf{G}\neg\varphi$ - niemożliwe jest osiągnięcie stanu posiadającego φ (bezpieczeństwo)

– $\mathbf{G}(\varphi \Rightarrow \mathbf{F}\psi)$ - dla dowolnego stanu, jeśli posiada on φ, w końcu znajdziemy stan posiadający ψ (żywotność)

– $\mathbf{FG}\varphi$ - w końcu znajdziemy taki stan, od które zawsze spełnione będzie φ (stabilność)

2.4. Automat Büchiego

W poprzedniej sekcji przedstawiona została liniowa logika LTL. Niestety w obecnej postaci nie można jej wykorzystać do weryfikacji modelowej. Punktem startowym jest system tranzycyjny TS oraz formuła LTL φ, która formalizuje wymaganie dla TS. Zadanie polega na sprawdzeniu, czy $TS \models \varphi$. Jeśli φ jest naruszona, szczegóły błędy powinny zostać dostarczone w celu jego naprawienia. Manualne dowodzenie $TS \models \varphi$ to zazwyczaj bardzo trudny proces (zwykle systemy tranzycyjne są ogromne). Dodatkowo rzadkim przypadkiem jest jedna formuła do weryfikacji. Wymagania składają się na ich zbiór,



Rys. 2.3. Intuicyjna semantyka temporalnych modalności (źródło [1]).

taki jak $\varphi_1, ..., \varphi_k$. Wynikają z tego dwie opcje - można połączyć je w jedną $\varphi_1 \wedge ... \wedge \varphi_k$, aby uzyskać specyfikację wszystkich wymagań naraz lub traktować każde wymaganie φ_i niezależnie. Drugie podejście często okazuje się wydajniejsze. Ponadto znalezienie błędu podczas sprawdzania pełnej specyfikacji nie dostarcza tak precyzyjnych danych diagnostycznych.

Pomiędzy logiką temporalną a teorią ω-regularnych języków zachodzi bliska relacja [7][8]. Języki ω-regularne są analogiczne do regularnych, lecz są zdefiniowane na nieskończonych słowach (zamiast skończonych). Rozpoznawane są one przez automat Büchiego [9]. To skończony automat, który operuje na nieskończonych słowach. Nieskończone słowo jest akceptowane przez automat Büchiego wtedy i tylko wtedy, gdy stan akceptujący wystąpi nieskończenie wiele razy [10]. Formalnie, deterministyczny automat Büchiego to krotka $A = (Q, \Sigma, \delta, q_0, F)$ składająca się z następujących elementów:

- Q - skończony zbiór. Elementy Q nazywane są stanami A.
- Σ - skończony zbiór nazywany alfabetem.
- $\delta : Q \times \Sigma \to Q$ - funkcja nazywana funkcją tranzycji A.
- q_0 - element Q, nazywany stanem początkowym A
- $F \subseteq Q$ - warunek akceptujący. A akceptuje dokładnie takie przejścia, gdzie przynajmniej jeden z nieskończenie często występujących stanów jest w F.

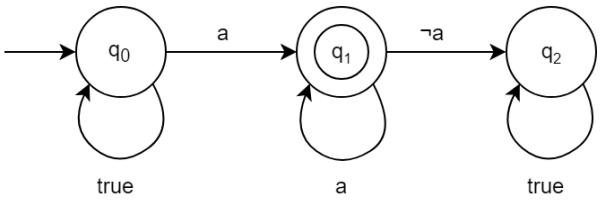
W niedeterministycznym automacie Büchiego (NBA - ang. *non-deterministic Büchi automaton*), funkcja tranzycji δ zastąpiona jest relacją tranzycji Δ, która zwraca zbiór stanów, a w miejscu stanu początkowego q_0 znajduje się zbiór stanów początkowych.

Uogólniony automat Büchiego (GBA - ang. *generalized Büchi automaton*), to krotka $A = (Q, \Sigma, L, \Delta, Q_0, F)$:

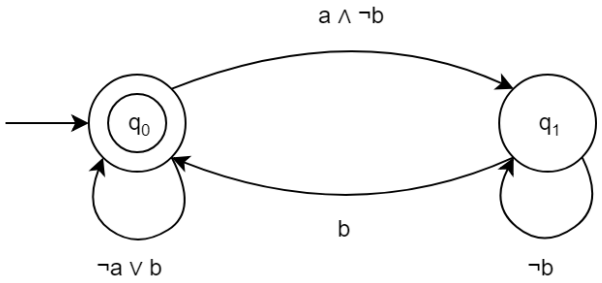
- Q - skończony zbiór. Elementy Q nazywane są stanami A.
- Σ - skończony zbiór nazywany alfabetem.
- $\delta : Q \times \Sigma \to 2^Q$ - funkcja nazywana funkcją tranzycji A.
- Q_0 - podzbiór Q, nazywany stanami początkowymi
- F - warunek akceptujący. Składa się z zera lub więcej akceptujących zbiorów. Każdy taki zbiór $F_i \in F$ jest podzbiorem Q.

