КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Кваліфікаційна робота
на здобуття ступеня магістра

за спеціальністю 121 Інженерія програмного забезпечення

на тему:

АЛГОРИТМ ПЕРЕВІРКИ ПУСТОТИ МОВИ, ЯКА ВІДПОВІДАЄ ЗАДАНИМ СПЕЦИФІКАЦІЯМ ЛІНІЙНО ТЕМПОРАЛЬНОЇ ЛОГІКИ Й АКЦЕПТУЄТЬСЯ АВТОМАТАМИ БЮХІ, МЕТОДОМ ПОШУКУ В ГЛИБИНУ

Виконав студент 2-го курсу магістратури
Максим ГАЛЬЧЕНКО _____
(підпис)

Науковий керівник:
професор, доктор фіз.-мат. наук
Сергій КРИВИЙ _____
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем «____» _____ 2021 р., протокол № ___
Завідувач кафедри
Олександр ПРОВОТАР _____
(підпис)

Київ – 2021

РЕФЕРАТ

Обсяг роботи 64 сторінки, 10 ілюстрацій, 1 таблиця, 12 використаних джерел, 3 додатки.

Ключові слова: НЕДЕТЕРМІНОВАНИЙ, АВТОМАТ, БЮХІ, TWO-STACK, NESTED, NGA, NBA, ВЕРИФІКАЦІЯ, MODEL CHECKING, ЛІНІЙНО ТЕМПОРАЛЬНА ЛОГІКА, C++.

Об'єктом роботи є дослідження процесу верифікації властивостей лінійно-темпоральної формули за допомогою автоматів Бюхі, використовуючи алгоритм пошуку у глибину для перевірки пустоти мови.

Метою кваліфікаційної роботи є створення крос-платформового програмного забезпечення для визначення пустоти мови в автоматах, що є частиною процесу верифікації.

Методи розробки: Інструменти розроблення: безкоштовне, вільно поширюване інтегроване середовище розробки CLion 2020, мова програмування C++ 20 з використанням лише вбудованих STL бібліотек, крос-платформовий відкритий генератор сценаріїв складання CMake 3.14, система документування початкового коду програм Doxygen.

Результати роботи: Реалізовано алгоритми для перевірки пустоти мови яка акцептується автоматом Бюхі на основі алгоритмів Nested для NBA та Two-stack для NBA та NGA. Розроблена підсистема для перетворень автоматів NGA в NBA. Проаналізовано переваги та недоліки різних алгоритмів. Програмний продукт може бути інтегрованим в додатки для роботи з автоматами з більш складнішим користувацьким інтерфейсом.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

— **NFA** (Nondeterministic Finite Automaton) - недетермінований кінцевий автомат.

— **NBA** (Nondeterministic Büchi Automaton) - недетермінований автомат Бюхі.

— **NGA** (Nondeterministic Generalized Büchi Automaton) - недетермінований узагальнений автомат Бюхі.

— **DFS** (Depth-first search) - пошук у глибину.

— **ЛТЛ** (LTL - Linear Temporal Logic) - лінійно-темпоральна логіка.

— **Model checking** - перевірки моделі на відповідність заданим специфікаціям.

— **AP** (atomic propositions) - непуста множина атомарних пропозиційних формул.

— **ω-мова** - найменший клас мов, який включає всі слова скінченної і нескінченної довжини в заданому алфавіті і замкнутий відносно операцій об'єднання, перетину, конкатенації і сильної ітерації.

ВСТУП

Оцінка сучасного стану об'єкта розробки.

Сьогодні існує багато складних систем, які розв'язують різні задачі. В сучасному світі є багато реалізацій тієї чи іншої ідеї, але кожна з них чимось відрізняється. Споживач буде зацікавлений в тій, яка буде найзручнішою або кращою для нього. Одним з таких критеріїв є час. Чим швидше система справляється з посталеною задачею, тим кращою вона вважається. Але кожна система складається з підсистем, кожен її складний крок розкладається на більш простий. Одним з таких кроків є саме обробка моделей автоматів, їх властивостей, вирішення яких буде розглядатися.

Model checking часто використовуються для перевірки критичних систем, які вимагають високої степені надійності при функціонуванні. В роботі описуються методи для тестування моделей реальних систем специфікованих формулами лінійно-темпоральної логіки з перевіркою виконуваності цих умов автоматами Бюхі, зосереджується увага, зокрема, на перетворенні ЛТЛ-формули в автомати Бюхі. Пропонується програмне забезпечення яке ґрунтується на порівнянні реалізацій алгоритмів перевірки пустоти мови, яка приймається відповідним автоматом Бюхі побудованим за ЛТЛ-формулою і моделлю системи.

Нижче будуть наведені приклади використання систем рівнянь:

— **Програмування апаратного забезпечення:** в залежності від приладу ми маємо певну кількість входів (змінних) в систему та повинні отримувати значення на виході за певним логічним алгоритмом. Маючи інформацію про зв'язки компонентів, ми можемо побудувати математичну модель у вигляді автоматної системи.

— **Систематизація:** налагодження складних процесів в корпоративних компаніях.

— **Промисловість:** спрощення виробничих ліній на заводах і тому поді-

бних установах.

Актуальність роботи та підстави для її виконання.

Верифікація програмного забезпечення - це фундаментальний розділ інформатики, який ґрунтується на формальних методах. Вивчення формальних методів пов'язане зі складністю сприйняття і часто це є наслідком відірваності цих методів від практики. В середовищі програмістів поширена думка, що формальні методи не мають ніякого реального застосування при розв'язанні практичних проблем і що вони в певному сенсі є безплатним додатком до існуючих технік програмування. Підхід, який називається верифікацією на моделі (model checking), спростовує цю думку, незважаючи на те, що він ґрунтується на таких абстрактних поняттях, як темпоральні логіки, моделі Кріпке, ω-автомати, ω-мови, нерухомі точки тощо. На основі цих формалізмів працюють автоматичні системи верифікації, які дають можливість розробникам систем шляхом "натискання клавіші"верифікувати системи, які повинні працювати в критичних пристроях. До таких пристроїв належать бортові системи управління космічними станціями та супутниками, вбудовані системи військового призначення, мобільні телефони, медична апаратура, системи керування автомобілем тощо. У зв'язку з такою ситуацією методи model checking називають "реабілітацією формальних методів оскільки формальні моделі дали можливість отримати результати, без застосування яких неможливо виконати довільну практичну розробку програмних і апаратних систем для серйозних застосувань в критичних областях. [12]

Model checking часто використовуються для перевірки властивостей систем, що відповідають безпеці або бізнесу. Тому важливо, щоб ми могли довіряти результатам перевірки моделі, отриманим в результаті верифікації системи. Це ставить високі вимоги до надійності впровадження перевірки моделей. Однак повна перевірка складного програмного забезпечення такого роду, особливо коли вона реалізована на мові програмування загального призначення, такій як C++, все ще недосяжна у сучасних відкритих ресурсах з програмним забезпечення. Тим не менше, навіть методи лише часткового підвищення надійності

model checkers все одно будуть вітатися.

Мета й завдання роботи. Метою кваліфікаційної роботи є створення крос-платформового програмного забезпечення для перевірки пустоти мови (верифікації), яка приймається відповідним добутком автоматів.

Перевірка моделі: специфікованою формулами лінійно-темпоральної логіки (ЛТЛ) може бути здійснюється за допомогою теоретико-автоматного підходу [5]. Цей метод перевірки моделі використовує перетворення властивостей, виражених у ЛТЛ, у автомати над словами нескінченної довжини (автомати Бюхі), які потім використовуються для визначення того, чи відповідає дана модель системи даній властивості ЛТЛ.

Для досягнення цієї мети поставлено такі завдання:

— Формалізувати представлення ЛТЛ-формули у програмному додатку.

— Побудувати відповідний до заданої формули автомат.

— Дослідити існуючи алгоритми на основі пошуку у глибину для знаходження порожнечі.

— Розробити структурну модель даних для спільного використання в алгоритмах за допомогою перетворювань автоматів до канонічної NBA форми.

— Реалізувати та порівняти алгоритми.

— Реалізувати генератор випадкових автоматів зі зростаючим числом станів.

— Реалізувати консольний додаток.

Об'єкт, методи й засоби розроблення. Об'єктом розробки програмного засобу є побудова моделі реальної системи (моделі Кріпке) та автоматної моделі ЛТЛ-специфікації з перевіркою пустоти мови, яка приймається цими автоматами. Побудови автомату за цією формулою і передача його у програму з обчисленням його властивостей методами пошуку у глибину.

Розробка програмного засобу націлена на верифікацію систем, які будуються за формалізованим представленням[3].

В якості інструменту створення програмного засобу було обрано JetBrains

CLion - інтегроване середовище розробки (IDE) мовою програмування C++. Використано лише вбудовані шаблонні бібліотеки STL. Для забезпечення кросплатформовості використовувався відкритий генератор сценаріїв складання проектів CMake. Для майбутньої підтримки розробки коду було згенеровано документацію початкового коду програмою Doxygen.

Можливі сфери застосування. Програмний продукт може застосовуватися в повсякденному житті користувачів. Може бути інтегрований в додатки для оптимізації складних систем з великою кількістю станів і зв'язків між ними. А також задля верифікації моделей складних систем.

РОЗДІЛ 1
АВТОМАТИ БЮХІ

1.1. Регулярні вирази: мова для опису мов

Алфавіт - це скінченний, непустий набір. Елементи алфавіту називаються *буквами*. Кінцева, можливо порожня послідовність літер - це *слово*. Слово $a_1, a_2 \dots a_n$ має *довжину* n. Порожнє слово - слово довжиною 0, і воно позначається ϵ. Об'єднання двох слів $w_1 = a_1 \dots a_n$ і $w_2 = b_1 \dots b_m$ - це слово $w_1 w_2 = a_1 \dots a_n b_1 \dots a_m$, іноді також позначається $w_1 \cdot w_2$. Зверніть увагу, що $\epsilon \cdot = w = w \cdot \dots \epsilon = w$. Для кожного слова w визначаємо $w^0 = \epsilon$ та $w^{k+1} = w^k w$.

Враховуючи алфавіт $\sum$, ми позначаємо $\sum^*$ набір усіх слів над $\sum$. Набір $L \subseteq \sum^*$ слів - це *мова* над $\sum$.

Доповненням мови L є мова $\sum^* \ L$, яку часто позначають $\bar{L}$ (зауважте, що це позначення неявно передбачає, що алфавіт$\sum$ є фіксованим). *Конкатенація* двох мов L_1 і L_2 дорівнює $L_1 \cdots L_2 = \{w_1 w_2 \in \sum^? | w_1 \in L_1, w_2 \in L_2\}$. *Ітерацією* мови $L \subseteq \sum^*$ мова $L^* = \cup_{i \geqslant 0} L^i$, де $L_0 = \{\epsilon\}$ і $L^{i+1} = L^i \cdots L$ для кожного $i \geqslant 0$.

Автомати використовуються для представлення наборів об'єктів, закодованих як мови. Мови можна математично описати, використовуючи стандартні позначення теорії множин, але це часто громіздко. Для стислого опису простих мов регулярні вирази часто є найбільш придатним позначенням.

