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 mgr inż. Michałowi Wypychowi za poświęcony czas i wszelkie wskazówki w czasie powstawania tej pracy.

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

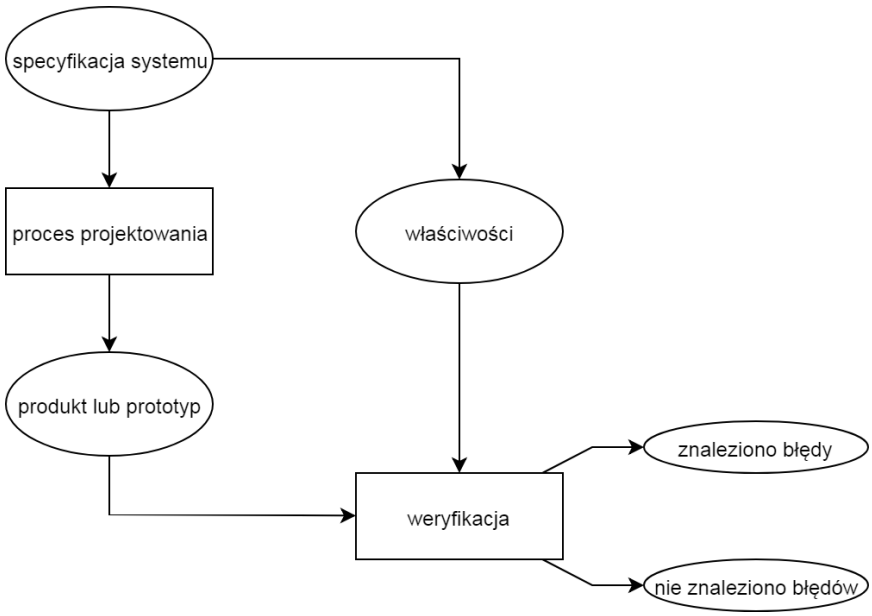
W następnym rozdziale pracy omówione zostanie zagadnienie weryfikacji modelowej w kontekście tematu pracy. Rozdział trzeci zawiera ogólny opis architektury stworzonego systemu. W czwartym rozdziale znajdują się szczegóły dotyczące zastosowanego algorytmu. Piąty rozdział zawiera prezentację rozproszonej 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 zawiera oczekiwane właściwości. Mogą one być podstawowe i niezależne od implementowanej dziedziny (np. nigdy nie dojdzie do zakleszczenia) lub z nią związane (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 błędami jest testowanie oprogramowania (testy jednostkowe, integracyjne, systemowe itp.). Proces testowania oprogramowania polega na uruchamianiu skompilowanej całej aplikacji lub jednie poszczególnych jej elementów wraz z wymuszeniem różnych przepływów sterowania, po czym porównuje się rezultaty rzeczywistego działania fragmentu kodu z oczekiwanymi wynikami. Niestety, przetestowanie wszystkich wariantów w sposób wyczerpujący okazuje się praktycznie niemożliwe, zwykle sprawdzane są je dynie warunki brzegowe, które stanowią mały podzbiór wszystkich potencjalnych kombinacji.

Szczególnym wyzwaniem okazują się testy programów współbieżnych. Błędy zawarte w takich aplikacjach mogą ujawniać się w specyficznych warunkach. Zdarza się, że testy działają niedeterministycznie i trudno wyjaśnić czemu dochodzi do ich niepowodzenia co kilka tysięcy uruchomień. Wynika to zwykle z braku kontroli nad współdzielonym zasobem lub niesynchronizowania pracy wątków działających równolegle.

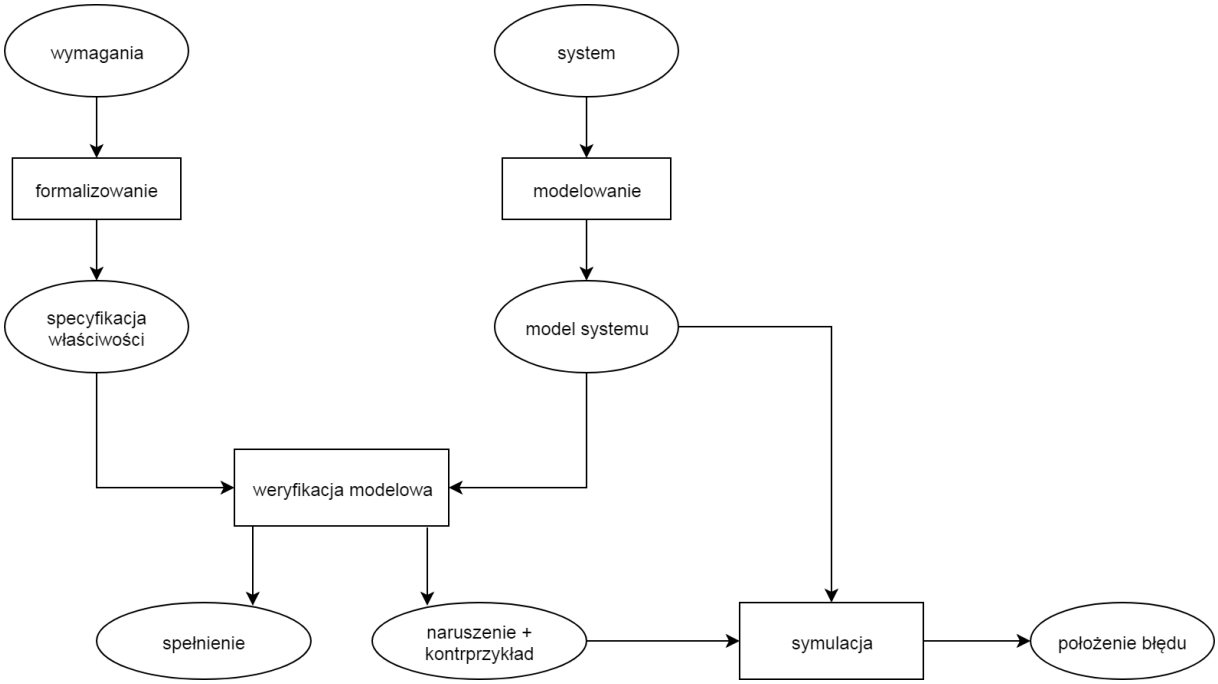
2.2. Weryfikacja modelowa

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

Modelem nazywamy opis mechaniki systemu (zwykle uproszczony), który pozwala utworzyć graf reprezentujący wszystkie stany systemu. Techniki weryfikacji bazujące na modelu opisują zachowanie systemu deterministycznie i kompletnie. Samo tworzenie pełnego modelu może wykryć luki lub niespójności w projekcie. Kolejnym krokiem jest eksploracja stanów systemu. Dzieje się ona w podejściu “brute-force“ – przejrzane zostają wszystkie możliwe scenariusze. W ten sposób udowadnia się spełnialność właściwości. Weryfikacji podlega również kolejność zdarzeń w czasie, np. czy system zawsze odpowie na żądania klienta w tym samym porządku, w jakim je otrzymał.

Kategorie właściwości współbieżnego systemu [1]:

- 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



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

Model systemu zazwyczaj jest generowany automatycznie na podstawie opisu w odpowiednim języku lub dialekcie wspieranym przez narzędzie. Weryfikator modelowy przeszukuje kolejne stany. Następnie sprawdzane są one pod kątem właściwości. Po znalezieniu naruszenia specyfikacji prezentowany zostaje kontrprzykład, czyli cała ścieżka wykonania, która do niego prowadzi. Zaprezentowanie kontrprzykładu pozwala lepiej zrozumieć problem, namierzyć źródło niespójności w modelu – schemat na rys. 2.2.