Pomimo różnic, GBA to ekwiwalent NBA w kategorii siły ekspresji. W związku z tym możliwa jest jego konwersja do NBA [11][12].

Na rys. 2.4 oraz 2.5 znajdują się przykładowe NBA, wraz z odpowiadającymi im formułami.



Rys. 2.4. NBA dla $\mathbf{FG}a$ (źródło [1]).



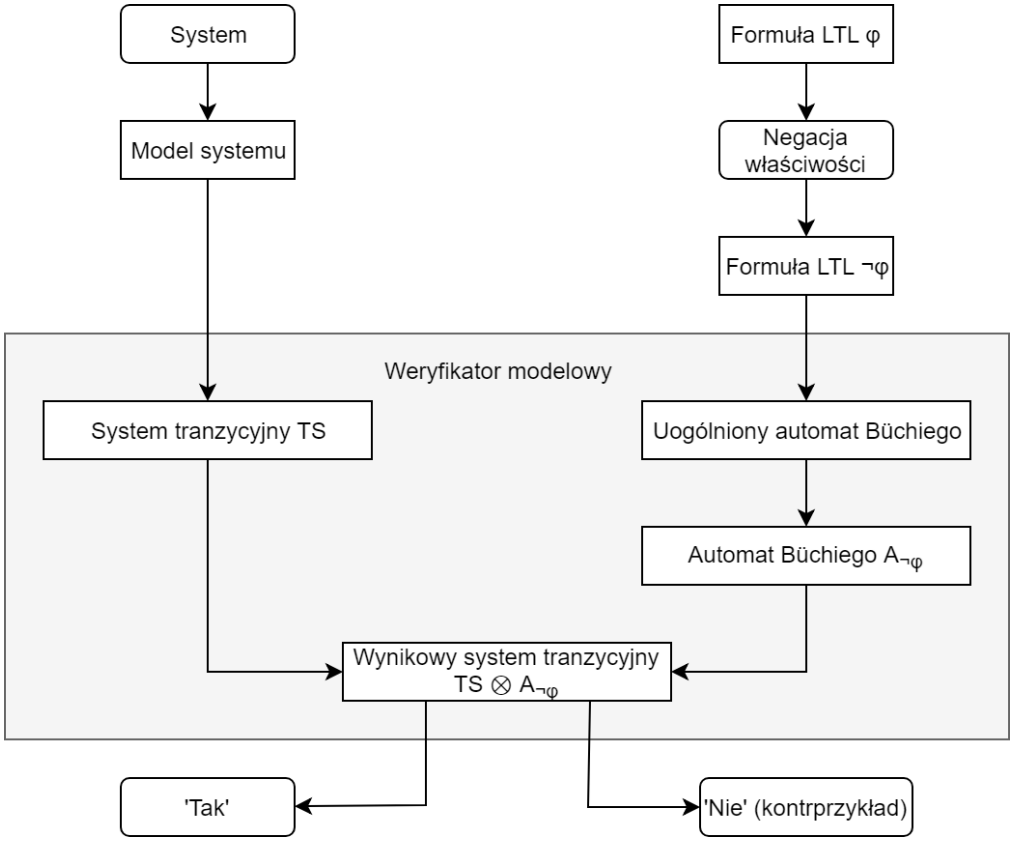
Rys. 2.5. NBA dla $\mathbf{G}(a \Rightarrow \mathbf{F}b)$ (źródło [1]).

Algorytm weryfikacji formuły LTL φ w systemie tranzycyjnym TS wygląda następująco:

1. Zaneguj formułę φ otrzymując $\neg\varphi$.
2. Skonstruuj niedeterministyczny automat Büchiego $A_{\neg\varphi}$ odpowiadający formule $\neg\varphi$.
3. Skonstruuj wynikowy system tranzycyjny $TS \otimes A$.

4. Jeśli istnieje ścieżka π w $TS \otimes A$ spełniająca akceptujący warunek A, zwróć “nie“ wraz z opisem ścieżki π, w przeciwnym razie zwróć “tak“.

Sam schemat przedstawiający jak działa weryfikacja modelowa, kiedy wykorzystujemy logikę LTL do opisu właściwości, znajduje się na rys. 2.6.



Rys. 2.6. Przegląd weryfikacji modelowej LTL (źródło [1]).

2.5. Eksploracja stanów systemu

Posiadanie modelu systemu nie oznacza dostępności pełnego grafu stanów. Może to być niepraktyczne lub nawet niemożliwe ze względu na liczbę stanów. Preferowanym podejście to generowanie kolejnych stanów i ich weryfikacja ich w locie. Takie rozwiązanie pozwala na wykrycie niespójności we wczesnym etapie, unikając generacji całej przestrzeni stanów.