Визначення 1.1. *Регулярні вирази r над алфавітом $\sum$ визначаються наступною граматикою, де $a \in \sum$*

$$r ::= \varnothing | \varepsilon | a | r_1 r_2 | r_1 + r_2 | r^* | r^\omega \tag{1.1}$$

Набір усіх регулярних виразів над $\sum$ записується $RE(Z)$. Мова $L(r) \subseteq \sum^*$ регулярного виразу $r \in RE(\sum)$ визначається індуктивно:

1. $L(\varnothing) = \varnothing$

2. $L(\varepsilon) = \{\varepsilon\}$
3. $L(a) = \{a\}$
4. $L(r_1 r_2) = L(r_1) \cdot L(r_2)$
5. $L(r_1 + r_2) = L(r_1) \cap L(r_2)$
6. $L(r^*) = L(r)^*$
7. $L^\omega = L(r)^\omega$

Мова L є ω-регулярною, якщо існує ω-регулярний вираз r такий, що $L = L(r)$.

Ми іноді зловживатимемо мовою та визначатимемо регулярний вираз та його мову. Наприклад, коли немає загрози плутанини, ми пишемо "мову r"замість "мову $L(r)$".

1.2. NBA та NGA

Формально детермінований автомат Бюхі - це кортеж $A = (Q, \Sigma, \delta, q_0, F)$, який складається з таких компонентів:

1. Q - скінченна множина. Елементи Q називаються станами .
2. Σ - скінченна множина, яка називається алфавітом A.
3. $\delta : Q \times \Sigma \longrightarrow Q$ - функція, яка називається функцією переходу .
4. q_0 - елемент Q, який називається початковим станом A.
5. $F \subseteq Q$ - умова кінця (прийняття). A приймає саме ті слова, в якому принаймні один кінцевий стан з F зустрічається нескінченно часто.

У недетермінованому автоматі Бюхі перехідна функція δ замінюється перехідним відношенням δ, яке повертає набір станів, а єдиний початковий стан q_0 замінюється набором I початкових станів. Взагалі термін Бюхі-автомат без класифікатора відноситься до недетермінованих автомати Бюхі A.1.

Автомати Бюхі мають той же синтаксис, що і NFA, але інше визначення прийняття. Припустимо, що NFA $A = (Q, \Sigma, \delta, Q_0, F)$ задано як введення нескінченного слова $w = a_0 a_1 a_2 \ldots \Sigma$. Інтуїтивно, запуск A на w ніколи не закінчується, і тому ми не можемо визначити прийняття з точки зору стану, досягнутого в кінці проходу. Насправді навіть назва "кінцевий стан"вже не підходить

для автоматизованих машин Бюхі. Отже, відтепер ми говоримо про "приймаючі стани хоча ми все ще позначаємо набір прийнятих станів F. Ми говоримо, що запуск автомата Бюхі приймає, якщо якийсь приймаючий стан відвідується по ходу нескінченно часто. Оскільки набір станів, що приймають, є кінцевим, "деякий приймаючий стан відвідується нескінченно часто"еквівалентно "набору станів приймання нескінченно часто"[1].

1.3. Перетворення

Узагальнені автомати Бюхі - це розширення Бюхі-автоматів, зручних для здійснення деяких операцій, наприклад, перетину. Узагальнений автомат Бюхі (NGA) відрізняється від автомата Бюхі своїм кінцевим станом. Замість набору F кінцевих станів NGA має набір множин кінцевих станів $\pounds = F_0, ..., F_{m-1}$. Запуск ρ кінцевий, якщо для кожного набору $F_i \in \pounds$ деякий стан F_i відвідується ρ нескінченно часто. Формально ρ кінцевий, якщо$\inf(\rho) \cap F_i \neq \varnothing$ для кожного $i \in 0, \ldots, m-1$. Зловживаючи мовою, ми говоримо про узагальнений стан Бюхі $\pounds$. Звичайні автомати Бюхі відповідають окремому випадку $m = 1$. NGA з n станами і m наборами кінцевих станів може бути переведена в NBA з mn станами. Перехід базується на наступному спостереженні: пробіг ρ відвідує кожен набір $\pounds$ нескінченно тоді і лише у тому випадку, якщо виконуються дві наступні умови:

1. ρ зрештою відвідує F_0
2. для кожного $i \in 0, \ldots, m-1$, за кожним відвідуванням ρ до F_iв кінцевому підсумку слідує подальше відвідування $F_{i\oplus 1}$, де $\oplus$ позначає додавання по модулю m. (Між відвідуваннями F_iта $F_{i\oplus 1}$ може бути довільно багато відвідувань інших наборів $\pounds$.)

Це передбачає взяти для NBA m "копії"NGA, але з модифікацією: NBA "перескакує"з i на $i\oplus 1$ копію кожного разу, коли вона відвідує стан F_i. Точніше, переходи i-ї копії, що залишає стан F_i, переспрямовуються з i-ї копії на $i \oplus 1$-ву копію. Таким чином, відвідування кінцевих станів першої копії нескінченно

часто рівнозначне відвідуванню кінцевих станів кожної копії нескінченно часто. Більш формально, стани NBA - пари $[q, i]$, де q - стан NGA і $i \in 0, \ldots, m-1$. Інтуїтивно $[q, i]$ є i-ї копією q. Якщо $q \notin F_i$, то наступниками $[q, i]$ є стани i-ї копії, а в іншому - стани $i \oplus 1$-ої копії 1.1.

NGAtoNBA(A)
Input: NGA $A = (Q, \Sigma, Q_0, \delta, \mathcal{F})$, where $\mathcal{F} = \{F_0, \ldots, F_{m-1}\}$
Output: NBA $A' = (Q', \Sigma, \delta', Q'_0, F')$

```
1  Q', δ', F' ← ∅; Q'_0 ← {[q_0, 0] | q_0 ∈ Q_0}
2  W ← Q'_0
3  while W ≠ ∅ do
4     pick [q, i] from W
5     add [q, i] to Q'
6     if q ∈ F_0 and i = 0 then add [q, i] to F'
7     for all a ∈ Σ, q' ∈ δ(q, a) do
8        if q ∉ F_i then
9           if [q', i] ∉ Q' then add [q', i] to W
10          add ([q, i], a, [q', i]) to δ'
11       else /* q ∈ F_i */
12          if [q', i ⊕ 1] ∉ Q' then add [q', i ⊕ 1] to W
13          add ([q, i], a, [q', i ⊕ 1]) to δ'
14 return (Q', Σ, δ', Q'_0, F')
```

Рис. 1.1: Псевдокод алгоритму перетворення з NGA до NBA (Б.1)

РОЗДІЛ 2
ПЕРЕВІРКА МОВИ НА ПУСТОТУ АЛГОРИТМАМИ DFS

2.1. Перевірка мови на пустоту

Тут я буду показувати ефективні алгоритми перевірки мови на пустоту. Визначимо NBA $A = (Q, \Sigma, \delta, Q_0, F)$. Оскільки мітки переходу не мають значення для перевірки пустоти, у цій проблемі ми замінюємо δ з підмножини $Q \times \delta \times Q$ у підмножину $Q \times Q$ наступним чином:

$$\delta := \{(q, q^{'}) \in Q \times Q | (q, a, q^{'}) \in \delta, a \in \Sigma\} \tag{2.1}$$

Оскільки в багатьох додатках нам доводиться мати справу з дуже великими автоматами Бюхі, нас цікавлять алгоритми, що працюють нальоту, які не вимагають знати автомат Бюхі заздалегідь, але перевіряють пустоту при його конструюванні. Точніше, ми припускаємо існування "оракула який за умови стану q повертає множину $\delta(q)$. Нам потрібно кілька графо-теоретичних понять. Якщо $(q, r) \in \delta$, то r є спадкоємцем a і q є попередником r. Шлях - це послідовність q_0, q_1, δ, q_n таких станів, що $q_i + 1$ є наступником q_i для кожного $i \in \{0, \ldots, n-1\}$; ми говоримо, що шлях веде від q_0 до q_n. Зауважте, що шлях може складатися лише з одного стану; в цьому випадку шлях порожній і веде від стану до себе. Цикл - це шлях, який веде від стану до себе. Пишемо q ??r, щоб позначити, що існує шлях від q до r. Зрозуміло, що A не пустий, якщо він має кінцеве ласо, тобто шлях $q_0 q_1 \ldots q_{n-1} q_n$ таким, що $q_n = q_i$ для деякого $i \in 0, \ldots, n-1$, і принаймні один з $\{q_i, q_{i+1}, \ldots, q_{n-1}\}$ кінцевий. Ласо складається зі шляху $q_0 \ldots q_i$, за яким слідує пустий цикл $q_i q_{i+1} \ldots q_{n-1} q_i$. Нас цікавлять перевірки пустоти, що на вході звітують **пустий** або **непустий**, а в останньому випадку повертають кінцеве ласо як свідка недопущення.

Далі буде описано два алгоритми перевірки на пустоту, які досліджують A за допомогою першого пошуку у глибину (DFS). Почнемо з короткого опису по-

шуку у глибину та деяких його властивостей. Перший пошук по глибині (DFS) A починається з початкового стану q_0. Якщо в поточному стані q все ще є недосліджені вихідні переходи, вибирається один з них. Якщо перехід призводить до ще не виявленого стану r, r стає поточним станом. Якщо всі q-вихідні переходи були досліджені, то пошук "повертає"до стану, з якого було виявлено q, тобто цей стан стає поточним. Процес триває до тих пір, поки q_0 знову не стане поточним станом і всі його вихідні переходи будуть вивчені. $d[q]$, записує, коли q спочатку розкрито, а друге, $f[q]$, записує, коли пошук закінчується, вивчаючи вихідні переходи q.

Зауважте, що DFS недетермінований, тому що ми не фіксуємо порядок, у якому стани $\delta(q)$ досліджуються циклом. Загальновідомо, що графік зі станами як вузли і ці переходи як ребра - це дерево з коренем q_0, яке називається DFS-деревом. Якщо деякий шлях дерева DFS веде від q до r, то ми говоримо, що q - це предок r, а r - нащадок q (у дереві). DFS легко змінити так, щоб він повертав дерево DFS разом із часовими позначками для станів.