To nie jedyne zalety. Kolejnymi mocnymi stronami weryfikacji modelowej są:

– Wsparcie częściowej weryfikacji – możemy sprawdzać poszczególne właściwości niezależnie, nawet gdy pełna specyfikacja nie jest gotowa.

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

– Dostarcza informacji diagnostycznych po wykryciu niespełnionej właściwości.

– Wszystkie możliwe ścieżki wykonania zostają sprawdzone.

– Uruchomienie weryfikatora nie wymaga ekspertyzy na tym polu, a przynajmniej nie jest znacząco trudniejsze od już szeroko stosowanych testów jednostkowych.

– Łatwo zintegrować to rozwiązanie z cyklem wytwarzania oprogramowania.

– Obecnie wzrasta zainteresowanie tym podejściem.

Oczywiście nie brak także wad:

- Weryfikowany jest model systemu, a nie sam system. Brak tu gwarancji, że implementacja poprawnie go odtwarza. Model mógł także ulec zbytniemu uproszczeniu.
- Sprawdzane są tylko wyspecyfikowane wymagania. Nie można wnioskować o poprawności właściwości, których nie ma (błąd niedokładnej specyfikacji [4]).
- Aplikacje oparte na dużej ilości danych mogą posiadać zbyt wiele stanów, aby weryfikator mógł je wszystkie wygenerować. W ekstremalnych przypadkach liczba stanów może być nieskończona.
- Podatność na problem eksplozji przestrzeni stanów [5].

2.3. Logika LTL

Logika LTL to jedna 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 $500ms$.

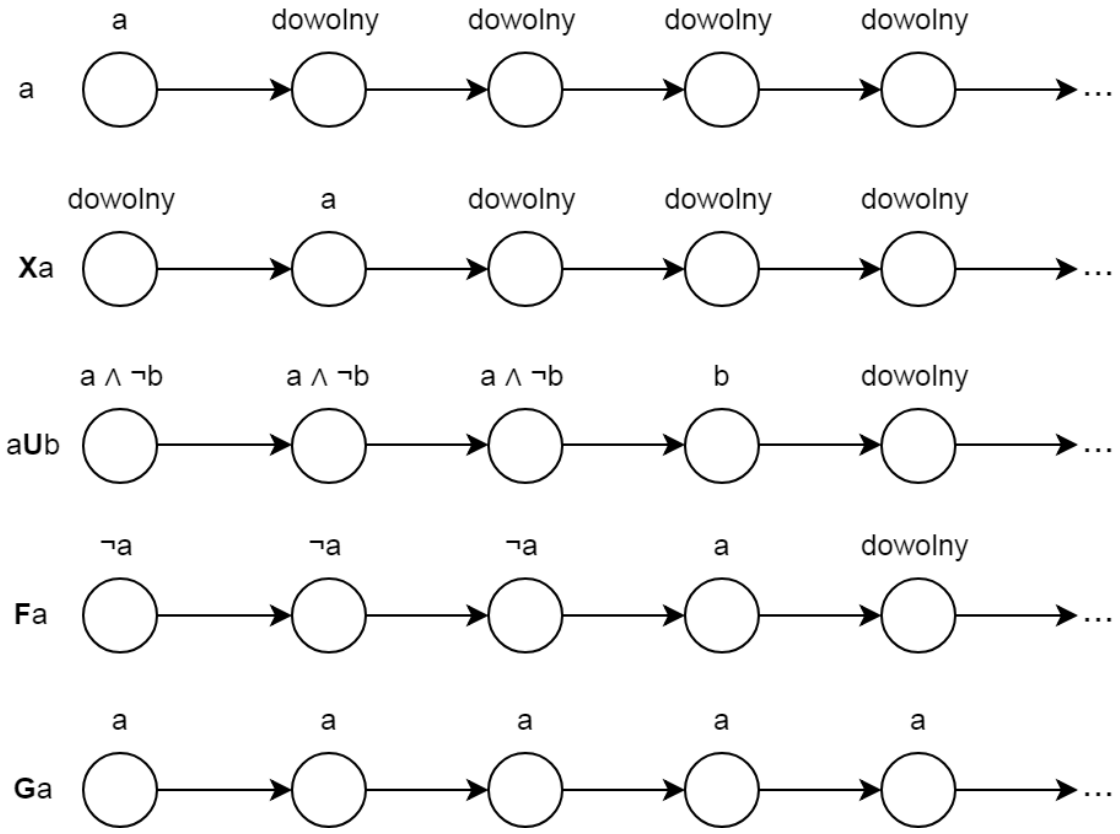
Formuły LTL ponad zbiorem AP wyrażeń atomowych przedstawia poniższy opis:

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

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 – $\mathbf{G}loball y$ – 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)



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

2.4. Automat Büchiego

W poprzedniej sekcji przedstawiona została logika czasu liniowego 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 błędu powinny zostać dostarczone w postaci sekwencji akcji lub stanów, które prowadzą do naruszenia wymagań. Dostarczona ścieżka ma ułatwić poprawienie modelu i wyeliminowanie błędu. 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 najczęściej z całego zbioru wymagań $\varphi_1, ..., \varphi_k$. 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, jak w przypadku gdy wiadomo, która z formuł φ_i została niespełniona.

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.

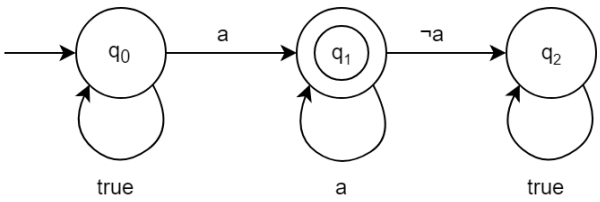
W niedeterministycznym automacie Büchiego (NBA – ang. *non-deterministic Büchi automaton*), funkcja tranzycji δ zastąpiona jest relacją tranzycji Δ, która zwraca zbiór stanów, a w miejscu stanu początkowego q_0 znajduje się zbiór stanów początkowych.

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].

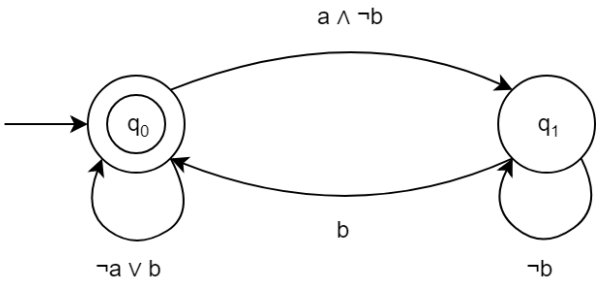
Na rys. 2.4 oraz 2.5 znajdują się przykładowe NBA, wraz z odpowiadającymi im formułami.



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

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 $\neg\varphi$.