Kolejnym czynnikiem wpływającym na wydajność jest sam algorytm budujący graf stanów. W przypadku skomplikowanych systemów spodziewać się można olbrzymiego grafu, więc również czas jego tworzenia będzie odzwierciedlał rozmiar. Moc obliczeniowa jednego procesora może okazać się niewystarczająca. Warto więc wykorzystać algorytmy równoległe. Standardowym rozwiązanie to DFS (przeszukiwanie w głąb - ang. *Depth-first search*) [13][14]. Występuje w wielu wersjach mających zwiększyć jego wydajność. Z czasem powstała też nowa grupa algorytmów. Opiera się na podziale grafu na silnie

spójne składowe, wywodzi się ona z algorytmu Tarjana [15]. Złożoność obliczeniowa tych algorytmów jest liniowa $O(m+n)$, gdzie m to liczba krawędzi, a n oznacza ilość stanów. Wszystkie korzystają z tej samej zasady eksploracji - przeszukiwania wstecznego (ang. *post-order*). Faktem jest, że problem takiego przeszukiwania jest P-zupełny, więc skalowany, równoległy algorytm tego typu najprawdopodobniej nie istnieje [16].

Algorytmem wartym uwagi jest *on-the-fly OWCTY* [17]. Opiera on się na algorytmie OWCTY (ang. *One Way Catch Them Young*) [18] oraz MAP (ang. *Maximal Accepting Predecessor*) [19]. Co więcej, pozwala na zrównoleglenie obliczeń.

OWCTY nie polega na przeszukiwaniu wstecznym DFS, dzięki czemu wykonalne jest jego równoległe wykonanie. Niestety procedura sortowania topologicznego nie wykrywa cykli akceptujących wprost. Algorytm skupia się na eliminowaniu cykli nieakceptujących. Wyliczany i przechowywany jest zbiór stanów poprzedzających akceptujący cykl. Jeśli po zakończeniu algorytmu zbiór ten okaże się pusty, oznacza to, że nie ma cyklu akceptującego. Całość składa się z dwóch faz, które wykonywane są w pętli, dopóki nie osiągnie się stałego punktu (kolejne wykonanie faz nic nie zmienia).

Algorytm MAP bazuje na propagacji maksimum akceptujących poprzedników i podobnie do OWCTY, jego wykonanie polega na wielu przejściach. Każde z nich w pełni przekazuje maksimum poprzedników do wszystkich stanów. Obliczenie poprawnej wartości maksymalnego poprzednika dla wierzchołka wymaga czasem przesłania nowej (większej) wartości poprzez krawędź, która była już wykorzystana. Zjawisko to określa się jako repropagacja. Po znalezieniu wierzchołka, który jest swoim maksymalnym poprzednikiem (czy wykryciu cyklu akceptującego), algorytm natychmiast się kończy i wskazuje kontrprzykład.

Algorytm on-the-fly OWCTY czerpie z dwóch powyższych. Wykorzystuje heurystykę opartą na algorytmie MAP, która w wielu przypadkach wykryje akceptujący cykl i zakończy procedurę. W przeciwnym wypadku działanie nie zajdzie w locie, a wygeneruje cały graf stanów zgodnie z instrukcjami OWCTY. Takie połączenie pozwala skorzystać z zalet obu metod - weryfikacji w locie, liniowym czasie wykonania, czy możliwości zrównoleglenia.

Algorytmy służące do detekcji cykli akceptujących klasyfikuje się ze względu na zdolność do wczesnego zakończenia. Pozwala to uniknąć eksploracji wszystkich stanów, co dla dużych systemów może okazać się nawet niemożliwe. Określa się je jako:

– Algorytm działający w locie poziomu 0, jeśli graf automatu zawiera błąd, dla którego algorytm nigdy nie zakończy się wcześniej.

– Algorytm działający w locie poziomu 1, jeśli dla wszystkich grafów automatów zawierających błąd, algorytm może zakończyć się wcześniej, ale nie jest to zagwarantowane.

– Algorytm działający w locie poziomu 2, jeśli dla wszystkich grafów automatów zawierających błąd, algorytm gwarantuje wczesne zakończenie.

Warto zauważyć, że algorytmy poziomu zerowego bywają także rozważane jako niedziałające w locie. Poziom pierwszy oznacza, że w zależności od danych wejściowych algorytm zakończy swoje działanie

szybciej, jednak zdarzają się też sytuacje, które wymagają całej przestrzeni stanów, aby wykryć błąd. Przykładowy algorytm poziomu 0 to OWCTY, poziomu 1 on-the-fly OWCTY, poziomu 2 MAP.