Теорема 2.1 (Теорема Інтервалів)**.** *У DFS-дереві для будь-яких двох станів q і r виконується точно одна з наступних чотирьох умов, де $I(q)$ позначає інтервал $(d[q], f[q]]$ і $I(q) \prec I(r)$ позначає, що виконується $f[q] < d[r]$.*

1. *$I(q) \subseteq I(r)$ і q - нащадок r;*
2. *$I(r) \subseteq I(q)$ і r - нащадок q;*
3. *$I(q) \prec I(r)$, і ні q не є нащадкомr, ні r не є нащадком q;*
4. *$I(r) \prec I(q)$, і ні q не є нащадком r, ні r - нащадок q.*

Теорема 2.2 (Теорема білого контуру)**.** *У дереві DFS r є нащадком q (і так $I(r) \subseteq I(q)$) якщо і лише тоді, коли в момент $d[q]$ стану r можна досягти від q в A по шляху білих станів.*

2.2. Nested алгоритм

Щоб визначити, чи A пустий, ми можемо здійснити пошук прийнятих станів A і перевірити, чи принаймні один із них належить до циклу. Наявна реалізація

проходить у дві фази, шукаючи сприйняття станів у першій, а циклів у другій. Час виконання квадратичний: оскільки автомат з n станами і m переходами має $O(n)$ кінцевих станів, і оскільки пошук циклу, що містить заданий стан, займає час $O(n+m)$, ми отримуємо $O(n^2+nm)$. Вкладений алгоритм DFS працює в часі $O(n+m)$, використовуючи першу фазу не тільки для виявлення доступних кінцевих станів, але і для їх сортування. Пошуки другої фази проводяться у порядку, визначеному сортуванням. Як ми побачимо, проведення обшуку в цьому порядку дозволяє уникнути повторних відвідувань того ж стану. Перша фаза виконується DFS, і приймаючі стани сортуються за збільшенням часу виходу (не відкриття!). Це відоме як зворотній обхід DFS. Припустимо, що на другій фазі ми вже здійснили пошук, починаючи зі стану q, який не вдався, тобто жоден цикл A не містить q. Припустимо, ми переходимо до пошуку з іншого стану r (що означає $f[q] < f[r]$), і цей пошук виявляє деякий стан s, який вже був виявлений пошуком з q. Ми стверджуємо, що не потрібно знову досліджувати наступників s. Точніше, ми стверджуємо, що $s???r$, і тому марно досліджувати наступників s, оскільки дослідження не може повернути жоден цикл, що містить r. Доказ твердження ґрунтується на наступній лемі 2.3:

Лема 2.3. *Якщо $q \rightsquigarrow r$ і $f[q] < f[r]$ в якомусь дереві DFS, то деякий цикл A містить q.*

Введення другого пошукового запиту Нагадаємо, що ми шукаємо алгоритми, які повертають кінцеве ласо, коли A непустий. Описаний нами алгоритм для цієї мети не підходить. Визначення шляху DFS для стану як унікальний шлях дерева DFS, що веде від початкового стану до нього. Коли друга фаза відповідає **непустий**, шлях DFS стану, що досліджується, скажімо, q, є циклом приймання, але зазвичай не приймає ласо. Для прийняття ласо ми можемо приєднати цей шлях за допомогою DFS-шляху q, отриманого протягом першої фази. Однак, оскільки перша фаза не може передбачити майбутнє, вона не знає, який кінцевий стан, якщо такий є, буде визначений другою фазою як належний кінцевому ласо. Отже, або перший пошук повинен зберігати DFS-шляхи всіх кінцевих станів, які він виявляє, або необхідна третя фаза, в якій перера-

ховується новий шлях DFS. Цю проблему можна вирішити шляхом введення першої та другої фаз: Кожен раз, коли перший DFS чорніє кінцевим станом q, ми негайно запускаємо другий DFS, щоб перевірити, чи q доступний від себе [2].

1. Виконаємо DFS з q_0
2. Щоразу, коли пошук чорніє кінцевим станом q, запускаємо новий DFS з q. Якщо ця друга фаза DFS відвідує q ще раз (тобто, якщо він досліджує деякий перехід, що веде до q), зупиниться на **непустий**. В іншому випадку, коли друга фаза DFS припиняється, продовжуємо роботу з першою DFS.
3. Якщо перший DFS припиниться, виводимо **пустий**.

ImprovedNestedDFS(A)
Input: NBA $A = (Q, \Sigma, \delta, Q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

```
1  S ← ∅; P ← ∅
2  dfs1(q0)
3  report EMP
4  proc dfs1(q)
5     add [q, 1] to S; add q to P
6     for all r ∈ δ(q) do
7        if [r, 1] ∉ S then dfs1(r)
8     if q ∈ F then dfs2(q)
9     remove q from P
10    return
11 proc dfs2(q)
12    add [q, 2] to S
13    for all r ∈ δ(q) do
14       if r ∈ P then report NEMP
15       if [r, 2] ∉ S then dfs2(r)
16    return
```

Рис. 2.1: Псевдокод алгоритму Nested (B.1)

Оцінка Сильною стороною алгоритму Nested DFS є його дуже скромні ви-

моги до простору. Крім простору, необхідного для зберігання стека викликів для рекурсивної процедури dfs, алгоритму потрібно лише два додаткових біта для кожного стану A. У багатьох практичних програмах A може легко мати мільйони або десятки мільйонів станів, і кожен стан може знадобитися багато байтів для зберігання. У цих випадках два зайвих біта на стан незначні. Але алгоритм також має дві важливі слабкі сторони: його не можна поширити на NGA, і він не є оптимальним у формальному сенсі, визначеному нижче.

Алгоритм вкладеного DFS працює, спочатку визначаючи кінцеві стани, а потім перевіряючи, чи належать вони до певного циклу. Цей принцип більше не працює для умови прийняття NGA, де ми шукаємо цикли, що містять щонайменше один стан кожного сімейства кінцевих станів. Наразі не описана краща процедура, ніж переклад NGA в NBA. Для NGA, які мають велику кількість кінцевих множин, переклад може передбачати значні затрати при виконанні.

Алгоритм перевірки на пустоту починає з початкового стану. У кожний момент часу t алгоритм досліджує підмножину станів і переходи алгоритму, які утворюють під-NBA $A_t = (Q_t, \Sigma, \delta_t, q_0, F_t)$ з A (тобто $Q_t \subseteq Q$, $\delta_t \subseteq \delta$, і $F_t F$)). Зрозуміло, що алгоритм, заснований на пошуку, може повідомити лише про **непустий** за час t, якщо A_t містить кінцеве ласо. Алгоритм, що базується на пошуку, є оптимальним, якщо зворотне значення має місце, тобто якщо воно повідомляє про **непустий** в найкоротші терміни t таким чином, що A_t містить кінцеве ласо. Неважко помітити, що Nested-DFS не є оптимальним.

2.3. Two-stack алгоритм

Нагадаємо, що алгоритм Nested DFS здійснює пошук кінцевих станів A, а потім перевіряє, чи належать вони до певного циклу. Two-stack алгоритм проходить навпаки: він здійснює пошук станів, що належать до певного циклу A за допомогою однієї DFS, і перевіряє, чи кінцеві вони. Перше спостереження полягає в тому, що до моменту, коли DFS почорніє, він вже достатньо досліджен, щоб вирішити, чи належить він до циклу:

Лема 2.4. *Нехай A_t є суб-NBA A, що містить стани та переходи, досліджені DFS до (включаючи) часу t. Якщо стан q належить якомусь циклу A, то він вже належить до деякого циклу $A_{f[q]}$.*

Ця лема пропонує підтримувати під час DFS набір C кандидатів, що містить стани, для яких ще невідомо, належать вони до якогось циклу чи ні. Стан додається до C при його виявленні. Поки стан сірий, алгоритм намагається знайти цикл, що містить його. Якщо це вдається, то стан видаляється з C. Якщо ні, то стан видаляється з C, коли він чорніє. У будь-який момент t кандидатами є нині сірі стани, які не належать до жодного циклу A_t [8].

Лема 2.5. *Щоразу t, стек C містить пару $[q, I]$ якщо q - корінь A_t, а I - підмножина індексів $i \in K$, така що деякий стан F_i належить до S по q.*

Two-stack можна легко перетворити на перевірку пустоти для узагальнених автоматів Бюхі, які не потребують побудови еквівалентної NBA. Нагадаємо, що в NGA загалом є кілька наборів $F_0, \ldots, F_{k-1}$ кінцевих станів і що запуск ρ кінцевий, якщо $\inf \rho \cap F_i \neq \varnothing$ для кожного $i \in 0, \ldots, k-1$. Таким чином, ми маємо таку характеристику **не пустоти**, де $K = 0, \ldots, k-1$:

Факт 2.6. *Нехай A - це NGA з умовою кінця $F_0, \ldots, F_{k-1}$. A **непустий** якщо деякий S із A задовільняє умові $S \cap F_i \neq \varnothing$ для кожного $i \in K$.*

Two-stack видає **непустий** якщо A **непустий**. Більш того Two-stack оптимальний. А тепер доведемо це: Якщо Two-stack повідомляє про **непустий**, то цикл повторення у рядках 10-13 видаляє деяку пару $[q, K]$. За лемою2.5 q належить до циклу A, що містить деякий стан F_i для кожного $i \in K$. власність. Якщо A не пустий, то деякий S із A задовольняє $S \cap F_i \neq \varnothing$ для кожного $i \in K$.Так є найдавніший час t такий, що A_t містить S $t \subseteq S$, що задовольняє однакові властивості. За лемою 2.5, Two-stack повідомляє про **непустий**, за час t або раніше, і це оптимально.

Повинен відмітити, що підхід NGA або NBA легко модифікувати один до одного. Для NBA умова виходу буде відрізнятися: замість перевірки на входження елементу S в множину кінцевих станів, ми будемо перевіряти перетин множин. Легко побачити, що вони не будуть змінюватися для NBA з самого

TwoStackNGA(A)
Input: NGA $A = (Q, \Sigma, \delta, q_0, \{F_0, \ldots, F_{k-1}\})$
Output: EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

```
1   S, C, V ← ∅;
2   dfs(q_0)
3   report EMP
4   proc dfs(q)
5     add [q, F(q)] to S ; push([q, F(q)], C); push(q, V)
6     for all r ∈ δ(q) do
7       if r ∉ S then dfs(r)
8       else if r ∈ V then
9         I ← ∅
10        repeat
11          [s, J] ← pop(C);
12          I ← I ∪ J; if I = K then report NEMP
13        until d[s] ≤ d[r]
14        push([s, I], C)
15    if top(C) = (q, I)for some I then
16      pop(C)
17      repeat s ← pop(V) until s = q
```

Рис. 2.2: Псевдокод алгоритму Two-stack (B.2)

початку, коли ми підрахуємо цю множину $F(q)$.

Оцінка

Нагадаємо, що два слабкі моменти алгоритму Nested DFS полягали в тому, що він не може бути безпосередньо розширений до NGA, і це не є оптимальним. Обидва є сильними моментами Two-stack алгоритму. Сильною стороною алгоритму Nested DFS були його дуже скромні вимоги до простору: всього два зайвих біта для кожного стану A. Простір же необхідний для алгоритму Two-stack буде теж легко підрахувати. Це зручно для обчислення для **пустих** автоматів, оскільки в цьому випадку і вкладений DFS, і алгоритми з двома стеками повинні відвідувати всі стани. Через контрольну $d[s] \leqslant d[r]$ алгоритму потрібно зберігати час відкриття кожного стану. Це робиться шляхом розширення хеш-таблиці S. Якщо стан q можна зберігати за допомогою c бітів, то для зберігання $d[q]$ потрібні ;n бітів; однак на практиці $d[q]$ зберігається за допомогою "слова"пам'яті, тому що якщо число станів A перевищує 2^w, де w - кількість біт слова, то A ніяк не може зберігатися в основній пам'яті. Тож хеш-таблиця S вимагає $c + w + 1$ біт на стан (додатковий біт - це той, який використовується для перевірки членства в V). Стеки C і V зберігають не самі стани, а адреси пам'яті, в яких вони зберігаються. Ігноруючи колізії хешування, для цього потрібно $2w$ додаткових біт на стан. Для узагальнених автоматів Бюхі ми також повинні додати k біт, необхідний для зберігання підмножини K у другому компоненті елементів C. Отже, двоскладовий алгоритм використовує в цілому $c + 3w + 1 + (k)$ біт на стан порівняно з бітами $c + 2$, необхідними алгоритму Nested DFS. У більшості випадків $w \ll c$, тому вплив додаткових вимог до пам'яті на продуктивність невеликий.