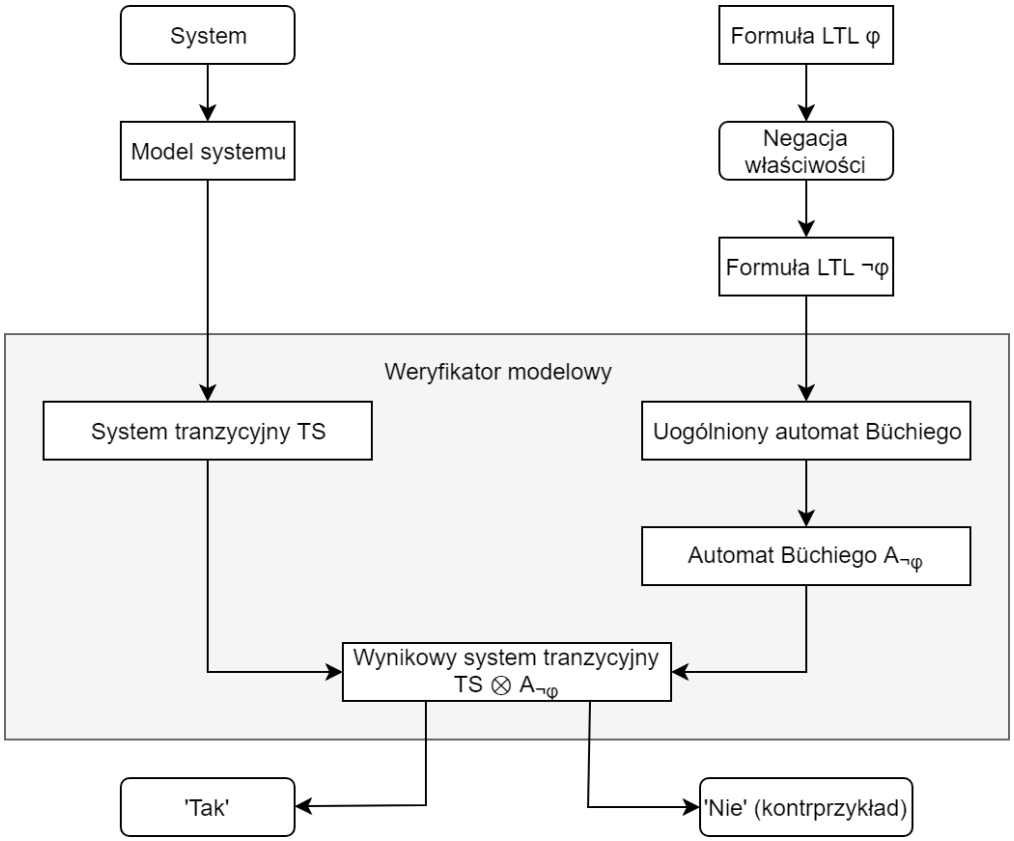
Rys. 2.5. NBA dla $\mathbf{G}(a \Rightarrow \mathbf{F}b)$ (źródło [1]).

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".

$TS \otimes A$ – iloczyn synchroniczny (ang. *synchronous product*) oznacza system tranzycyjny, którego składowe systemy tranzycyjne (TS i A) wykonują wszystkie akcje synchronicznie, bez ich pomijania przez którykolwiek składowy system tranzycyjny [1].

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 ich liczbę. Preferowane podejście to generowanie kolejnych stanów i ich weryfikacja ich w locie, czyli w trakcie ich generacji. Takie rozwiązanie pozwala na wykrycie niespójności we wczesnym etapie, unikając przeglądania 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 prawdopodobnie nie istnieje [16].

Celem algorytmu weryfikacji modelowej jest eksploracja stanów systemu wraz z synchronicznym wykonywaniem tranzycji w automacie Büchiego (odwzorowującym zadaną formułę LTL). Znalezienie stanu akceptującego dla automatu Büchiego nie oznacza jeszcze sukcesu. Należy znaleźć taki cykl w systemie tranzycyjnym, aby znajdował się w nim stan akceptujący (zgodnie z definicją automatu Büchiego) – wtedy formuła została spełniona. Zadanie algorytmów sprowadza się więc do wyszukiwania cykli zawierających przynajmniej jeden stan akceptujący wskazany przez automat Büchiego - cykli akceptujących.

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 post-order DFS, dzięki czemu możliwe jest jego zrównoleglenie. Procedura sortowania topologicznego oparta na BFS, niestety 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, czyli indeksu akceptującego stanu, który jest największy. Podobnie do OWCTY, jego wykonanie polega na wielu przejściach aż do uzyskania stabilnego stanu. Każde z nich w pełni przekazuje maksimum poprzedników do wszystkich stanów. Obliczenie poprawnej wartości maksymalnego poprzednika dla wierzchołka wymaga czasem wielokrotnego przesłania nowej (większej) wartości z tego samego stanu źródłowego do następnika.

Zjawisko to określa się jako repropagacja. Po znalezieniu wierzchołka, który jest swoim maksymalnym poprzednikiem (czyli 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 dzięki temu może zakończyć generację całego grafu stanów wcześniej przed jego wyczerpującą eksploracją. Jeśli jednak cykl akceptujący nie został znaleziony przez powyższą heurystykę, to nie oznacza, że takiego cyklu nie ma. Była to jedynie heurystyka pozwalająca wcześniej zakończyć generację całego grafu stanów. Z drugiej strony, jeśli zakończona została poprzednia faza, tzn. że cały graf został już wygenerowany, wykonane zostaną instrukcje zgodnie z algorytmem 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.

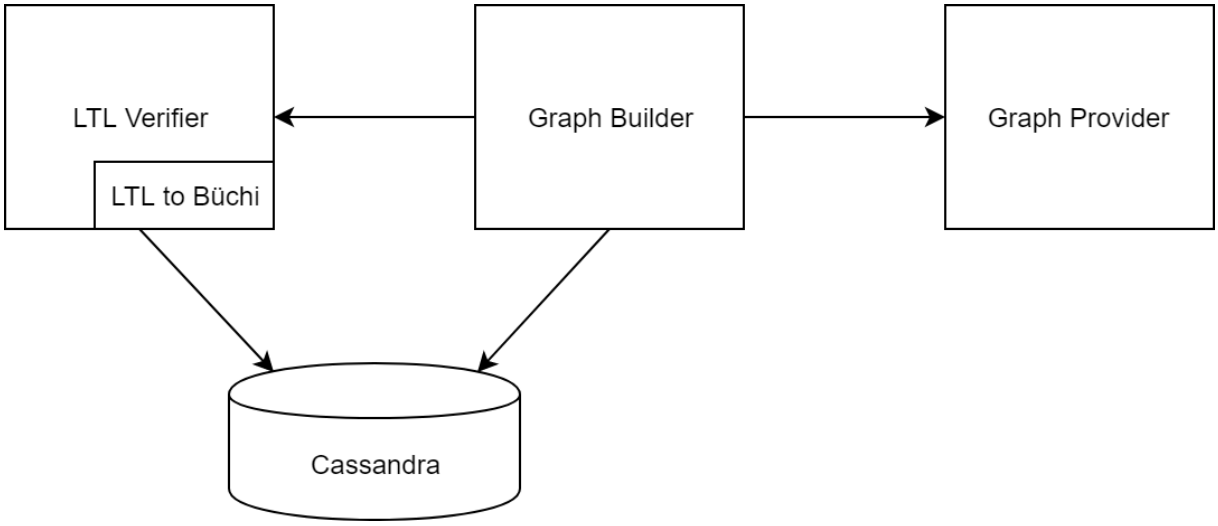
Warto zauważyć, że algorytmy poziomu zerowego bywają także rozważane jako niedziałające w locie. Poziom pierwszy oznacza, że w zależności od danych wejściowych algorytm zakończy swoje działanie szybciej, jednak zdarzają się też sytuacje, które wymagają przeszukania całej przestrzeni stanów, aby wykryć błąd. Przykładowy algorytm poziomu 0 to OWCTY, poziomu 1 on-the-fly OWCTY, poziomu 2 MAP. Dokładniejszy opis tych algorytmów znajduje się w rozdziale 4.