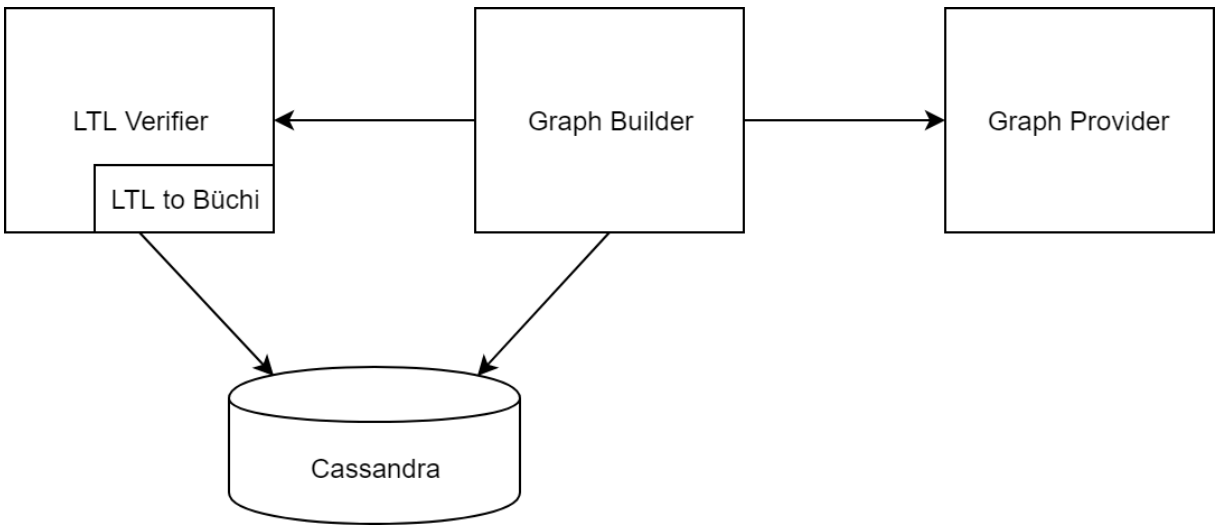
3. Architektura systemu

Założeniem projektu było stworzenie algorytmu weryfikacji modelowej w oparciu o własności LTL dla języka Alvis w środowisku rozproszonym.

Alvis to język formalny, którego celem jest dostarczenie elastyczności w modelowaniu systemów współbieżnych i czasu rzeczywistego wraz z możliwością weryfikacji opartej o metody formalne. Stanowi połączenie zalet języków wysokiego poziomu z graficznym budowaniem zależności między podsystemami (nazywanymi agentami).

3.1. System rozproszony

Architektura systemu została zaplanowana tak, aby wydzielić funkcjonalności do odseparowanych aplikacji. Całość składa się z 3 programów oraz bazy danych. Jej schemat ogólny został przedstawiony na rys. 3.1.



Rys. 3.1. Schemat ogólny systemu.

Pierwszym elementem jest *Graph Provider* (**GP**). Spełnia on dwa zadania. Pierwsze z nich to dostarczanie wszystkich osiągalnych tranzycji dla danego stanu. Drugie umożliwia otrzymanie stanów dla

zadanej tranzycji, które można odwiedzić z wybranego stanu źródłowego. Serwis ten enkapsuluje działanie całej domeny. Jako jedyny dostarcza dane, dzięki którym da się zbudować całą przestrzeń stanów. GP spełnia prosty interfejs przedstawiony w listingu 3.1.

```
public interface GraphProvider {
    Collection<Transition> allTransitionFromState(SystemState from);

    Collection<SystemState> allReachableStates(SystemState from,
                                               Transition through);
}
```

Listing 3.1. Interfejs implementowany przez GP.

LTL Verifier (**LV**) to główna aplikacja zajmująca się weryfikacją modelową i zawiera główną część tego algorytmu. Konwertuje ona także formuły LTL do automatów Büchiego. Interfejs implementowany przez LV opisuje listing 3.2.

```
public interface LtlVerifier {
    public VerificationJobId createVerificationJob(List<State> initialStates);

    public VerificationResult newStates(NewStates newStates,
                                        VerificationJobId id);

    public VerificationResult finish(VerificationJobId id);
}
```

Listing 3.2. Interfejs implementowany przez LV.

Graph Builder (**GB**) spełnia rolę serca systemu. To centrum sterowania, które inicjuje zapytania do pozostałych aplikacji. Zarządza eksploracją kolejnych stanów, a także wysyła je do weryfikacji. Zarówno GB jak i LV komunikują się z bazą danych (Cassandra), aby czytać i zapisywać stany czy tranzycje. Komunikacja między GB a GP zachodzi za pomocą Apache Thrift, wykorzystując binarny protokół przesyłając dane siecią. Zapytania GB do LV realizowane są poprzez protokół HTTP w formacie JSON.

4. Algorytm weryfikacji

Zastosowany algorytm weryfikacji modelowej opiera się głównie na *on-the-fly OWCTY* [17], który został dostosowany do wykorzystania w środowisku rozproszonym.

Algorithm 1 $detectAcceptingCycle()$

Require: $G = (V, E, ACC)$

1: $initialStates \leftarrow getInitialStates()$
2: $approximationSet \leftarrow initialise(initialStates)$
3: $oldSize \leftarrow \infty$
4: **while** $(approximationSet.size \neq oldSize)$ **and** $(approximationSet.size > 0)$ **do**
5: $\quad oldSize \leftarrow approximationSet.size$
6: $\quad eliminateNoAccepting(approximationSet)$
7: $\quad eliminateNoPredecessors(approximationSet)$
8: **end while**
9: **return** $approximationSet.size > 0$

Algorytm OWCTY wykorzystuje sortowanie topologiczne dla detekcji cykli. Zapewnia liniową złożoność czasową przy jednoczesnym uniknięciu przeszukiwania w głąb, co umożliwia wykonanie równoległe. Niestety procedura sortowania topologicznego nie może bezpośrednio wykryć cykli akceptujących. Zamiast tego wykorzystuje się eliminację cykli nieakceptujących. Obliczany jest zbiór stanów poprzedzanych przez cykl akceptujący (*approximationSet*). Jeśli po zakończeniu algorytmu zbiór ten jest pusty, nie ma cyklu akceptującego. Sam zbiór wylicza się w kilku fazach. Pierwsza z nich, czyli *initialise()* (algorytm 2) eksploruje pełną przestrzeń stanów systemu oraz przygotowuje niezbędne dane dla kolejnych faz. Kolejne dwie usuwają ze zbioru stany, które nie mogą być częścia cyklu akceptującego.