РОЗДІЛ 3
АНАЛІЗ РЕАЛІЗАЦІЇ

3.1. Основні можливості програми

Програма може працювати, як в режимі обчислення заданого автомату з файлу *–infile NAME*, чи консолі, так і в режимі генерації для отримання статистики по часу виконання операцій конвертації, генерації, обчислення NBA методом Nested, обчислення NBA та NGA методом Two-stack, що дає можливість подальшого дослідження. Також програма має можливість зберігати автомат у відповідному до вводу форматі в окремий новий файл *–outfile NAME*.

3.1.1. Пояснення синтаксису. Вхідні дані.

Для того щоб обрахувати мову її потрібно задати. Правила, які необхідно знати, щоб правильно ввести автомат будуть такі:

— перше число відповідає кількості множин кінцевих станів (використовуйте 1, якщо хочете задати NBA)

— кількість наступних рядків буде дорівнювати "першому"числу. Кожне перше число рядка відповідає кількості наступних чисел для зчитування (в ньому) - ці рядки відповідають кінцевим станам кожної множини

— (опційно, якщо використовувати введення з консолі, а не з файлу) кількість зв'язків в автоматі

— зчитування зв'язків (пари чисел) до кількості попереднього числа, або до кінця файлу - тобто наявність переходу з первого стану до другого

— в програму вбудовано парсер, який буде перевіряти кожен рядок на коректність. Автомат повинен бути логічно коректним. Наприклад, не містити множини з 0 елементів

Зазначте параметр *–help* для допомоги у використанні програми з обчислення автомату. Підказка буде виглядати таким чином (3.1).

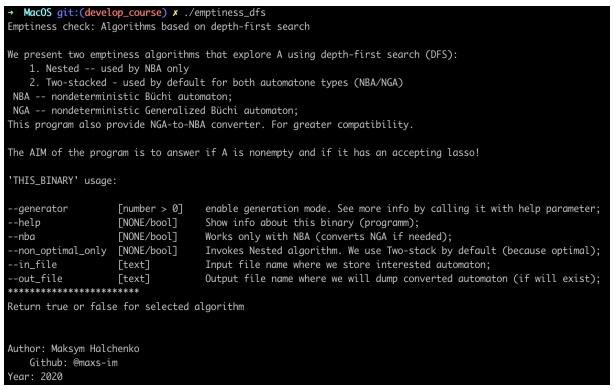


Рис. 3.1: Інтерфейс головного режиму (обчислення вводу)

Наведемо приклад введення автомату з консолі, який буде відповідати NGA з двома множинами станів 3.1

2

1 5

3 4 5 1

4

1 2

4 5

5 1

2 3

Таблиця 3.1

Приклад на введення NGA

Якщо такий автомат задати в програму, то результат буде як зазначено на зображенні 3.2.

Рис. 3.2: Вивід програми після обчислення автомату з файлу 3.1

3.1.2. Модифікації програми. Програма може працювати в режимі генерування. Для цього необхідно вказати параметр виклику *–generator* та додати

число більше 0 для вказання числа спроб на кожному кроці. Потрібно зауважити, що максимальна кількість станів у степені 10, тобто параметр *–states* обмежений 10, адже 64-бітна система не зможе зберегти більші значення. Також може не вистачити RAM обчислювальної машини, що приведе до помилки **segmentation fault**.

Приклад помічника в режимі генерації виглядає, як на малюнку 3.3.

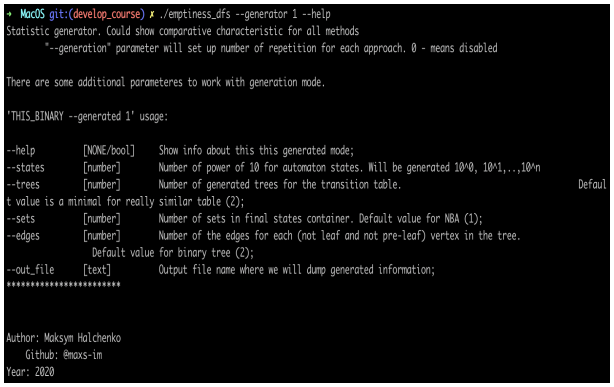
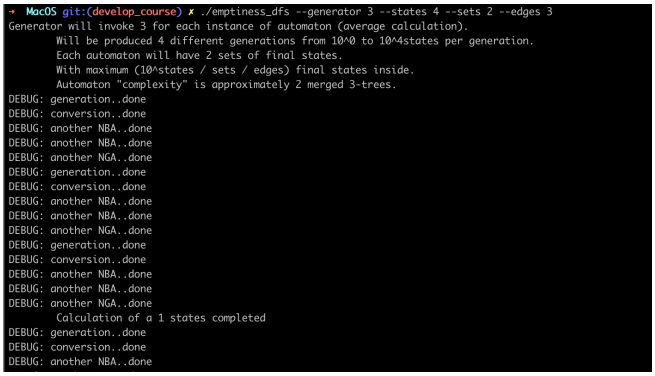


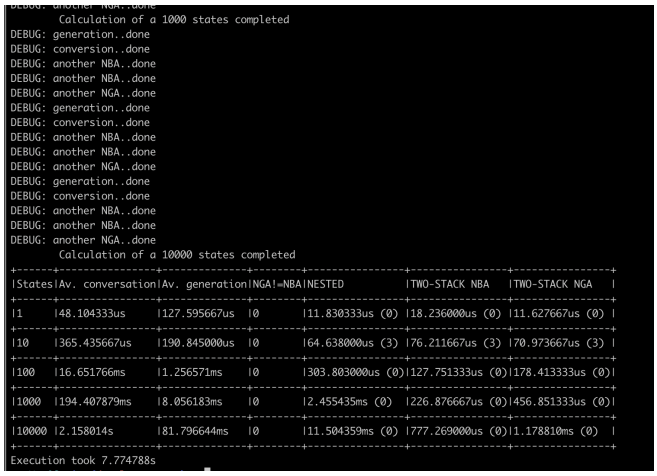
Рис. 3.3: Інтерфейс режиму генерації

3.2. Результати обчислювань на довільних вибірках

Потрійна генерація автомату з 2-ма множинами кінцевих станів, тобто всі NGA, та з кількістю станів від 1, 10, 100, 1000 та 10000 при тернарному розподілі листів на двох деревах генерації буде давати результат, як вказано на малюнку 3.4. Зверніть увагу, що для користувацького інтерфейсу програма говорить про кожний крок виконання складної операції (**DEBUG** повідомлення). Початок зазначено на малюнку 3.4а, а от підсумкові результати тут 3.4б. Кожна генерація відбувається за допомогою рандомізації станів та їх зв'язків. Фіксованою залишається лише кількість станів, яка задається на кожному кроку в степені 10. Сама ідея побудови ґрунтується на побудові та зливанні *–trees* збалансованих *–edge*-листових дерев методом проходу в ширину.



(а) Старт виконання



(б) Результат виконання

Рис. 3.4: Приклад роботи режиму генерації

РОЗДІЛ 4
ВЕРИФІКАЦІЯ І ТЕМПОРАЛЬНА ЛОГІКА

4.1. Лінійно Темпоральна Логіка

Опишемо представлення мову для визначення властивостей, яка називається лінійно темпоральна логіка. ЛТЛ близька до природної мови, але має формальну семантику.

Формули LTL будуються з безлічі AP атомарних пропозицій. Інтуїтивно атомні пропозиції - це абстрактні назви основних властивостей конфігурацій, значення яких фіксується лише після розгляду конкретної системи. Формально, враховуючи систему з набором C конфігурацій, значення атомарних пропозицій фіксується функцією оцінки $V : AP \to 2^C$, яка присвоює кожному абстрактному імені набір конфігурацій, в яких воно зберігається. Позначимо $LTL(AP)$ набором формул ЛТЛ над AP.

Атомарні пропозиції поєднуються за допомогою звичайних булевих операторів та темпоральних операторів **X** ("наступний") та **U** ("до"). Інтуїтивно, як перше наближення $\mathbf{X}\varphi$ означає "φ виконується при наступній конфігурації"(конфігурація досягається після одного кроку програми), а $\varphi\mathbf{U}\psi$ означає "φ виконується до тих пір, поки не буде досягнута конфігурація, що задовольняє ψ". Формально синтаксис $LTL(AP)$ визначається наступним чином:

Визначення 4.1. *Нехай AP - скінченний набір атомних пропозицій. $LTL(AP)$ - це набір виразів, породжених граматикою*

$$\varphi := true|p|\neg\varphi_1|\varphi_1 \wedge \varphi_2|\boldsymbol{X}\varphi_1|\varphi_1 \boldsymbol{U}\varphi_2 \tag{4.1}$$

Формули виражаються на послідовностях $\sigma = \sigma_0\sigma_1\sigma_2\ldots$, де $\sigma_i \subseteq AP$ для кожного $i \geqslant 0$. Ми називаємо ці послідовності обчисленнями. Сукупність усіх обчислень за AP позначається $C(AP)$. Виконавчими обчисленнями системи є

обчислення σ, для яких існує ω-виконання $c_0c_1c_2\ldots$ така, що для кожного $i \geqslant 0$ сукупність атомарних пропозицій для c_i є точно σ_i. Тепер ми формально визначили, коли обчислення задовольняє формулу.

Визначення 4.2. *Враховуючи обчислення $\sigma \in C(AP)$, нехай σ^j позначає суфікс $\sigma_j\sigma_{j+1}\sigma_{j+2}\ldots$ із σ. Співвідношення, яку задовільняє $\sigma \models \varphi$ (читається "σ задовольняє φ") індуктивно визначається наступним чином:*

- $\sigma \models true$.
- $\sigma \models p$ *якщо* $p \in \sigma(0)$.
- $\sigma \models \neg\varphi$ *якщо* $\sigma \not\models \varphi$.
- $\sigma \models \varphi_1 \wedge \varphi_2$ *якщо* $\sigma \models \varphi_1$ *і* $\sigma \models \varphi_2$.
- $\sigma \models \boldsymbol{X}\varphi$ *якщо* $\sigma^1 \models \varphi$.
- $\sigma \models \varphi_1 \boldsymbol{U} \varphi_2$ *якщо існує* $k \geqslant 0$ *таке що* $\sigma^k \models \varphi_2$ *і* $\sigma^i \models \varphi_1$ *для кожного* $0 \leqslant i \leqslant k$.