3. Architektura systemu

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

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 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 serwerowa aplikacja zajmująca się weryfikacją modelową i zawiera główną część tego algorytmu. Konwertuje ona także formuły LTL do automatów Büchiego. Napisana została w języku Java z wykorzystaniem frameworka Spring. Interfejs implementowany przez LV opisuje listing 3.2.

```
public interface LtlVerifier {
  VerificationId initializeVerification(VerificationInit verificationInit);

  VerificationResult newStates(NewStates newStates,
                               VerificationId id);

  VerificationResult finish(VerificationId id);
}
```

Listing 3.2. Interfejs implementowany przez LV.

Graph Builder (**GB**) odgrywa 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 do przesyłania danych siecią. Zapytania GB do LV realizowane są poprzez protokół HTTP w formacie JSON.

3.2. Alvis

Alvis to język formalny, którego celem jest dostarczenie elastyczności w modelowaniu systemów współbieżnych i czasu rzeczywistego wraz z możliwością weryfikacji opartej o metody formalne. Jego nazwa pochodzi od połączenia słów **al**gebra i **vis**ual. To łatwy w użyciu dla inżynierów formalny język modelowania. Oferuje także język wizualny do projektowania połączeń między współbieżnymi elementami systemu. Za pomocą Alvisa modelować można systemy współbieżne, czasu rzeczywistego, wbudowane oraz rozproszone.

Model Alvisa to system agentów, które zazwyczaj działają współbieżnie, komunikują się między sobą, konkurują o współdzielone zasoby itp. Agenty dzielą się na aktywne i pasywne. Zachowanie każdego z nich definiuje się w warstwie kodu, którego konstrukcje przypominają języki wysokiego poziomu.

W tworzonym rozwiązaniu potrzebny był model systemu, dla którego przeprowadzona zostanie weryfikacja. Wykorzystany został taki, który modeluje algorytm eksploracji stanów dla języka Alvis (czyli aplikacje GB, GP oraz bazę danych). To konstrukcja, która pozwala sprawdzić samą siebie, a więc czy algorytm budujący pełny graf stanów nie zawiera błędów. W jej skład wchodzi wiele elementów, np. stan heurystyki przeszukiwania BFS, bazy danych, wątku budującego graf, czy repozytoriów na dane. To one będą weryfikowane pod kątem spełnienia formuł LTL.

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, oznacza to, że nie istnieje cykl akceptujący. 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 częścią 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 tu repropagacji, aby złożoność obliczeniowa pozostała i liniowa

i była proporcjonalna do rozmiaru grafu. Skutek tego ograniczenia to brak wykrywalności wszystkich cykli (stąd klasyfikacja tej fazy jako heurystyka).
Wyróżnić można dwa podstawowe przypadki, kiedy metoda ta będzie nieskuteczna w wykryciu istniejącego cyklu akceptującego:

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.

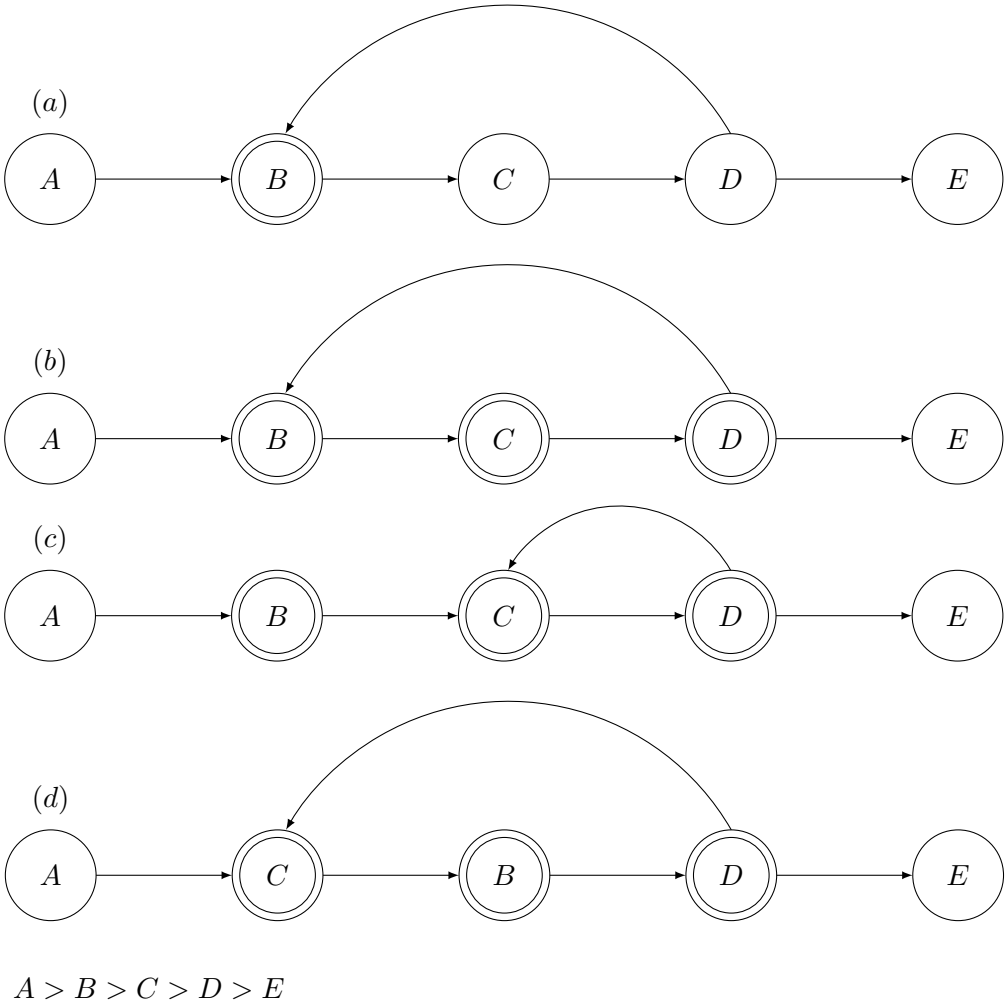
W sytuacji, gdy żadne z powyższych nie zachodzi, akceptujący cykl zostanie wykryty. Przykłady z rys. 4.1:

(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. Mimo że wewnątrz cyklu są stany akceptujące, to nie zostają one spropagowane, bo wartość B jest większa ($B > C \wedge B > D$). Taki cykl zostanie pominięty przez zastosowaną heurystykę (1. podstawowy powód).