Jedna z nich to *eliminateNoAccepting()* (algorytm 3). Zaczyna się od pozostawienia w zbiorze jedynie stanów akceptujących (linie 3-10). Następnie obliczane są wartości liczby poprzedników dla każdego wierzchołka. To część procedury osiągalności, która poszerza cały zbiór, a ten może zawierać już nie tylko stany akceptujące (linie 11-22).

Ostatnia faza OWCTY to *eliminateNoPredecessors()* (algorytm 4). Bazuje ona na sortowaniu topologicznym. Wykorzystuje liczbę poprzedników wyliczonych w *eliminateNoAccepting()*, aby iteracyjnie

usuwać wierzchołki ze zbioru, kiedy ich stopień (w podgrafie tworzonym przez stany pozostałe w zbiorze) równy jest 0. Kiedy wierzchołek znika ze zbioru, należy zmniejszyć stopień jego następników o 1. Brak takich następników oznacza zakończenie fazy. *EliminateNoAccepting()* oraz *eliminateNoPredecessors()* wykonywane są w pętli, dopóki zachodzą jakieś zmiany.
Opis użytych zmiennych/procedur w algorytmach 1 i 2:

- $G = (V, E, ACC)$ - wejściowy graf składający się ze zbiorów wierzchołków, krawędzi i stanów akceptujących
- *getInitialStates()* - procedura zwracająca zbiór stanów początkowych dla grafu G
- *isAccepting(x)* - zwraca prawdę, jeśli stan x jest akceptujący, fałsz w przeciwnym wypadku
- *acceptingCycleFound()* - służy do wcześniejszego zakończenia algorytmu (kiedy cykl akceptujący został wykryty bez eksploracji całej przestrzeni stanów)

Algorithm 2 $initialise(initialStates)$

Require: $initialStates$

1: $approximationSet \leftarrow initialStates$
2: $q \leftarrow new\ Queue()$
3: $q.pushBack(initialStates)$
4: **while** $q.isNotEmpty()$ **do**
5: $s \leftarrow q.popFront()$
6: **for all** $t \in getSuccessors(s)$ **do**
7: **if** $t \notin approximationSet$ **then**
8: $approximationSet.add(t)$
9: $q.pushBack(t)$
10: **end if**
11: **if** $isAccepting(t)$ **then**
12: **if** $(t == s)$ **or** $(approximationSet.getMap(s) == t)$ **then**
13: $acceptingCycleFound()$
14: **return**
15: **end if**
16: $approximationSet.setMap(t, max(t, approximationSet.getMap(s)))$
17: **else**
18: $approximationSet.setMap(t, approximationSet.getMap(s))$
19: **end if**
20: **end for**
21: **end while**
22: **return** $approximationSet$

4.1. Heurystyka

To zaaplikowanie heurystyki w fazie inicjowania (funkcja *initialise()* w algorytmie 2) modyfikuje oryginalny OWCTY. Jedyna różnica to linie 11-19, które wykorzystują pomysł z algorytmu MAP. Propaguje się jednego akceptującego poprzednika przez wszystkie nowo odkryte krawędzie. Jeśli stan akceptujący zostanie przekazany do samego siebie, oznacza to wykrycie cyklu akceptującego, a obliczenia zostają przerwane (linia 13). Zgodnie z działaniem algorytmu MAP, na stan akceptujący do rozpropagowania wybiera się ten maksymalny spośród stanów akceptujących odwiedzonych na ścieżce ze stanu początkowego do obecnego.

Algorithm 3 $eliminateNoAccepting(approximationSet)$

Require: $approximationSet$

1: $tmpApproximationSet \leftarrow \emptyset$
2: $q \leftarrow new\ Queue()$
3: **for all** $s \in approximationSet$ **do**
4: **if** $isAccepting(s)$ **then**
5: $q.pushBack(s)$
6: $tmpApproximationSet.add(s)$
7: $tmpApproximationSet.setPredecessorCount(s, 0)$
8: **end if**
9: **end for**
10: $approximationSet \leftarrow tmpApproximationSet$
11: **while** $q.isNotEmpty()$ **do**
12: $s \leftarrow q.popFront()$
13: **for all** $t \in getSuccessors(s)$ **do**
14: **if** $t \in approximationSet$ **then**
15: $approximationSet.incrementPredecessorCount(t)$
16: **else**
17: $q.pushBack(t)$
18: $approximationSet.add(t)$
19: $approximationSet.setPredecessorCount(t, 0)$
20: **end if**
21: **end for**
22: **end while**