Будемо використовувати наступні скорочення:

- **false**, $\vee$, $\rightarrow$ та $\leftrightarrow$ інтерпретуються звичайним способом.
- $\mathbf{F}_\varphi = \mathbf{true}\mathbf{U}\varphi$ ("з часом φ"). Згідно з наведеною вище семантикою, $\sigma \models \mathbf{F}\varphi$ якщо існує $k \geqslant 0$ така що $\sigma^k \models \varphi$.
- $\mathbf{G}_\varphi = \neg\mathbf{F}\neg\varphi$ ("завжди φ"чи "глобально φ"). Згідно з наведеною вище семантикою, $\sigma \models \mathbf{G}\varphi$ якщо $\sigma^k \models \varphi$ для кожного $k \geqslant 0$.

Набір обчислень, які задовольняють формулу φ, позначається $L(\varphi)$. Система задовольняє φ, якщо всі її виконувані обчислення задовольняють φ.

4.2. Проблематика перевірки лінійно-темпоральної моделі

Мета перевірки ЛТЛ моделі полягає у визначенні, чи задовольняється властивість, виражена як формула ЛТЛ, у кінцевій моделі системи, що перевіряється. Модель може бути представлена у вигляді структури Кріпке, яка представляє систему як графік стану-переходу, кожен стан якого доповнено набором атомних пропозицій для кодування властивостей, що існують у стані.

Формально структурою Кріпке є кортеж $M = \langle S, \rho, \pi\rangle$, де

- S набір фінальних станів,
- $\rho \subseteq S \times S$ є відношенням переходів, яке задовольняє обмеженню $\forall A \in S : \exists s^{'} \in S : (s, s^{'}) \in \rho$,
- $\pi : S \to 2^{AP}$ це функція маркування, яка пов'язує кожен стан із набором атомарних пропозицій. Семантично $\pi(s)$ представляє набір пропозицій, що містяться в стані $s \in S$.

Нескінченний шлях у структурі Кріпке - це нескінченна послідовність станів $x = \langle s_0, s_1, s_2, \dots \rangle \in S^{\omega}$ такий, що для всіх $i \geqslant 0, (s_i, s_{i+1}) \in \rho$.

Легко помітити, що кожен нескінченний шлях $x = \langle s_0, s_1, s_2, \dots \rangle \in S^{\omega}$ структури Кріпке відповідає нескінченній послідовності $\xi = \langle \pi(s_0), \pi(s_1), \pi(s_2) \dots \rangle \in (2^{AP})^{\omega}$. Для зручності позначимо $\pi(x) = \xi$ і називаємо ξ тимчасовою інтерпретацією x. З цим проблему перевірки ЛТЛ моделі для структур Кріпке можна сформулювати так:

Проблема 4.3 (Формулювання проблеми лінійно-темпоральної моделі)**.** *Дана структура Кріпке $M = \langle S, \rho, \pi \rangle$ зі станом $s \in S$ і LTL-формула φ чи існує нескінченний шлях $x = \langle s_0, s_1, s_2, \dots \rangle \in S^{\omega}$ (де $s_0 = s$) таке, що $\pi(x) \models \varphi$ приймає x?*

Називатимемо це малою проблемою перевірки ЛТЛ моделі. Проблема перевірки глобальної моделі зводиться до знаходження всіх станів $s \in S$, які виконують попередню умову.

Зауважимо, що наведена вище постановка задачі перевірки ЛТЛ моделі дещо відрізняється від класичної дефініції, яка міститься в літературі [5], де проблема зазвичай представляється як питання, чи виконується якась властивість ψ для всіх кінцевих шляхів, що починаються в якомусь визначеному стані s. Відповідь на це питання можна знайти в [5], перевіривши, чи існує такий шлях x у структурі Кріпке, для якого виконується властивість $\pi(x) \models \neg\psi$; отже, властивість $\pi(x) \models \psi$ тоді виконується для всіх нескінченних шляхів, починаючи з s, лише якщо це не так. Однак це точно відповідає вирішенню задачі локальної перевірки моделі, як визначено вище для формули $\varphi = \neg\psi$.

У локальній задачі перевірки моделі даний стан визначає безліч безкінечних

шляхів, які потрібно враховувати при пошуку відповіді на проблему. Розглядаючи цей набір шляхів як мову $\zeta_{M,s} = \{\pi(x)|x$ це кінцевий шлях M, починаючи з $s\} \subseteq (2^{AP})^{\omega}$, проблема перевірки локальної ЛТЛ моделі може бути компактно виражена як питання, чи $\zeta_{M,s} \bigcap \zeta_{\varphi} \neq \varnothing$.

Крім того, структуру Кріпке $M = \langle S, \rho, \pi \rangle$ (із початковим станом $s \in S$) можна розглядати як автомат Бюхі, який розпізнає мову $\zeta_{M,s}$; справді, легко перевірити, що автомат $A_{M,s} = \langle \sum, Q, \triangle, Q^0, F \rangle$, де $\sum = 2^{AP}, Q = S, Q^0 = \{s\}, F = Q$ і $\triangle = \{(s, a, s^{'})|(s, s^{'}) \in \rho, \pi(s) = a\}$ приймає нескінченну послідовність $\xi \in (2^{AP})^{\omega}$ лише тоді, коли ξ є часовою інтерпретацією $\pi(x)$ деякого нескінченного шляху x, що починається із стану s у структурі Кріпке M .

Зараз ми знаємо, що за допомогою ЛТЛ формули φ і обраного стану $s \in S$, Кріпке-структуру M можна перетворити на автомат Бюхі A_{φ} та $A_{M,s}$, які розпізнають мови L_{φ} та $L_{M,s}$ відповідно. Отже, рішення проблеми локальної перевірки ЛТЛ моделі у стані s можна знайти, спочатку побудувавши автомат Бюхі $A_{M,s} \otimes A_{\varphi}$, який приймає мову $L_{M,s} \cap L_{\varphi}$, а потім перевіривши, чи приймає цей автомат якусь кінцеву послідовність над 2^{AP}. Цей тест, який зазвичай називають перевіркою пустоти автомата, що можна виконати за допомогою алгоритму, який перевіряє, чи може автомат перетину досягти з будь-якого з його початкових станів циклу стану, що проходить через якийсь прийнятний стан. Час, необхідний для цієї перевірки, є лінійним за розміром перетину автомата. Вимоги до пам'яті при перевірці моделей часто можна покращити, використовуючи поточні методи перевірки моделей [[6], [10], [7], [11]], які здатні виявляти порушення ЛТЛ властивостей, не будуючи автомат перетину $A_{M,s} \otimes A_{\Phi}$ явно поєднуючи побудову автоматів $A_{M,s}$ та A_{Φ} з алгоритмом перевірки пустоти. Однак усі фактичні поточні інструменти перевірки ЛТЛ моделі, які ми знаємо, як і раніше, виділяють конструкцію A_{Φ} (тобто фазу перетворення ЛТЛ в Бюхі) в окремий програмний модуль.

Найпростішим способом вирішення глобальної проблеми перевірки ЛТЛ моделі є вирішення проблеми перевірки локальної моделі окремо для кожного стану s структури Кріпке. Ця процедура вимагає в цілому $|S|$ композиції формули

автомата A_Φ з деяким автоматом $A_{M,s}$ побудованим із структури Кріпке; кожна з окремих композицій призводить до автомата із станами $O(|Q| \cdot |S|)$ у гіршому випадку.

Можна вдосконалити процедуру глобальної перевірки моделі, так що автомат формули A_Φ повинен бути складений зі структурою Кріпке лише один раз, зберігаючи при цьому найгіршу верхню оцінку для числа станів $O(|Q| \cdot |S|)$ в результаті композиції. В основному, композицію можна зробити ефективніше, поділившись загальними підструктурами, які виникали б в окремих «малих» композиціях. По суті, модифікований склад відповідає перетину автомата формули з автоматом Бюхі A_M, який можна отримати з будь-якого $A_{M,s}$ шляхом розширення його початкового набору станів, щоб охопити всі стани автомата. Більш детальний опис цієї простої ідеї, а також зміни, які вона вимагає щодо перевірки пустоти, можна знайти в [4].

4.3. Від ЛТЛ формул до NGA

Покажемо алгоритм, який, отримуючи формулу $\varphi \in LTL(AP)$, повертає NGA A_φ над алфавітом 2^{AP}, що розпізнає $L(\varphi)$, а потім виводить повністю автоматичну процедуру, яка з урахуванням системи та ЛТЛ формули вирішує, чи виконувані обчислення системи задовольняють формулі.

4.3.1. Задовільна послідовність та послідовності Гінтікки (Hintikka). Визначимо задовільну послідовність ті послідовність Гінтікки з обчислення ω та формули φ. Спочатку потрібно ввести поняття замикання формули та атомарного замикання.

Визначення 4.4. *За умови формули φ заперечення φ є формулою ψ, якщо $\varphi = \neg\psi$, а формулою $\neg\varphi$ - в іншому випадку. Замикання $cl(\varphi)$ формули φ - це множина, що містить усі підформули φ та їх заперечення. Непуста множина $\alpha \subseteq cl(\varphi)$ є атомом $cl(\varphi)$, якщо вона задовольняє наступним властивостям:*

(a0) *Якщо* $\boldsymbol{true} \in cl(\varphi)$, *то* $\boldsymbol{true} \in \alpha$.

(a1) *Для кожного* $\varphi_1 \wedge \varphi_2 \in cl(\varphi) : \varphi_1 \wedge \varphi_2 \in \alpha$ *тільки якщо* $\varphi_1 \in \alpha$ *і* $\varphi_2 \in \alpha$.

(a2) *Для кожного* $\neg\varphi_1 \in cl(\varphi) : \neg\varphi_1 \in \alpha$ *тільки якщо* $\varphi_1 \notin \alpha$.

де множина всіх атомів з $cl(\varphi)$ *визначається з* $at(\varphi)$.

Звернемо увагу, що якщо α - набір усіх формул $cl(\varphi)$, що задовільняють обчислення σ, то α обов'язково є атомом. Дійсно, кожне обчислення задовольняє **true**; якщо обчислення задовольняє сполученню двох формул, то воно задовольняє кожному зі сполучників; нарешті, якщо обчислення задовольняє формулу, то воно не задовольняє її заперечення, і навпаки. Також зауважте, що із (a2) випливає, що якщо $cl(\varphi)$ містить k формул, то кожен атом $cl(\varphi)$ містить точно $k/2$ формул.

Визначення 4.5. *Задовільна послідовність для обчислення* σ *і формули* φ *є нескінченною послідовністю атомів*

$$sats(\sigma, \varphi) = sats(\sigma, \varphi, 0)sats(\sigma, \varphi, 1)sats(\sigma, \varphi, 2)\ldots \tag{4.2}$$

де $sats(\sigma, \varphi, i)$ *- атом, що містить формули* $cl(\varphi)$*,що задовольняють* σ^i.

Інтуїтивно зрозуміло,що задовільна послідовність обчислення σ отримується шляхом "задоволення" σ: тоді як σ лише вказує, які атомарні пропозиції виконуються в кожний момент часу, послідовність що задовільняє також вказує, який атом утримується в кожен момент.

