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

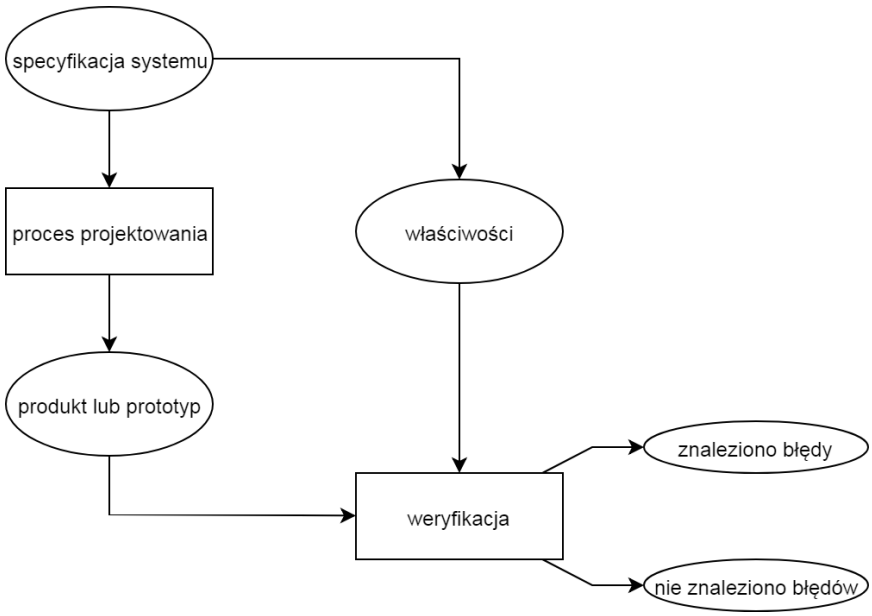
W następnym rozdziale pracy omówione zostanie zagadnienie weryfikacji modelowej w kontekście tematu pracy. Rozdział trzeci zawiera opis stworzonego rozwiązania. W czwartym rozdziale znajduje się prezentacja uzyskanych wyników. Piąty rozdział zawiera podsumowanie.

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.



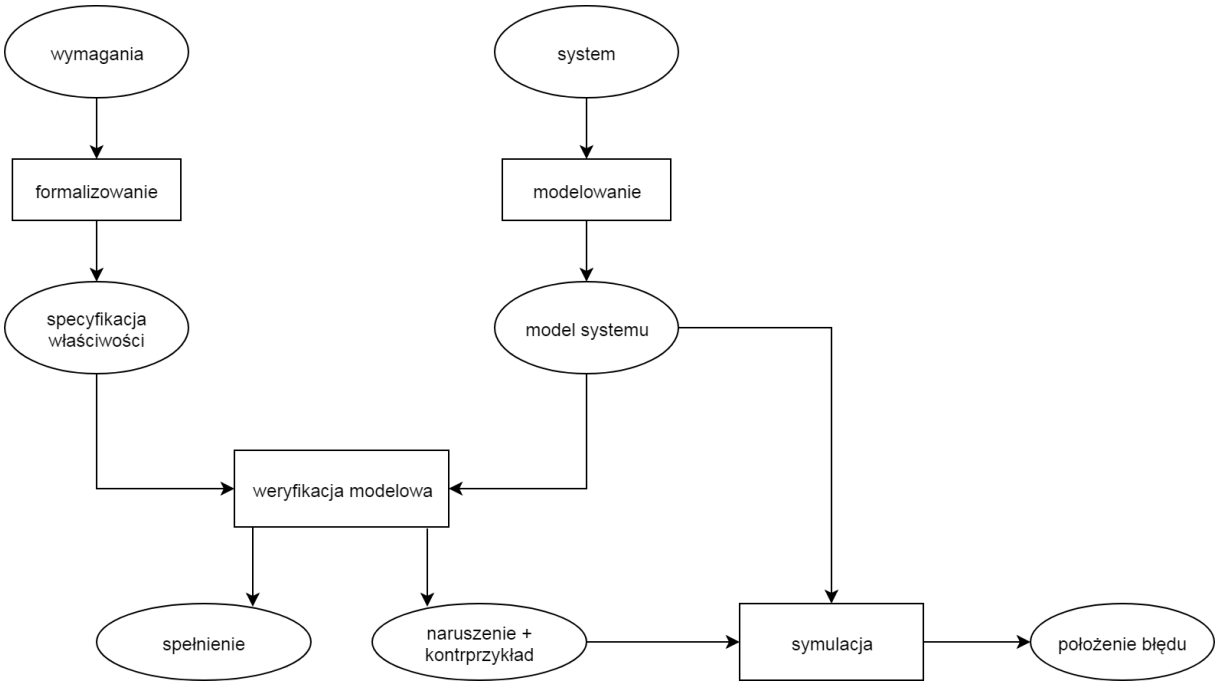
Rys. 2.1. Schemat tworzenia systemu wraz z jego weryfikacją (źródło [1]).