Faza inicjacji algorytmu OWCTY wymaga eksploracji całej przestrzeni stanów, więc została użyta do wykonania detekcji cykli, wykorzystując propagację maksymalnego akceptującego stanu. W przeciwieństwie do algorytmu MAP brakuje to repropagacji, aby złożoność obliczeniowa pozostała i liniowa

i była proporcjonalna do rozmiaru grafu. Skutek tego ograniczenia to brak wykrywalności wszystkich cykli (stąd to heurystyka).
Wyróżnić można dwa podstawowe powody, kiedy metoda ta będzie nieskuteczna (pominie cykl):

1. Maksymalny akceptujący poprzednik nie leży wewnątrz cyklu.
2. Wartość maksymalnego akceptującego poprzednika nie wróci do źródła, mimo że cykl istnieje.

Algorithm 4 $eliminateNoPredecessors(approximationSet)$

Require: $approximationSet$

```
1:  q ← new Queue()
2:  for all s ∈ approximationSet do
3:      if approximationSet.getPredecessorCount(s) == 0 then
4:          q.pushBack(s)
5:      end if
6:  end for
7:  while q.isNotEmpty() do
8:      s ← q.popFront()
9:      approximationSet.remove(s)
10:     for all t ∈ getSuccessors(s) do
11:         approximationSet.decrementPredecessorCount(t)
12:         if approximationSet.getPredecessorCount(t) == 0 then
13:             q.pushBack(t)
14:         end if
15:     end for
16: end while
```

Pierwszy przypadek obsługiwany jest w oryginalnym algorytmie MAP poprzez iteracyjne usuwanie stanów akceptujących, co wymaga dodatkowej liczby przejść o liniowej złożoności. Drugi przypadek obsługuje repropagacja, która również nie mogła zostać zawarta ze względu na podniesienie złożoności obliczeniowej.

W sytuacji, gdy żadne z powyższych nie zachodzi, akceptujący cykl zostanie wykryty. Przykłady z rys. 4.1:

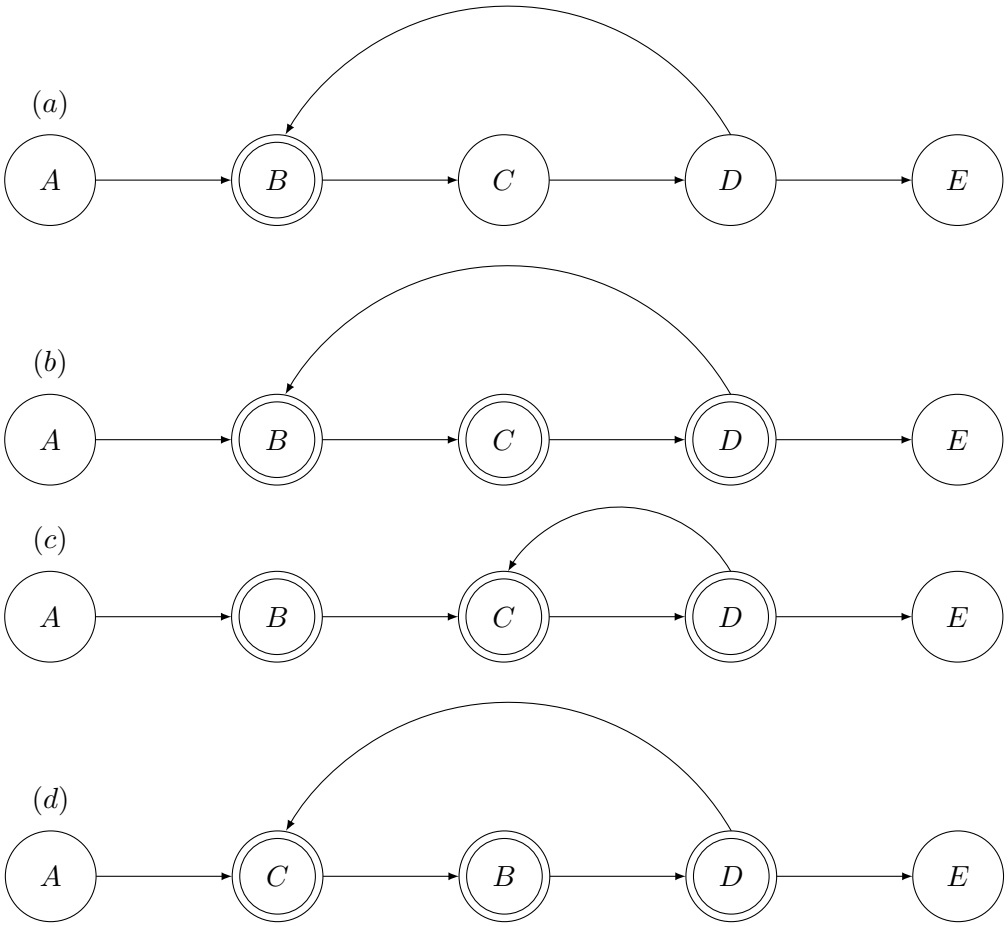
(a) Przypadek trywialny (jeden stan akceptujący). Wartość B zostanie rozpropagowana do C i D. W wyniku tego B powróci do węzła źródłowego, więc cykl zostanie wykryty.