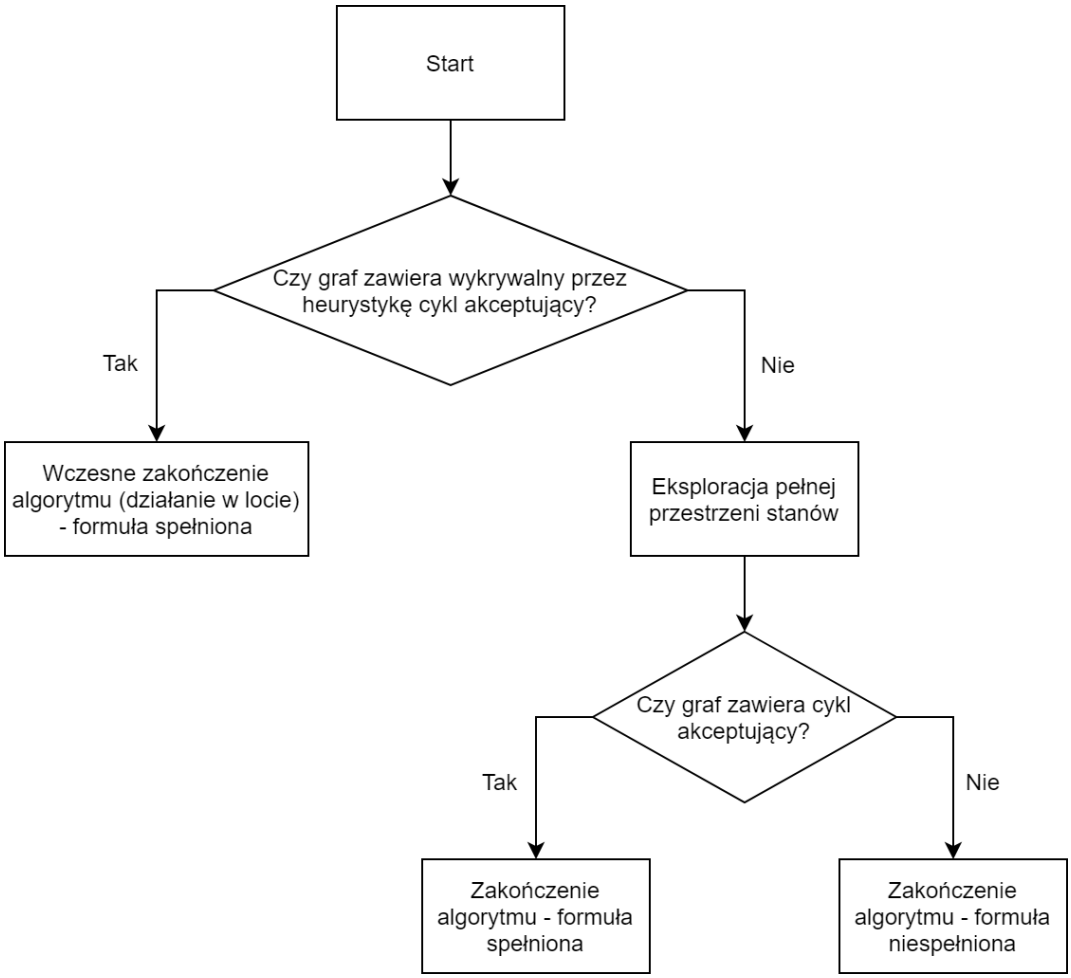
(d) Maksymalny akceptujący poprzednik znajduje się w cyklu, jednak to nie on go zaczyna. Próba propagacji wartości C do B zakończy się fiaskiem, ponieważ $C < B$. Dalej B przekazane zostanie do D. W wyniku tego porównuje się ze sobą obecną wartość D ze stanem, do którego wraca krawędź. $B \neq C$, więc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



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 cała procedura zakończy się wcześnie (bez eksploracji wszystkich wierzchołków, a zaraz po domknięciu się cyklu).

W przypadku grafu posiadającego cykl akceptujący, który zostanie pominięty (zgodnie z dwoma 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

Opisany algorytm dostosowany został do środowiska rozproszonego. Taka implementacja pozwala uzyskać wysoką wydajność całego systemu, który staje się skalowalny. Osiąga się to poprzez horyzontalne replikowanie aplikacji, która obsługuje weryfikację modelową. Uruchomione zostaje wiele instancji tego samego serwisu.

Dzięki jego bezstanowości (żadnych danych sesyjnych, które przechowywane są w pamięci pomiędzy zapytaniami do serwera) nie ma znaczenia, która replika obsłuży żądanie. Każda wykona je identycznie, a cały stan odczytywany i umieszczany jest z wykorzystaniem bazy danych. W środowisku chmurowym system może być dynamicznie dostosowywany do aktualnych potrzeb, aby zapewnić szybkość, ale także nie marnować zasobów.

Poniżej przedstawiony zostanie opis zrealizowanego systemu.

5.1. Graph Builder

GB inicjuje cały algorytm oraz działa jako dostawca stanów dla weryfikatora (uzyskuje je od GP). Najpierw wybierana jest formuła LTL, która zostanie sprawdzona (szczegóły jej specyfikacji w sekcji 5.2). Następnie zaczyna się już główna część algorytmu. Pobierane są stany początkowe, po czym przekazywane są LV oraz heurystyce przeszukującej graf.

Każda weryfikacja posiada swój unikalny identyfikator, aby móc prowadzić wiele z nich równolegle na tym samym grafie stanów. Nadawanie go następuje podczas wysyłania pierwszych stanów do LV. Używa się go później w każdej komunikacji z weryfikatorem.

Wykorzystywane jest przeszukiwanie wszerz (BFS) zgodnie z fazą *initialise* (algorytm 2). Kolejne stany uzyskiwane od GP zostają zapisywane w bazie danych, a także sprawdzane przez LV, który zwraca informacje o statusie weryfikacji. Może ona zakończyć się szybciej (działanie w locie) w przypadku formuły, która nie jest spełniana przez graf. W przeciwnym wypadku pobierane są kolejne stany aż do pełnej eksploracji.

Po odwiedzeniu wszystkich węzłów w grafie następuje wykonanie weryfikacji odpowiadającej liniom 4-8 w algorytmie 1 przez LV. Po tym etapie jednoznacznie wiadomo, czy formuła została spełniona. Linie odwołujące się do *verificationService* sprowadzają się wykonania zapytań do LV.

Kod prezentujący funkcjonalność realizowaną przez GB znajduje się w listingu 5.1.

```
public VerificationState checkModel() {
  Collection<SystemState> states = initialStates.initialStates();
  List<StoredSystemState> saved = store.save(states).getNewlyStoredStates();
  heuristic.newStates(saved);
  VerificationId verificationId = verificationService.init(saved);
  while (heuristic.hasMoreToExplore()) {
    if (heuristic.shouldVisitState()) {
      StoredSystemState toVisit = heuristic.nextStateToVisit();
      Collection<Transition> transitions = description.allTransitionFromState(
          toVisit.getSystemState());
      heuristic.newTransitions(toVisit, transitions);
    }
    if (heuristic.shouldExploreTransition()) {
      Pair<StoredSystemState, Transition> toExplore =
          heuristic.nextTransitionToExplore();
      Collection<SystemState> reachableStates = description.allReachableStates(
          toExplore.getFirst().getSystemState(), toExplore.getSecond());
      StoredStatesResult result = store.save(reachableStates);
      List<StoredSystemState> storedOnlyNewStates = result.getNewlyStoredStates();
      if (!storedOnlyNewStates.isEmpty()) {
        store.save(toExplore.getFirst(),
            storedOnlyNewStates, toExplore.getSecond());
        heuristic.newStates(storedOnlyNewStates);
      }
      List<StoredSystemState> storedOnlyOldStates = result.getAlreadyStoredStates();
      if (!storedOnlyOldStates.isEmpty()) {
        store.save(toExplore.getFirst(), storedOnlyOldStates,
            toExplore.getSecond());
      }
      List<StoredSystemState> allReachableStates =
          new ArrayList<>(storedOnlyNewStates);
      allReachableStates.addAll(storedOnlyOldStates);
      if (!allReachableStates.isEmpty()) {
        VerificationResult verificationResult = verificationService.verifyNewStates(
            verificationId, toExplore, allReachableStates);
        if (verificationResult.getVerificationState() == FORMULA_UNSATISFIED) {
          return FORMULA_UNSATISFIED;
        }
      }
    }
  }
  VerificationResult finishResult = verificationService.finish(verificationId);
  return finishResult.getVerificationState();
}
```

Listing 5.1. Kod algorytmu realizowanego przez GB w języku Java.

5.2. Specyfikacja formuły LTL

Obiektowa reprezentacja formuły LTL nie wystarcza, aby móc je tworzyć. Potrzebny jest sposób umożliwiający zapis tekstem. Dzięki temu można przygotowywać zestawy formuł, jakie powinien spełniać system, po czym wykorzystać je jako parametr dla weryfikatora. Konieczne było więc ich parsowanie.