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:

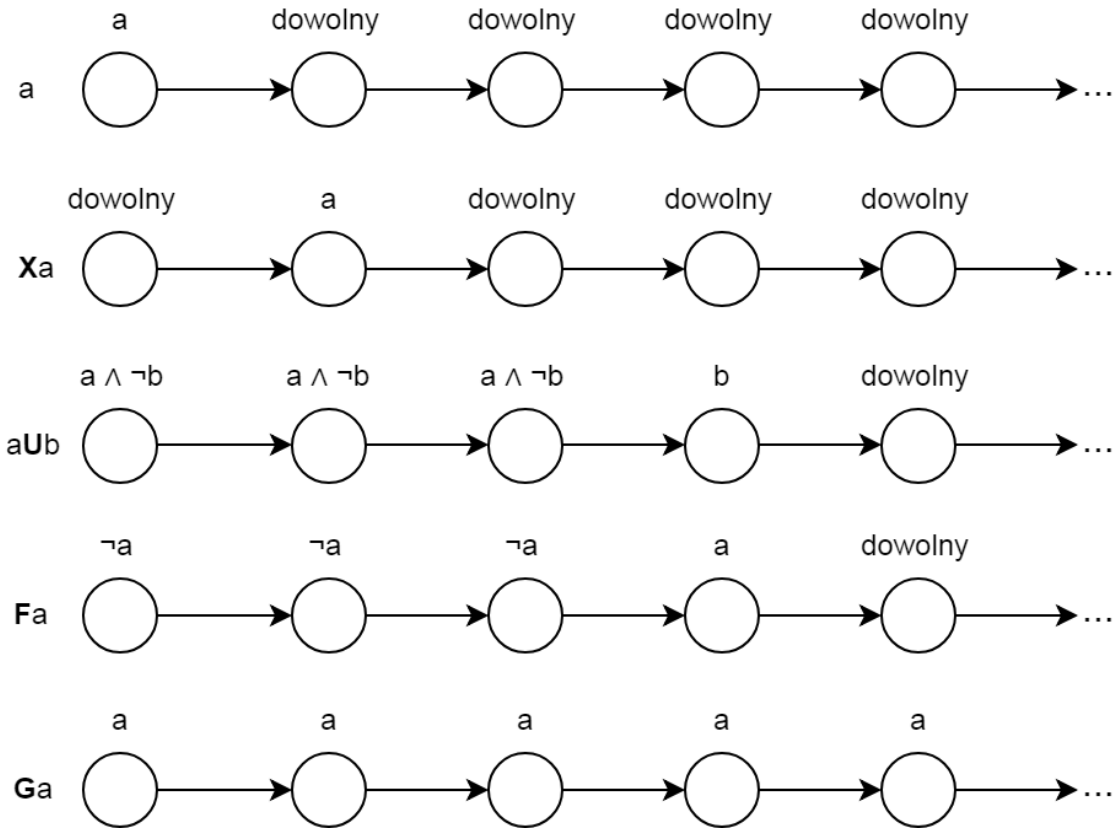
– $\mathbf{G}\neg\varphi$ - niemożliwe jest osiągnięcie stanu posiadającego φ (bezpieczeństwo)

– $\mathbf{G}(\varphi \Rightarrow \mathbf{F}\psi)$ - 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.

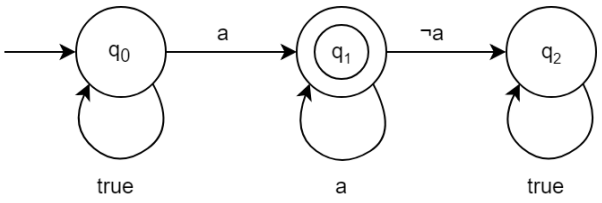
W niedeterministycznym automacie Büchiego (NBA - ang. *non-deterministic Büchi automaton*), funkcja tranzycji δ zastąpiona jest relacją tranzycji Δ, która zwraca zbiór stanów, a w miejscu stanu początkowego q_0 znajduje się zbiór stanów początkowych.