(b) W tej sytuacji także nastąpi wykrycie cyklu, jednak po drodze występuje kilka stanów akceptujących. Ponieważ $B > C \wedge B > D$, B zostanie przesłane dalej.

(c) Przypadek podobny do poprzedniego, lecz krawędź powrotna skierowana jest w C (zamiast w B). Efekt tej zmiany to umiejscowienie największego akceptującego poprzednika poza cyklem

($B > C \wedge B > D$). Taki cykl zostanie pominięty przez zastosowaną heurystykę (1. podstawowy powód).

(d) Maksymalny akceptujący poprzednik znajduje się w cyklu, jednak to nie on go zaczyna. Próba propagacji wartości C do B zakończy się fiaskiem, ponieważ $C < B$. Dalej B przekazane zostanie do B. W wyniku tego porównuje się ze sobą obecną wartość D ze stanem, do którego wraca krawędź. $B \neq C$, więc do wykrycia cyklu nie dojdzie.

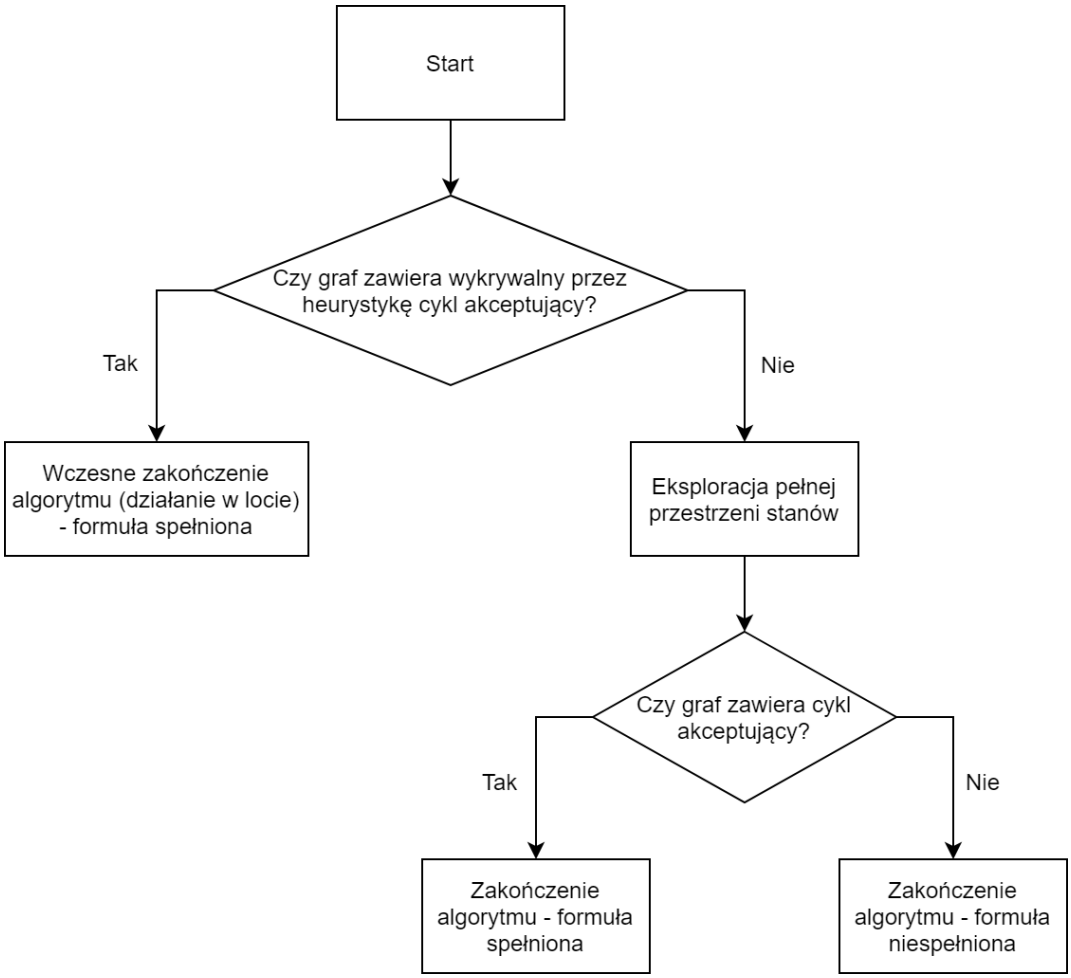


$A > B > C > D > E$

Rys. 4.1. Przykłady działania heurystyki algorytmu. W przypadkach (a) i (b) akceptujący cykl zostanie wykryty, jednak dla (c) i (d) już nie.