Spełnić należało dwa wymagania. Pierwszym była obsługa spójników zdaniowych i operatorów temporalnych, jakie oferuje LTL. Drugie to pozostawienie elastyczności, co do samych warunków, tak aby móc je dowolnie komponować i łączyć. Wspierać należało porównywanie dowolnych wartości modelu na różne sposoby, np. przez tę samą wartość, wyrażenia regularne, czy relację porządku.

Zamiast tworzenia własnego języka, wykorzystany został Kotlin poprzez jego silnik skryptowy. Pozwala na nieograniczoną elastyczność w tworzeniu formuł. Kod wykonywany jest locie i zamieniany na reprezentację obiektową. Jego składnia pozwala na elegancką notację z użyciem operatorów infiksowych.

Obiekt formuły opisany został w sekcji 5.3. Do porównywania wartości wykorzystana została biblioteka *Hamcrest* (*http://hamcrest.org*). Oferuje ona bogaty zestaw komparatorów, możliwość ich łączenia i testowania na dowolnym obiekcie systemu.

Przykładową formułę prezentuje listing 5.2.

```
val bfsPcEq1 = proposition(hasProperty<SystemState>("bfsHeuristicState",
        hasProperty<Int>("pc", equalTo(1))))
val bfsPcGt1 = proposition(hasProperty<SystemState>("bfsHeuristicState",
        hasProperty<Int>("pc", greaterThan(1))))
val formula = globally(bfsPcEq1 implies finally(bfsPcGt1))
```

Listing 5.2. Przykładowa formuła LTL w języku Kotlin.

5.3. LTL to Büchi

Do obiektowej reprezentacji formuły wykorzystana została biblioteka *ltl2buchi* (*https://github.com/fraimondi/ltl2buchi*). Pozwala ona na ich tworzenie z dowolnymi twierdzeniami (używa klas generycznych).

Jej główną funkcjonalnością jest ich konwersja do grafów reprezentujących uogólnione automaty Büchiego. Te są redukowane, degeneralizowane i znów redukowane. Brakuje jednak zautomatyzowanej możliwości poruszania się po tym automacie.

Biblioteka wymagała zaadaptowania do potrzeb projektu. Wyodrębniona została funkcja konwertująca formułę do grafu (listing 5.3).

```
<T> Graph<T> ltlToBuchiAutomaton(Formula<T> formula) {
  val optimizedFormula = new Rewriter<>(formula).rewrite();
  Graph<T> gba = Translator.translate(optimizedFormula);
  Graph<T> reducedGba = SuperSetReduction.reduce(gba);
  Graph<T> ba = Degeneralize.degeneralize(reducedGba);
  Graph<T> sccReducedBa = SCCReduction.reduce(ba);
  Graph<T> simplifiedBa = Simplify.simplify(sccReducedBa);
  Graph<T> sfsReducedBa = SFSReduction.reduce(simplifiedBa);
  Formula.resetStatic();
  return sfsReducedBa;
}
```

Listing 5.3. Funkcja konwertująca formułę LTL do grafu.

Drugim krokiem było stworzenie funkcji do budowania formuł, które wykorzystają potencjał Kotlina. Każdy operator dwuargumentowy otrzymał wersję infiksową. Kilka funkcji prezentuje listing 5.4.

```
infix fun <T> Formula<Matcher<T>>.and(right: Formula<Matcher<T>>)
    : Formula<Matcher<T>> = LtlFactory.And(this, right)

infix fun <T> Formula<Matcher<T>>.implies(right: Formula<Matcher<T>>)
    : Formula<Matcher<T>> = LtlFactory.Implies(this, right)

infix fun <T> Formula<Matcher<T>>.or(right: Formula<Matcher<T>>)
    : Formula<Matcher<T>> = LtlFactory.Or(this, right)

fun <T> next(formula: Formula<Matcher<T>>): Formula<Matcher<T>>
    = LtlFactory.Next(formula)

fun <T> globally(formula: Formula<Matcher<T>>): Formula<Matcher<T>>
    = LtlFactory.Globally(formula)
```

Listing 5.4. Przykładowe funkcje do budowania formuł w języku Kotlin.

Najważniejsza część integracji to implementacja klasy automatu Büchiego, która umożliwi przechodzenie pomiędzy węzłami na podstawie kolejnego stanu systemu. Stworzony został algorytm, który wybiera krawędź, którą należy podążyć. Szukana jest taka, aby spełnić wszystkie stojące na jej straży wyrażenia. W tym celu eksploruje się wewnętrzną strukturę grafu, sprawdza listę wyrażeń oraz ich ewentualną negację. Następnie zwrócony zostaje nowy automat, którego aktualny węzeł, to ten znaleziony.

Klasa pozwala także na sprawdzenie, czy aktualnie znajduje się w stanie akceptującym. Mieści się tutaj również funkcja, która przyjmuje formułę, neguje ją oraz buduje automat Büchiego (zgodnie z listingiem 5.3). Kod prezentujący powyższą funkcjonalność znajduje się w listingu 5.5.

```
public class BuchiAutomaton<T> {
  private final Graph<Matcher<T>> buchiAutomaton;
  private final Node<Matcher<T>> currentNode;

  /* [...] */

  public BuchiAutomaton<T> nextState(T targetSystemState) {
    val outgoingEdges = this.currentNode.getOutgoingEdges();
    if (outgoingEdges.isEmpty()) {
      throw new IllegalStateException("No more edges to follow");
    }

    Node<Matcher<T>> nextNode = outgoingEdges.stream()
        .sorted(noGuardEdgeToTheEndComparator)
        .filter(edge -> allGuardsMatch(targetSystemState, edge))
        .findFirst()
        .map(Edge::getNext)
        .orElse(this.currentNode);

    return new BuchiAutomaton<>(this.buchiAutomaton, nextNode);
  }

  private boolean allGuardsMatch(Object system, Edge<Matcher<T>> edge) {
    return edge.getGuard().stream().allMatch(guard -> {
      val matches = guard.getAtom().matches(system);
      return guard.isNegated() != matches;
    });
  }

  public boolean isAccepting() {
    return this.currentNode.getAttributes().getString(ACCEPTING_KEY) != null;
  }

  public static <T> BuchiAutomaton<T> prepareAutomaton(
          Formula<Matcher<T>> formulaToCheck) {
    val negatedFormula = formulaToCheck.negate();
    return newBuchiAutomaton(negatedFormula);
  }
}
```

Listing 5.5. Implementacja automatu Büchiego.

5.4. Weryfikator LTL

Najważniejsza część algorytmu znajduje się module LV. To bezstanowa aplikacja serwerowa, która wystawia interfejs przez protokół HTTP. Możliwe są trzy typy zapytań:

– Inicjujące – przyjmuje stany początkowe oraz formułę do sprawdzenia, po czym inicjuje algorytm. Ponadto zwraca *VerificationId*, czyli identyfikator weryfikacji, który będzie potrzebny w kolejnych zapytaniach.

– Odbierające kolejne stany systemu – to element weryfikacji w locie, która może się zakończyć na tym etapie. W takim przypadku zwracana jest stosowana informacja do klienta. To jedyne zapytanie, które wykonuje się wielokrotnie dla jednej formuły (ilość zależna od wielkości grafu ze stanami systemu).

– Finalizujące weryfikację – wykorzystywane, kiedy cały graf stanów został zeksplorowany, a formuła jest spełniona lub zastosowana heurystyka nie znalazła kontrprzykładu. Wykonana zostaje pozostała część algorytmu.