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

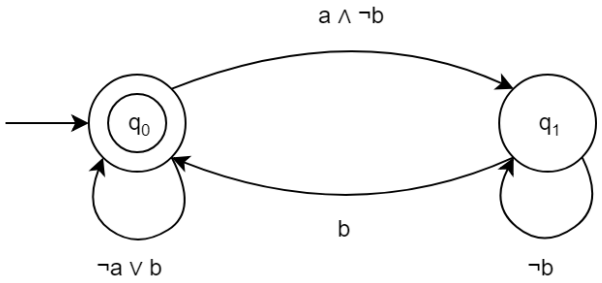
- Q - skończony zbiór. Elementy Q nazywane są stanami A.
- Σ - skończony zbiór nazywany alfabetem.
- $\delta : Q \times \Sigma \to 2^Q$ - funkcja nazywana funkcją tranzycji A.
- Q_0 - podzbiór Q, nazywany stanami początkowymi
- F - warunek akceptujący. Składa się z zera lub więcej akceptujących zbiorów.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



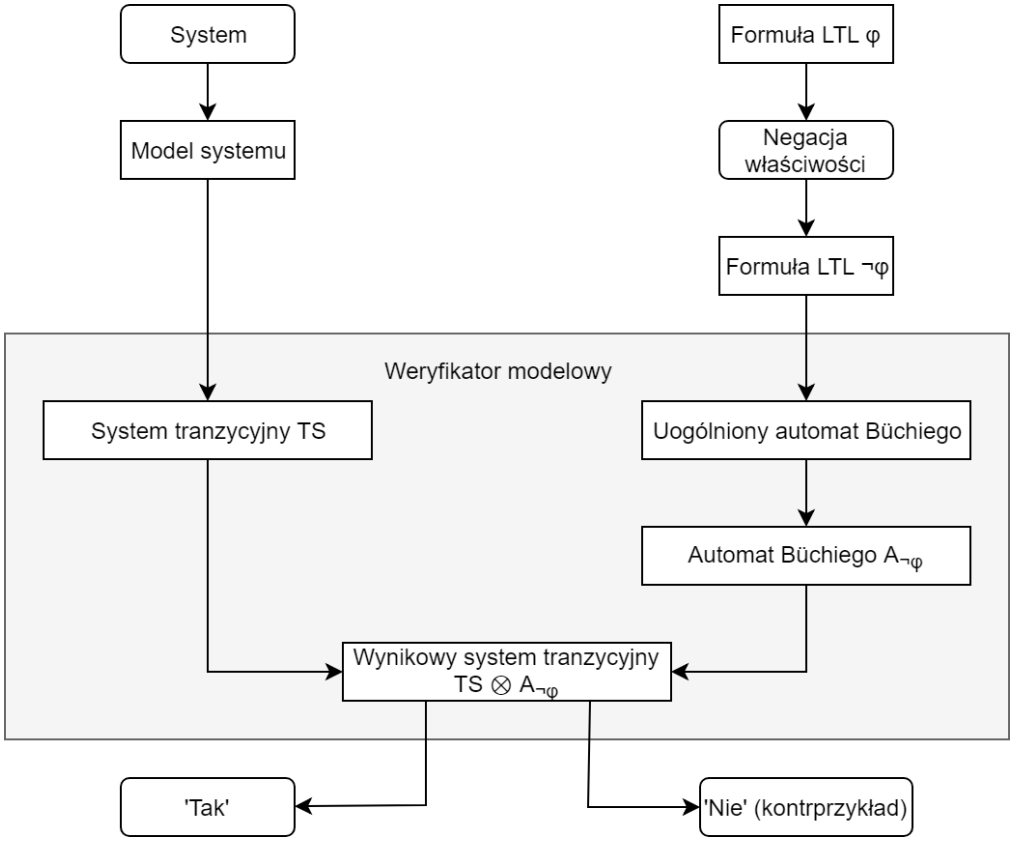
Rys. 2.5. NBA dla $\mathbf{G}(a \Rightarrow \mathbf{F}b)$ (źródło [1]).

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 algorithm* [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.

3. Implementacja

3.1. Ignore

TODO

przestrzen stanow, automaty, automaty buchiego (przykladowe obrazki automatow z podanej formuly LTL, opis ze potrzebna negacja i szukanie spelnialnosci) weryfikacja hardware i software alogrytmy wykorzystywane w przeszukiwaniu i weryfikacji co to i po co on-the-fly Alvis, co to, po co

przejrzec publikacje profesora

w czesci implementacji: opis weryfikowanego systemu

A time-optimal on-the-fly parallel algorithm for model checking of weak LTL properties, in: Formal Methods and Software Engineering

4. Prezentacja wyników

5. Podsumowanie

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.