Визначення 4.6. *Послідовність пре-Гінтікки для* φ *- це нескінченна послідовність* $\alpha_0\alpha_1\alpha_2\ldots$ *атомів, що задовольняють наступні умови для кожного* $i \geqslant 0$*:*

(l1) *Для кожного* $\boldsymbol{X}\varphi \in cl(\varphi) : \boldsymbol{X}\varphi \in \alpha_i$ *тільки якщо* $\varphi \in \alpha_{i+1}$

(l2) *Для кожного* $\varphi_1 \boldsymbol{U} \varphi_2 \in cl(\varphi) : \varphi_1 \boldsymbol{U} \varphi_2 \in \alpha_i$ *тільки якщо* $\varphi_2 \in \alpha_i$ *чи* $\varphi_1 \in \alpha_i$ *і* $\varphi_1 \boldsymbol{U} \varphi_2 \in \alpha_{i+1}$.

Послідовність пре-Гінтікки - це послідовність Гінтікки, якщо вона також задовольняє

(g) *Для кожного* $\varphi_1 \boldsymbol{U} \varphi_2 \in \alpha_i$ *існує* $i \geqslant i$ *такі що* $\varphi_2 \in \alpha_j$.

Послідовність пре-Гінтікки або Гінтікки α *відповідає обчисленням* σ*, якщо* $\sigma \subseteq \alpha_i$ *для кожного* $i \geqslant 0$.

З визначення послідовності Гінтікки безпосередньо випливає, що якщо $\alpha = \alpha_0\alpha_1\alpha_2\ldots$ є задовільною послідовністю, тоді кожна пара α_i, α_{i+1} задовольняє (l1) і (l2), а сама послідовність α задовольняє (g). Отже, кожна задовільна послідовність - це послідовність Гінтікки. Наступна теорема показує, що має місце і зворотне: кожна послідовність Гінтікки є задовільною послідовністю.

Теорема 4.7. *Нехай σ - обчислення, а φ - формула. Унікальною послідовністю Гінтікки для φ, що відповідає σ, є послідовність задоволення $sats(\sigma, \varphi)$.*

4.3.2. Побудова NGA для ЛТЛ формули. Отримавши формулу φ, ми побудуємо узагальнений автомат Бюхі A_φ, що розпізнає $L(\varphi)$. За визначенням задовільної послідовності, обчислення задовольняє φ тоді і лише тоді, коли $\varphi \in sats(\sigma, \varphi, 0)$. Більше того, за теоремою 4.7 $sats(\sigma, \varphi)$ є (унікальною) послідовністю Гінтікки для φ, що відповідає σ. Отже, A_φ повинен визнати обчислення σ задовільними: перший атом унікальної послідовності Гінтікки для φ, що відповідає σ, містить φ.

Для цього ми застосовуємо таку стратегію:

(a) Визначимо стани та переходи автомата так, щоб проходи A_φ були усіма послідовностями $\alpha_0 \to^{\sigma_0} \alpha_1 \to^{\sigma_1} \alpha_2 \to^{\sigma_2} \ldots$ такі, що $\sigma = \sigma_0\sigma_1\ldots$ є обчисленням, а $\alpha = \alpha_0\alpha_1\ldots$ є послідовністю пре-Гінтікки φ, що відповідає σ.

(b) Визначимо набори станів прийняття автомата (нагадаємо, що a_φ є NGA), щоб прохід приймав тільки тоді, коли відповідна йому послідовність до Гінтікки також є послідовністю Гінтікки.

Умова (a) визначає алфавіт, стани, переходи та початковий стан A_φ:

— Алфавіт з A_φ буде 2^{AP}.

— Стани з A_φ будуть атомами з φ.

— Початкові стани це атоми з α такі, що $\varphi \in \alpha$.

— Вихідної таблицею переходів стану α (де α - атом) є трійки $\alpha \to^\sigma \beta$ такі, що σ відповідає α, а пара α, β задовольняє умовам (l1) та (l2) (де α і β грають роль α_i відповідно α_{i+1}).

Набори станів прийняття A_φ визначаються умовою (b). За визначенням послідовності Гінтікки, ми повинні гарантувати, що кожен шлях $\alpha_0 \to^{\sigma_0} \alpha_1 \to^{\sigma_1} \alpha_2 \to^{\sigma_2} \ldots$ при будь-якому α_i містить підформулу $\varphi_1 \mathbf{U} \varphi_2$ тоді є $j \geqslant i$ такі що $\varphi_2 \in \alpha_j$. За умовою (l2) це гарантує, що кожен шлях містить нескінченно багато індексів i таких, що $\varphi_2 \in \alpha_i$, або нескінченно багато індексів j таких, що $\neq (\varphi_1 \mathbf{U} \varphi_2) \in \alpha_j$. Отже, ми обираємо набори станів прийняття наступним чином:

— Умова прийняття містить набір $F_{\varphi_1 \mathbf{U} \varphi_2}$ станів прийняття для кожної підформули $\varphi_1 \mathbf{U} \varphi_2$ з φ. Атом належить $F_{\varphi_1 \mathbf{U} \varphi_2}$, якщо він не містить $\varphi_1 \mathbf{U} \varphi_2$ або якщо він містить φ_2.

LTLtoNGA(φ)
Input: formula φ of AP
Output: NGA $A_\varphi = (Q, 2^{AP}, Q_0, \delta, \mathcal{F})$ with $L\left(A_\varphi\right) = L(\varphi)$

```
1  Q_0 ← {α ∈ at(φ) | φ ∈ α}; Q ← ∅;δ ← ∅
2  W ← Q_0
3  while W ≠ ∅ do
4     pick α from W
5     add α to Q
6     for all φ_1 U φ_2 ∈ cl(φ) do
7        if φ_1 U φ_2 ∉ α or  φ_2 ∈ α then add α to F_{φ_1 U φ_2}
8     for all β ∈ at(φ) do
9        if α,β satisfies (l1) and (l2) then
10          add (α, α ∩ AP, β) to δ
11          if β ∉ Q then add β to W
12 F ← ∅
13 for all φ_1 U φ_2 ∈ cl(φ) do F ← F ∪ {F_{φ_1 U φ_2}}
14 return (Q, 2^{AP}, Q_0, δ, F)
```

Рис. 4.1: Псевдокод алгоритму побудови NGA для ЛТЛ формули

NGA, отримані з KNK формул за допомогою LTLtoNGA, мають дуже особливу структуру:

— Як зазначалося вище, всі переходи, що виходять із стану, мають однакову позначку.

— Кожне обчислення, прийняте NGA, має один єдиний шлях прийняття.

Згідно з визначенням NGA, якщо $\alpha_0 \to^{\sigma_0} \alpha_1 \to^{\sigma_1} \dots$ є шляхом прийняття, то $\alpha_0\alpha_1\alpha 2\dots$ є задовільною послідовністю $\sigma_0\sigma_1\sigma 2\dots$. Оскільки задовільна послідовність даного обчислення за визначенням є унікальною, тут може бути лише цикл прийняття.

— Набори обчислень зроблені з будь-яких двох різних станів NGA будуть непересічними. Нехай σ - обчислення, і нехай $sats(\sigma, \varphi) = sats(\sigma, \varphi, 0)sats(\sigma, \varphi, 1)\dots$ будуть його задовільною послідовністю. Тоді σ є прийнятним лише зі стану $sats(\sigma, \varphi, 0)$.

4.3.3. Розмір NGA. Нехай n - довжина формули φ. Неважко помітити, що множина $cl(\varphi)$ має розмір $O(n)$. Отже, NGA A_φ має не більше $O(2^n)$ станів. Оскільки φ містить не більше n підформул виду $\varphi_1\mathbf{U}\varphi_2$, то автомат A_φ має не більше n наборів кінцевих станів.

Тепер доведемо відповідність нижньої оцінки кількості станів. Продемонструємо сімейство формул $\{\varphi_n\}_{n\geqslant 1}$ таке, що φ_n має довжину $O(n)$, і кожен NGA, що розпізнає $L_\omega(\varphi_n)$, має принаймні 2^n станів. Покажемо, що сімейство $\{D_n\}_{n\geqslant 1}$ ω-мов над алфавітом $\sum$ таке, що для кожного $n \geqslant 0$:

1. кожен NGA, що розпізнає D_n, має щонайменше 2^n станів; і
2. існує формула $\varphi_n \in LTL(\sum)$ довжиною $O(n)$ така, що $L_\omega(\varphi_n) = D_n$.

Зверніть увагу, що в (2) ми зловживаємо мовою, тому що якщо $\varphi_n \in LTL(\sum)$, то $L_\omega(\varphi_n)$ містить слова над алфавітом $2^\sum$, і тому $L_\omega(\varphi_n)$ і D_n є мовами над різними алфавітами. Під $L_\omega(\varphi_n) = D_n$ ми маємо на увазі, що для кожного обчислення $\sigma \in (2^\sum)^\omega$ маємо $\sigma \in L_\omega(\varphi_n)$ якщо $\sigma = \{a_1\}\{a_2\}\{a_3\}\dots$ для деякого ω-слова $a_1a_2a_3\dots \in D_n$.

Допустимо $\sum = \{0, 1, \sharp\}$ і виберемо мову D_n наступним чином:

$$D_n = \{ww\sharp^\omega | w \in \{0,1\}^n\} \tag{4.3}$$

1. Кожен NGA, що розпізнає D_n, має щонайменше 2^n станів.

   Припустимо, що узагальнений автомат Бюхі $A = (Q, \{0, 1, \sharp\}, \delta, q_0, \{F_1, \dots, F_k$ з $|Q| < 2^n$ розпізнає D_n. Тоді для кожного слова $w \in \{0,1\}^n$ n існує

такий стан q_w, що A приймає $w\sharp^\omega$ від q_w. За принципом Діріхле маємо $q_{w_1} = q_{w_2}$ для двох різних слів $w_1, w_2 \in \{0,1\}^n$. Але тоді A приймає $w_1 w_2 \sharp^\omega$, але не належить D_n, що суперечить гіпотезі.

2. Існує формула $\varphi_n \in LTL(\sum)$ довжиною $O(n)$ така, що $L_\omega(\varphi_n) = D_n$.

   Спочатку будуємо наступні допоміжні формули:

   — $\varphi_{n_1} = \mathbf{G}((0 \vee 1 \vee \sharp) \wedge \neg(0 \vee 1) \wedge \neg(0 \vee \sharp) \wedge \neg(1 \wedge \sharp))$.

   Ця формула виражає, що в кожній позиції виконується рівно одне атомне твердження.

   — $\varphi_{n_3} = \neg\sharp \wedge (\wedge_{i=1}^{2n-1} \mathbf{X}^i \neg\sharp) \wedge \mathbf{X}^{2n}\mathbf{G}\sharp$.

   Ця формула виражає, що $\sharp$ не тримається на жодній з перших $2n$ позицій, і вона тримається на всіх наступних позиціях.

   — $\varphi_{n_3} = \mathbf{G}((0 \to \mathbf{X}^n(0 \vee \sharp)) \wedge (1 \to \mathbf{X}^n(1 \vee \sharp)))$.