4.2. Działanie w locie

Zastosowany algorytm działa w locie (poziom 1). W tym przypadku oznacza to, że może on zakończyć się przed eksploracją całego grafu stanów. Stany są generowane i na bieżąco (każdy po kolei) weryfikowane. Powodem, że to poziom 1 (a nie 2), przez który nie dzieje się tak zawsze, jest zastosowana heurystyka. Pozwala ona wykryć część cykli akceptujących w locie, jednak są przypadki, kiedy nie wystarcza. Wtedy cykl zostanie pominięty. Do tego momentu algorytm ten okazuje się niewystarczający i może zwrócić niepoprawny wynik. Zostało to uniknięte poprzez dodanie drugiej części, która odnajdzie już każdy cykl akceptujący (OWCTY). Niestety nie działa ona w locie, więc potrzebuje wygenerowanej całej przestrzeni stanów.



Rys. 4.2. Diagram prezentujący, kiedy dochodzi do wczesnego zakończenia algorytmu.

Ustalenie, kiedy algorytm zadziała w locie, przedstawia rys. 4.2. Kluczowe pytanie to czy graf zawiera wykrywalny przez heurystykę cykl akceptujący. Jeśli tak, wtedy z cała procedura zakończy się wcześnie (bez eksploracji wszystkich wierzchołków, a zaraz po zamknięciu się cyklu).

W przypadku grafu posiadającego cykl akceptujący, który zostanie pominięty (zgodnie z 2 powodami opisanymi w sekcji 4.1), dojdzie do pełnego przeszukania, aby go wykryć (brak działania w locie). Niespełnienie formuły zawsze implikuje przegląd wszystkich wierzchołków grafu. Nie można orzekać niespełnienia przed sprawdzeniem całości.

5. Rozproszona implementacja

6. Podsumowanie

6.1. TODO

Bibliografia

[1] Joost-Pieter Katoen Christel Baier. *Principles of Model Checking*. MIT Press, 2008.

[2] Fred B. Schneider Bowen Alpern. "Recognizing safety and liveness". In: *Distributed Computing* 2.3 (1987), pp. 117–126.

[3] Fred B. Schneider Bowen Alpern. "Defining liveness". In: *Information Processing Letters* 21.4 (1985), pp. 181–185.

[4] William K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall, 2005.

[5] Edmund M. ClarkeWilliam KlieberMiloš NováčekPaolo Zuliani. "Model Checking and the State Explosion Problem". In: *LASER Summer School* (2011), pp. 1–30.

[6] Ebru Aydin Gol Calin Belta Boyan Yordanov. *Formal Methods for Discrete-Time Dynamical Systems*. Springer International Publishing, 2017.

[7] A. Prasad Sistla. "Theoretical issues in the design and verification of distributed systems". PhD thesis. Harvard University, 1983.

[8] A. P. Sistla P. Wolper M. Y. Vardi. "Reasoning about infinite computation paths". In: *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. 1983, pp. 185–194. DOI: *10.1109/SFCS.1983.51*.

[9] J Richard Büchi. „Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic". W: t. 44. Grud. 1966, s. 1–11.

[10] Pierre Wolper A. Prasad Sistla Moshe Y. Vardi. "The complementation problem for Büchi automata with applications to temporal logic". In: *Theoretical Computer Science* 49 (1987), pp. 217–237.

[11] Jun Pang Stephan Merz. *Formal Methods and Software Engineering*. Springer International Publishing, 2014.

[12] Rob Gerth i in. „Simple On-the-Fly Automatic Verification of Linear Temporal Logic". W: *Proceedings of the 6th Symposium on Logic in Computer Science* (grud. 1995). DOI: *10.1007/978-0-387-34892-6_1*.

[13] Patrice Godefroid i Gerard Holzmann. „On the Verification of Temporal Properties". W: *IFIP Transactions C: Communication Systems* (wrz. 1994).

[14] Gerard Holzmann, Doron Peled i Mihalis Yannakakis. „On Nested Depth First Search". W: *The spin verification system, DIMACS series in discretemathematics and theoretical computer science, vol 32. AMS* (maj 1999).

[15] Antti Valmari Jaco Geldenhuys. "More efficient on-the-fly LTL verification with Tarjan's algorithm". In: *Theoretical Computer Science* 345.1 (Nov. 2005), pp. 60–82.

[16] John Reif. „Depth-First Search is Inherently Sequential". W: *Information Processing Letters* 20 (czer. 1985), s. 229–234. DOI: *10.1016/0020-0190(85)90024-9*.

[17] Petr Ročkai Jiří Barnat Luboš Brim. "On-the-fly parallel model checking algorithm that is optimal for verification of weak LTL properties". In: *Science of Computer Programming* 77.12 (Oct. 2012), pp. 1272–1288.

[18] Ivana Černá i Radek Pelánek. „Distributed Explicit Fair Cycle Detection (Set Based Approach)". W: *Model Checking Software*. Wyed. Thomas Ball i Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, s. 49–73. ISBN: 978-3-540-44829-7.

[19] Luboš Brim i in. „Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking". W: *Formal Methods in Computer-Aided Design*. Wyed. Alan J. Hu i Andrew K. Martin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, s. 352–366. ISBN: 978-3-540-30494-4.