Wszystkie elementy algorytmu, które należy współdzielić pomiędzy zapytaniami, przechowywane są w bazie danych. Stworzone zostały serwisy, które obsługują tę funkcjonalność dla zbioru stanów poprzedzających cykl, wartości algorytmu MAP, czy samych automatów Büchiego.

5.4.1. Inicjajcja weryfikacji modelowej

Tutaj generowany jest identyfikator weryfikacji, a stany początkowe aplikowane są do automatu Büchiego. Każdy z nich (po wykonaniu tranzycji) zostaje zapisany w bazie danych. Ponadto stany systemu trafią do zbioru stanów poprzedzających cykl. W listingu 5.6 znajduje się kod implementujący tę funkcjonalność.

```
public VerificationId initializeModelChecking(BuchiAutomaton<SystemState> automaton,
                                              List<String> initialStateHashes) {
  val verificationId = new VerificationId();
  initialStateHashes.stream()
      .map(systemStatesService::findStateByHash)
      .map(targetState -> {
        val nextState = automaton.nextState(targetState.getSystemState());
        return new StoredBuchiAutomaton(verificationId, nextState,
            targetState.getHash());
      })
      .peek(this::addToApproxSet)
      .forEach(buchiAutomatonRepository::save);
  return verificationId;
}
```

Listing 5.6. Kod obsługujący inicjację weryfikacji modelowej.

5.4.2. Weryfikacja eksplorowanych stanów

Wielokrotnie wykonywanym zapytaniem jest to, które weryfikuje nowe stany systemu. Pozwala zakończyć sprawdzanie formuły LTL, jeśli znajdzie się akceptujący cykl dla jej zaprzeczenia.

Parametry wejściowe to lista nowych stanów, stan źródłowy, z którego one wychodzą oraz identyfikator weryfikacji. Działanie zaczyna się od pobrania z bazy danych automatu Büchiego, który odpowiada źródłu. Następnie przetwarzana jest cała lista nowych stanów w sposób równoległy, co pozwoli przyspieszyć wykonanie algorytmu. Można było zastosować to podejście, ponieważ nie ma tu żadnej operacji, która modyfikuje wspólny zasób. Wszystkie działania są rozłączne.

```
public boolean checkNewStates(NewStates newStates, VerificationId verificationId) {
  String sourceStateHash = newStates.getSourceStateHash();
  val storedBuchiAutomaton = findSavedBuchiAutomaton(verificationId,
                                                     sourceStateHash);
  return newStates.getReachableStatesHashes().stream().parallel()
      .map(systemStatesService::findStateByHash)
      .map(targetState -> toNextStateAutomaton(verificationId, storedBuchiAutomaton,
                                               targetState))
      .peek(this::addToApproxSet)
      .map(buchiAutomatonRepository::save)
      .anyMatch(buchiAutomaton -> runMapAlgorithm(sourceStateHash, buchiAutomaton));
}

private boolean runMapAlgorithm(String sourceStateHash,
                                StoredBuchiAutomaton buchiAutomaton) {
  String targetStateHash = buchiAutomaton.getMatchingStateHash();
  VerificationId verificationId = buchiAutomaton.getVerificationId();
  String sourceStateMap =
      mapAlgorithmService.getMap(verificationId, sourceStateHash);
  if (buchiAutomaton.getBuchiAutomaton().isAccepting()) {
    if (targetStateHash.equals(sourceStateHash) ||
            targetStateHash.equals(sourceStateMap)) {
        log.info(buildAcceptingPath(buchiAutomaton));
        return true;
    }
    mapAlgorithmService.setMap(verificationId, targetStateHash,
        mapAlgorithmService.getMaxMap(targetStateHash, sourceStateMap));
  } else {
    mapAlgorithmService.setMap(verificationId, targetStateHash, sourceStateMap);
  }
  return false;
}
```

Listing 5.7. Implementacja heurystyki pozwalającej na weryfikację modelową w locie.

Dla każdego stanu uzyskuje się odpowiadający mu automat Büchiego (aplikując stan systemu do automatu źródłowego). Kolejny krok to dodanie do zbioru stanów poprzedzających cykl, po czym następuje kluczowy moment, czyli zastosowanie heurystyki opartej na algorytmie MAP. W przypadku wierzchołka akceptującego sprawdza się, czy nie został zamknięty cykl akceptujący. Gdy do tego dojdzie, następuje zakończenie algorytmu, poinformowanie o tym klienta oraz wypisanie całej ścieżki od liścia do korzenia, która spowodowała niespełnienie zadanej formuły LTL. Całość działa zgodnie z opisem w sekcji 4.1.

W pozostałych przypadkach (kiedy nie wykryto akceptującego cyklu) obliczane są kolejne wartości dla algorytmu MAP. Implementacja została zawarta w listingu 5.7.

5.4.3. Finalizacja weryfikacji modelowej

W przypadku, gdy nie dochodzi do wczesnego zakończenia algorytmu, a cała przestrzeń stanów została już wygenerowana, należy dokończyć sprawdzanie, zgodnie z OWCTY. Jedyny argument wymagany od klienta to identyfikator weryfikacji. Powala on na uzyskiwanie stanów w zbiorze poprzedników cyklu akceptującego. Kolejne operacje na zbiorze wykonywane są w pętli do czasu, gdy przestaną zachodzić w nim zmiany lub okaże się pusty. Brak elementów oznacza, że nie ma cyklu akceptującego. Przedstawia to listing 5.8.

Ze zbioru najpierw usuwane są stany nieakceptujące, po czym dodaje się do niego następniki pozostałych oraz ustala wartości liczby poprzedników (listing 5.9).

Druga wykorzystywana funkcja to *eliminateNoPredecessors* – jej implementacja znajduje się w listing 5.10. Jej zadanie polega na usunięciu ze zbioru wszystkich stanów, które nie mają poprzedników. Co więcej, dla następników takich stanów, zmniejszana jest ich liczba poprzedników. Jeśli spadnie ona do zera, całość powtarza się rekurencyjnie.

```
public boolean finishAlgorithm(VerificationId verificationId) {
  int oldSize = Integer.MAX_VALUE;
  while (true) {
    int approxSetSize = approximationSetService.size(verificationId);
    if (approxSetSize == oldSize || approxSetSize <= 0) {
      break;
    }
    oldSize = approxSetSize;
    eliminateNoAccepting(verificationId);
    eliminateNoPredecessors(verificationId);
  }
  return approximationSetService.size(verificationId) > 0;
}
```

Listing 5.8. Kod obsługujący algorytm w przypadku, gdy nie zakończy się wcześniej.

```
  private void eliminateNoAccepting(String verificationId) {
    Queue<String> hashesQueue = new ArrayDeque<>();
    approximationSetService.streamAllStateHashes(verificationId)
        .forEach(stateHash -> {
          if (isAcceptingState(verificationId, stateHash)) {
            hashesQueue.offer(stateHash);
            approximationSetService.zeroPredecessorCount(verificationId, stateHash);
          } else { approximationSetService.remove(verificationId, stateHash); }
        });
    while (!hashesQueue.isEmpty()) {
      String sourceStateHash = hashesQueue.poll();
      findSuccessorStateHashes(sourceStateHash).forEach(stateHash -> {
            if (approximationSetService.contains(verificationId, stateHash)) {
              approximationSetService.incrementPredecessorCount(verificationId,
                                                                 stateHash);
            } else {
              hashesQueue.offer(stateHash);
              approximationSetService.add(verificationId, stateHash);
              approximationSetService.zeroPredecessorCount(verificationId, stateHash);
            }
          });
    }
  }
```

Listing 5.9. Implementacja funkcji usuwającej stany nieakceptujące ze zbioru.

```
private void eliminateNoPredecessors(String verificationId) {
  Queue<String> hashesQueue = new ArrayDeque<>();
  approximationSetService.streamAllStateHashes(verificationId)
      .filter(stateHash -> approximationSetService
          .getPredecessorCount(verificationId, stateHash) == 0)
      .forEach(hashesQueue::offer);
  while (!hashesQueue.isEmpty()) {
    String sourceStateHash = hashesQueue.poll();
    approximationSetService.remove(verificationId, sourceStateHash);
    findSuccessorStateHashes(sourceStateHash).forEach(stateHash -> {
          approximationSetService.decrementPredecessorCount(verificationId
                                                             stateHash);
          if (approximationSetService
              .getPredecessorCount(verificationId, stateHash) == 0) {
            hashesQueue.offer(stateHash);
          }
        });
  }
}
```

Listing 5.10. Implementacja funkcji usuwającej stany niemające poprzedników ze zbioru.