   Ця формула виражає, що якщо атомарна пропозиція утримується в позиції 0 або 1, то n позицій пізніше атомарна пропозиція містить те саме, або $\sharp$.

   Очевидно, що $\varphi_n\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ - це формула, яку ми шукаємо. Зверніть увагу, що φ_n містить $O(n)$ символів.

4.4. Верифікація ЛТЛ формул

Тепер ми можемо описати процедуру автоматичної верифікації властивостей, виражених ЛТЛ формулами LTL.

Вхідні дані до процедури наступні:

- системний NBA A_s, отриманий або безпосередньо із системи, або шляхом обчислення асинхронного продукту мережі автоматів;
- ЛТЛ формула φ над набором атомарних пропозицій AP; і
- оцінка $\nu : AP \to 2^C$, де C - набір конфігурацій A_s, що описує для кожного атомарного твердження набір конфігурацій, у яких має місце пропозиція.

Процедура виконується наступними кроками:

1. Обчислення NGA A_ν для заперечення формули φ. A_ν розпізнає всі обчислення, що порушують φ.
2. Обчислення NGA $A_\nu \cap A_s$ визнання виконуваних обчислень системи, що порушують формулу.
3. Перевірка на пустоту $A_\nu \cap A_s$.

Крок (1) можна здійснити, застосувавши LTLtoNGA, а крок (3), скажімо, Two-Stack алгоритм 2.3. Для кроку (2) спочатку зауважте, що алфавіти A_ν і A_s відрізняються: алфавіт A_ν - це 2^{AP}, тоді як алфавіт A_s - це набір C конфігурацій. Застосовуючи оцінку ν, ми перетворюємо A_ν на автомат із алфавітом C. Оскільки всі стани системних NBA прийнятні, автомат $A_\nu ? A_s$ може бути обчислений за допомогою IntersNBA функції 4.2.

ЗАЗНАЧАЮ, цей алгоритм 4.2 подається без детального опису адже його реалізація після розуміння попередніх глав не складе труднощів. Також можна скористатися детальним поясненням із джерела [8].

Важливо зауважити, що три етапи можуть виконуватися одночасно. Стани $A_\nu \cap A_s$ - пари $[\alpha, c]$, де α - атом φ, а c - конфігурація. Наступний алгоритм приймає пару $[\alpha, c]$ на вхід і повертає своїх нащадків в NGA $A_\nu \cap A_s$. Алгоритм спочатку обчислює нащадків c в A_s. Потім для кожного нащадка c' він обчислює спочатку набір P атомарних пропозицій, що задовольняє c' відповідно до оцінки, а потім набір атомів β такий, що $(a)\beta$ відповідає P і (b) пара α, β задовольняє умовам (l1) та (l2). Нащадками $[\alpha, c]$ є пари $[\beta, c']$.

Цей алгоритм можна вставити в алгоритм перевірки пустоти. Наприклад, якщо ми використовуємо Two-stack, то ми просто замінюємо **рядок 6** 2.2

$$6 \text{ for all } r \in \delta(q) \text{ do} \tag{4.4}$$

на виклик з Succ:

$$6 \text{ for all } [\beta, c'] \in Succ([\alpha, c]) \text{ do} \tag{4.5}$$

IntersNBA(A_1, A_2)
Input: NBAs $A_1 = (Q_1, \Sigma, \delta_1, Q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, Q_{02}, F_2)$
Output: NBA $A_1 \cap_\omega A_2 = (Q, \Sigma, \delta, Q_0, F)$ with $L_\omega(A_1 \cap_\omega A_2) = L_\omega(A_1) \cap L_\omega(A_2)$

```
1  Q, δ, F ← ∅
2  q0 ← [q01, q02, 1]
3  W ← { [q01, q02, 1] }
4  while W ≠ ∅ do
5     pick [q1, q2, i] from W
6     add [q1, q2, i] to Q'
7     if q1 ∈ F1 and i = 1 then add [q1, q2, 1] to F'
8     for all a ∈ Σ do
9        for all q1' ∈ δ1(q1, a), q2' ∈ δ(q2, a) do
10          if i = 1 and q1 ∉ F1 then
11             add ([q1, q2, 1], a, [q1', q2', 1]) to δ
12             if [q1', q2', 1] ∉ Q' then add [q1', q2', 1] to W
13          if i = 1 and q1 ∈ F1 then
14             add ([q1, q2, 1], a, [q1', q2', 2]) to δ
15             if [q1', q2', 2] ∉ Q' then add [q1', q2', 2] to W
16          if i = 2 and q2 ∉ F2 then
17             add ([q1, q2, 2], a, [q1', q2', 2]) to δ
18             if [q1', q2', 2] ∉ Q' then add [q1', q2', 2] to W
19          if i = 2 and q2 ∈ F2 then
20             add ([q1, q2, 2], a, [q1', q2', 1]) to δ
21             if [q1', q2', 1] ∉ Q' then add [q1', q2', 1] to W
22 return (Q, Σ, δ, Q0, F)
```

Рис. 4.2: Псевдокод алгоритму побудови перетину двох NBA автоматів.

ВИСНОВКИ

В ході виконання кваліфікаційної роботи було досліджено складність перевірки пустоти мови автоматів Бюхі за допомогою DFS алгоритмів. Розглянута доцільність використання цієї роботи в алгоритмах верифікації автоматів.

Впродовж виконання кваліфікаційної роботи було проведено аналіз існуючих алгоритмів перевірки пустоти, а саме алгоритми Nested та Two-stack. Досліджено автоматичний підхід до верифікації моделей систем заданих LTL-формулами.

Розроблено кросплатформний консольний продукт на основі технологій C++ з використанням тільки стандартної бібліотеки шаблонів (STL), призначений для перевірки мови автоматів на пустоту. Врахована можливість використання узагальнених автоматів та їх перетворення для подальшого використання. Запропоновано алгоритм генерації довільних автоматів A.3. Виконано порівняння ефективності алгоритмів на довільних вхідних даних.

З повним представленням реалізованого додатку і його вихідним кодом можна ознайомитися на веб-сервісі для програмного забезпечення за наступним посиланням: https://github.com/maxs-im/DFA_algorithms.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] Javier Esparza: Automata theory. An algorithmic approach. September 24, 2019: Classes of ω-Automata and Conversions 235-237.

[2] Jean-Michel Couvreur, Alexandre Duret-Lutz, Denis Poitrenaud: On-the-Fly Empti- ness Checks for Generalized Büchi Automata. SPIN 2005: 169-184.

[3] Jean-Michel Couvreur: On-the-Fly Verification of Linear Temporal Logic. World Congress on Formal Methods 1999: 253-271.

[4] H. Tauriainen. Automated testing of Büchi automata translators for linear temporal logic. Technical Report A66, Laboratory for Theoretical Computer Science, Helsinki University of Technology, 2000. Available at.

[5] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In Proceedings of the 1st IEEE Symposium on Logic in Computer Science (LICS'86), сторінки 332–344. IEEE Computer Society Press, 1986.

[6] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. Formal Methods in SystemDesign, 1:275–288, 1992.

[7] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99), volume I, volume 1708 of Lecture Notes in Computer Science, сторінки 253–271. Springer-Verlag, 1999.

[8] Javier Esparza: Automata theory. An algorithmic approach. September 24, 2019: Emptiness check: Implementations 267-289.

[9] Javier Esparza: Automata theory. An algorithmic approach. September 24, 2019: Boolean operations: Implementations 252-254.

[10] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic

verification of linear temporal logic. In Proceedings of 15th Workshop Protocol Specification, Testing, and Verification, сторінки 3–18. Chapman & Hall, 1995.

[11] T. Latvala and K. Heljanko. Coping with strong fairness. Fundamenta Informaticae, 43(1–4):175–193,2000.

[12] Кривий С.Л. Скінченні автомати: теорія, алгоритми, складність. "Букрек"2020: 277-278

Додаток А

Реалізація представлення та генерації автоматів Бюхі

Лістинг А.1: Оголошення класу представлення автомату Бюхі

```
#pragma once

#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <optional>

#include <ostream>

namespace automates
{

/// \class (Nondeterministic) BuM€chi automaton
class buchi
{
public:
    buchi() = delete;
    buchi& operator=(const buchi&) = delete;

    /// \typedef Automation size limitation
    using atm_size = uint32_t;

    /// \brief an initial or start state: q0 ϵ€€ Q
```

```
/// \note Use NULL as default for all automates
static constexpr atm_size INITIAL_STATE = 0;

/// \typedef Container definition of the final states
    struct
using finals_container = std::vector<std::unordered_set<
    atm_size>>;
/// \typedef Container definition of the transition
    table struct
using table_container = std::unordered_map<atm_size, std
    ::unordered_set<atm_size>>;
/// \typedef Container definition of the final set
    indexes
using indexes_set = std::vector<atm_size>;

/// \brief Creates an BuM€chi object with simple input
    verification (by asserting)
/// \param finals: a correct set of sets of final states
/// \param trans_table: a correct transition table (map<
    state, set<next_state>>)
explicit buchi(finals_container finals, table_container
    trans_table) noexcept;

/// \brief Get number of final state
/// \return size of @m_final_states
[[nodiscard]] atm_size get_final_num_sets() const
    noexcept { return m_final_states.size(); }

/// \brief Check if this is Generalized BuM€chi
```

```
    automaton
/// \return true if is NGA (more than one set of
    acceptable states)
[[nodiscard]] bool is_generalized() const noexcept {
    return get_final_num_sets() > 1; }

/// \brief Check if input number is an accept/final
    state
/// \note May be a misunderstanding for NGA
/// \param state: automaton state
/// \param set_num: specify final set number. By default
     check in all
/// \return whether state belongs to @m_final_states
[[nodiscard]] bool is_final(atm_size state, std::
    optional<atm_size> set_num = std::nullopt) const
    noexcept;

/// \brief Denotes the set of all indices i ∈ K such
    that state ∈ Fi
/// \note Sorted ascending and unique
/// \param state: automaton state
/// \return Sorted ascending and unique vector of
    @m_final_states indexes that contain according state
[[nodiscard]] indexes_set indexes_final_sets(atm_size
    state) const noexcept;

/// \brief Simple and user-friendly BuMChi automaton
    representation
///      "-NUM->" - means NUM symbol for acceptable
```

```
        transition
    ///      \file representation.cpp
    /// \param out: output stream
    /// \param automaton: automaton that will be printed
    /// \return output stream
    friend std::ostream& operator<<(std::ostream& out, const
        buchi& automaton);

    /// \brief container for a transition function: Oґ: Q x
       Q в†’ вЄ‘
    const table_container m_trans_table = {};
protected:
    /// \brief container for a set of accept states: f = {F0
       ,..Fm}, Fi вЉ† Q
    const finals_container m_final_states = {};

};

} // namespace automates
```