5.5. Baza danych

Funkcjonalność magazynu danych w systemie realizuje *Apache Cassandra* (*https://cassandra.apache.org*). To rozproszona baza NoSQL zaprojektowana do obsługi dużych zestawów danych. Dostarcza wysoką dostępność bez ryzyka, że pojedyncza awaria uszkodzi system. Oferuje wiele strategii replikacji, skalowalność i konfigurację poziomu spójności.

Cassandra została wykorzystana w projekcie m.in. do przechowywania eksplorowanych stanów, tranzycji, czy automatów Büchiego.
Eksplorowane stany zapisywane są w tabeli o prostej strukturze. Poza samą wartością znajduje się tutaj tylko identyfikator oraz wyliczony hasz, który służy do wyszukiwania:

- $id : uuid$
- $hash : text$
- $systemstate : system\_state$

Tranzycje podobnie jak odwiedzane stany wykorzystywane są zarówno przez GB, jak i LV. Każda z nich ma swą wartość oraz identyfikatory stanów, które łączy:

- $id : uuid$
- $fromsystemstateid : uuid$
- $tosystemstateid : uuid$
- $transition : transition$

Pozostałe tabele służą tylko i wyłącznie LV. Przechowują dane, które muszą zostać zapamiętane pomiędzy kolejnymi zapytaniami klienta. Najważniejsza z nich to zawierająca automaty Büchiego. Serializowanie samego tego obiektu okazało się nietrywialne ze względu na skomplikowaną i cykliczną strukturę w *ltl2buchi*. Drugim problemem były obiekty porównujące z biblioteki *Hamcrest*. W związku z tym zapisywana w bazie jest formuła, z której można utworzyć automat oraz identyfikator aktualnego węzła, aby go ustawić. Kolejna kolumna to *verificationidwithstatehash*. Będzie ona się powtarzać także w innych tabelach. Pozwala powiązać rekord z konkretną weryfikacją oraz samym haszem stanu systemu, którego to dotyczy. Wartość *previousautomatonstateid* przechowuje identyfikator poprzedniego automatu, dzięki któremu można zawsze odnaleźć ścieżkę aż do korzenia (po znalezieniu akceptującego cyklu). Całość tabeli prezentuje się następująco:

- $id : uuid$
- $previousautomatonstateid : uuid$
- $currentnodeid : int$
- $stringformula : text$
- $verificationidwithstatehash : text$

W projekcie wykorzystane zostały jeszcze 3 tabele. Potrzebne są, aby przechowywać dane, jakie do tej pory uzyskano przez algorytm. Każda z nich składa się z identyfikatora, *verificationidwithstatehash* (opisane powyżej) oraz dodatkowej kolumny specyficznej dla siebie. Dla tabeli zawierającej zbiór stanów poprzedzających cykl akceptujący to hasz stanu systemu. Kolejna zajmuje się ilością poprzedników dla każdego stanu, więc w jej skład wchodzi licznik. Ostatnia obsługuje algorytm MAP, dlatego ją cechuje wartość MAP dla stanu.

6. Podsumowanie

Celem pracy było stworzenie rozproszonego algorytmu weryfikacji modelowej na podstawie własności logicznych czasu liniowego LTL dla języka Alvis z wykorzystaniem frameworka Spring. Wykonany został system, na który składa się kilka aplikacji komunikujących się za pomocą sieci.

Rozwiązanie to pozwala na weryfikację modelową w skalowalny sposób. Wszystko dzięki bezstanowym serwisom oraz bazie NoSQL. Wdrożenie na platformie chmurowej umożliwiłoby sprawdzanie wielu formuł LTL jednocześnie. Wydajność byłaby zależna od liczby uruchomionych instancji. Ponadto algorytm działa równolegle dla grup stanów wychodzących z jednego węzła, aby wykorzystać potencjał wielu rdzeni procesora i przyspieszyć długość weryfikacji pojedynczej formuły.

Sposób implementacji pozwala też na krokowe weryfikowanie. To aplikacja eksplorująca stany nadaje tempo i kontroluje całość. Umożliwia to nawet pauzowanie procesu, a dzięki trwałemu zapisowi w bazie danych – kontynuację w dowolnym momencie.

Kolejną cechą jest działanie w locie. Pozwala to na wczesne zakończenie algorytmu, czyli zanim przeszukana zostanie cała przestrzeń stanów. Istnieją pewne warunki, które należy spełnić, aby do tego doszło. Jednak sam ten fakt daje możliwość kontroli zmian w modelu z dużym prawdopodobieństwem szybkiego wykrycia problemów. Równoległe uruchomienie wielu formuł pozwoli odnaleźć w locie znaczną część nowych błędów.

Aby mieć pewność, że system spełnia wszystkie założenia, trzeba oczywiście dokonać sprawdzenia wszystkich jego stanów. Nie można orzec spełnienia formuły, jeśli proces zakończy się wcześniej. Zwykle wymaga to długotrwałych obliczeń, ale dzięki implementacji, która umożliwia wdrożenie w środowisku chmurowym (z cechami opisanymi powyżej), czas ten ulega skróceniu.

6.1. Propozycje rozwoju

Podobnie jak w większości systemów tutaj także jest miejsce na usprawnienia. Odnaleźć można kilka obszarów, które można rozwinąć.

Jeden z nich to sposób komunikacji pomiędzy aplikacjami GB i LV. Interfejs aplikacji weryfikującej opiera się na protokole HTTP z wykorzystaniem formatu JSON. Do wydajniejszego zakodowania danych użyć można np. Protocol Buffers, FlatBuffers, czy Apache Thrift. Co więcej – zmiana modelu integracji na przesyłanie komunikatów pozwoliłaby na łatwiejsze zrównoleglenie pracy klienta i serwera.

Kolejny punkt na usprawnienie to baza danych. Obecnie dwie tabele (przechowujące stany i tranzycje) używane są przez dwie aplikacje. Jedynie GB dokonuje w nich modyfikacji, jednak potrzebne dane mogłyby być transportowane razem z komunikatem. Pozbycie się integracji dwóch aplikacji za pośrednictwem bazy danych zmniejszyłoby wzajemne zależności w systemie.

Ostatnim dostrzeżonym miejscem do ewentualnych zmian jest tekstowa reprezentacja formuły. Wykorzystany język Kotlin świetnie sobie radzi z tym zadaniem, zapewnia pełną elastyczność, ale stwarza też zagrożenie bezpieczeństwa. Można za jego pomocą wykonać kod na serwerze LV, więc zastosowanie własnego języka opisu wraz z parserem byłoby bezpieczniejsze (lecz zapewne bardziej ograniczające).

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.