Лістинг А.2: Визначення класу представлення автомату Бюхі

```
#include "automates/buchi.hpp"

#include <cassert>

namespace automates
{

buchi::buchi(finals_container finals, table_container
   trans_table) noexcept
```

```
    : m_final_states(std::move(finals)), m_trans_table(std::
      move(trans_table))
{
    assert(!m_final_states.empty() && "Empty finals");
    for (const auto& set : m_final_states)
        assert(!set.empty() && "Empty final set");

    assert(!m_trans_table.empty() && "Empty transition table
      ");
    for (const auto& [curr_st, set] : m_trans_table)
        assert(!set.empty() && "Empty transition map");
}

bool buchi::is_final(const atm_size state, const std::
  optional<atm_size> set_num) const noexcept
{
    if (set_num)
    {
        // return false on too big final set index
        return *set_num < m_final_states.size() ?
            m_final_states[*set_num].find(state) !=
              m_final_states[*set_num].end() :
            false;
    }

    for (const auto& set : m_final_states)
        if (set.find(state) != set.end())
            return true;
```

```
    return false;
}

buchi::indexes_set buchi::indexes_final_sets(const atm_size
   state) const noexcept
{
    indexes_set res;
    res.reserve(m_final_states.size());

    for (atm_size i = 0; i < m_final_states.size(); ++i)
        if (m_final_states[i].find(state) != m_final_states[
           i].end())
            res.push_back(i);

    res.shrink_to_fit();
    return std::move(res);
}

} // namespace automates
```

Лістинг А.3: Визначення методу генерації довільного автомату

```
#include "utils/generator.hpp"

#include <random>
#include <cmath>
#include <algorithm>

/// \namespace Anonymous namespace. Helpers with generating
   automaton parts
namespace
```

```
{

/// \brief Generate oriented one-connected tree
/// \param states_num: number of vertex in tree
/// \param alphabet: number of different edges
/// \param edges: number of edges for each_vertex
/// \param is_initial: should we left root equal to 0 (
    initial automaton state) and connect it with all states
/// \return randomized transition table, but consisting of
    the one tree
automates::buchi::table_container generate_tree(const
    automates::buchi::atm_size states_num,
                                                    const
                                                        automates
                                                        ::buchi::
                                                        atm_size
                                                        edges,
                                                    const bool
                                                        is_initial
                                                        = false)
                                                        noexcept
{
    // check on emptiness
    if (!states_num || !edges)
        return {};

    std::random_device dev;
    std::mt19937 rng(dev());
```

```
// store states for visit. Where index - is a number of
   the turn and value is a state
std::vector<automates::buchi::atm_size> stotage(
   states_num);
// increasing fill elements from 0 to size()
std::iota(stotage.begin(), stotage.end(), 0);
// randomize state selection. Left 0 without changes (
   correct start point)
std::shuffle(stotage.begin() + static_cast<automates::
   buchi::atm_size>(is_initial), stotage.end(), rng);

automates::buchi::table_container tree;
// fully connected for initial state. Otherwise, tree
   could be smaller then max
const automates::buchi::atm_size max_turns = is_initial
   ? stotage.size() :
                              std::uniform_int_distribution
                                 <automates::buchi::atm_size
                                 >(1, stotage.size())(rng);
// usage of a state: visited or not. Index of a @storage
for (automates::buchi::atm_size turn = 0; turn <
   max_turns; ++turn)
{
    // generate edges for current state
    for (automates::buchi::atm_size i = 0; i <= edges &&
        turn < stotage.size() - i; ++i)
    {
        /// \note: symbol may be overwritten during
           further tree merge
```

```
                tree[stotage[turn]].insert(stotage[turn + i]);
            }
        }

        return std::move(tree);
}

/// \brief Generate final state sets
/// \param states_num: number of states in automaton
/// \param set_num: number of sets in the final container
/// \param ratio: special experimental value to control
    maximum elements in one set. Use number of edges for each
    vertex
/// \return randomized final states container
automates::buchi::finals_container generate_finals(const
    automates::buchi::atm_size states_num,
                                                    const
                                                        automates
                                                        ::buchi
                                                        ::
                                                        atm_size

                                                        set_num
                                                        ,
                                                    const
                                                        automates
                                                        ::buchi
                                                        ::
                                                        atm_size
```

```
                                                    ratio)

                                                    noexcept

{
    // check on emptiness
    if (!states_num || !set_num || !ratio)
        return {};

    std::random_device dev;
    std::mt19937 rng(dev());

    // Some manual decrease for maximum final states number
       per set
    std::uniform_int_distribution<automates::buchi::atm_size
      > dist_set(1,
            std::max(static_cast<double>(states_num) /
              set_num / ratio, 1.0));
    // Distribution for final states
    std::uniform_int_distribution<automates::buchi::atm_size
      > dist(0, states_num - 1);
    automates::buchi::finals_container finals(set_num);

    for (auto& set : finals)
    {
        auto max_in_set = dist_set(rng);
        set.reserve(max_in_set);
        for (automates::buchi::atm_size i = 0; i <
           max_in_set; ++i)
```

```
            set.insert(dist(rng));
        }

        return std::move(finals);
    }

} // namespace anonymous

automates::buchi utils::generator::generate_automaton(const
    generator_opts& opts) noexcept
{
    automates::buchi::table_container transitions;
    // trees generating and merging
    for (automates::buchi::atm_size tree = 0; tree < opts.
        trees; ++tree)
    {
        auto gt = generate_tree(opts.states, opts.edges,
            tree == 0);
        // merging trees
        for (auto& [from, set] : gt)
            for (auto& to : set)
                transitions[from].insert(to);
    }

    return automates::buchi(generate_finals(opts.states,
        opts.sets, opts.edges), std::move(transitions));
}
```

Додаток Б

Реалізація перетворення

Лістинг Б.1: Реалізація перетворення NGA в NBA

```
#include "utils/converters.hpp"

#include <queue>

/// \brief hash_combine you could hash an entire (ordered)
   container, for example, as long as each member
/// is individually hashable
/// \note Boost realization
template <class T>
inline void hash_combine(std::size_t & seed, const T & v)
{
    std::hash<T> hasher;
    seed ^= hasher(v) + 0x9e3779b9 + (seed << 6u) + (seed >>
        2u);
}
/// \struct helper for using pairs in unordered structs
template<typename S, typename T>
struct std::hash<std::pair<S, T>>
{
    std::size_t operator()(const std::pair<S, T> &pair)
       const
    {
        size_t seed = 0;
```

```
            ::hash_combine(seed, pair.first);
            ::hash_combine(seed, pair.second);

            return seed;
        }
};

namespace utils::converters
{

using namespace automates;

/// \namespace Anonymous namespace. Helpers with paired
    states
namespace
{

/// \typedef helper for conversion
/// \note need to be the same with std::hash
using pr = std::pair<buchi::atm_size, buchi::atm_size>;

/// \brief Translate unique paired states into the numbered
    and constructs new NBA
/// \param Q: all paired states
/// \param Q0: initial paired state
/// \param F: final states (one set) in paired states
/// \param delta: transition table in paired states
/// \return new NBA automaton (not generalized)
buchi decode_paired_states(const std::unordered_set<pr> &Q,
```

```
    const pr &Q0,
                                const std::unordered_set<pr> &F,
                                const std::unordered_map<pr, std
                                    ::unordered_set<pr>> &delta)
{
    std::unordered_map<pr, buchi::atm_size> dict;
    dict.reserve(Q.size());

    buchi::atm_size state_num = 0;
    for (const auto& it : Q)
        // reserve default initial state value correctly for
            the new automaton
        if (it == Q0)
            dict[it] = automates::buchi::INITIAL_STATE;
        else
            dict[it] = state_num == automates::buchi::
              INITIAL_STATE ? ++state_num : state_num++;

    std::unordered_set<buchi::atm_size> new_F;
    new_F.reserve(F.size());
    for (const auto& itF : F)
        new_F.insert(dict[itF]);

    buchi::table_container new_delta;
    new_delta.reserve(delta.size());
    for (const auto& [old_st, set] : delta)
        for (const auto& new_st : set)
            new_delta[dict[old_st]].insert(dict[new_st]);
```

```
    return buchi({std::move(new_F)}, std::move(new_delta));
}

} // namespace anonymous

std::optional<buchi> nga2nba(const buchi& automat) noexcept
{
    if (!automat.is_generalized())
        return std::nullopt;

    // state and final state new containers
    std::unordered_set<pr> Q, F;
    // container for the new transition table
    std::unordered_map<pr, std::unordered_set<pr>> delta;
    // new initial point
    pr Q0 {automates::buchi::INITIAL_STATE, 0};

    // queue tracker for new generated state
    std::queue<pr> W;
    W.push(Q0);
    while (!W.empty())
    {
        // current tracked new state
        auto pair = W.front();
        auto& [q, i] = pair;
        W.pop();

        Q.insert(pair);
        if (automat.is_final(q, 0) && i == 0)
```

```
            F.insert(pair);

        if (auto iter = automat.m_trans_table.find(q); iter
          != automat.m_trans_table.end())
        {
            for (const auto& qt : iter->second)
            {
                // generate either new or current states
                   copy
                pr new_pair {qt, automat.is_final(q, i) ? (i
                    + 1) % automat.get_final_num_sets() : i};
                if (Q.find(new_pair) == Q.end())
                    W.push(new_pair);

                // accepted state for current tracked state
                delta[pair].insert(new_pair);
            }
        }
    }

    return decode_paired_states(Q, Q0, F, delta);
}

} // namespace utils::converters
```

Додаток В

Реалізація алгоритмів перевірки мови на пустоту

Лістинг В.1: Реалізація Nested алгоритму

```
#include "dfs/nested.hpp"

#include <bitset>
#include <cassert>

namespace emptiness_check::dfs::nested
{

/// \typedef to storing DFS visiting info: <state, <first
    entrance bit, final state mark entrance>>
using um = std::unordered_map<automates::buchi::atm_size,
    std::bitset<2>>;
/// \typedef to storing states of the path
using us = std::unordered_set<automates::buchi::atm_size>;

/// \namespace Anonymous namespace. Helpers with DFS steps
namespace
{

/// \brief Check if q is reachable from itself. Will notify
    NONEMPTY
/// \param q: the state in which we are now
/// \param[in,out] S: DFS state visiting info
```

```
/// \param[in,out] P: current story of the state of the path
/// \param automat: investigated automat
/// \return true if we have to continue investigation
bool dfs2(const automates::buchi::atm_size q, um& S, const
    us& P, const automates::buchi &automat) noexcept
{
    S[q].set(1);

    if (auto iter = automat.m_trans_table.find(q); iter !=
        automat.m_trans_table.end())
    {
        for (const auto &r : iter->second)
        {
            if (P.find(r) != P.end())
                return false; // NONEMPTY NBA
            if (const auto &it_bits = S.find(r);
                    it_bits == S.end() || !it_bits->second.
                        test(1))
            {
                if (!dfs2(r, S, P, automat))
                    return false;
            }
        }
    }

    return true;
}

/// \brief Blackens an accepting state q. Handle @dfs2
```

```
    notification
/// \param q: the state in which we are now
/// \param[in,out] S: DFS state visiting info
/// \param[in,out] P: current story of the state of the path
/// \param automat: investigated automat
/// \return true if we have to continue investigation
bool dfs1(const automates::buchi::atm_size q, um& S, us& P,
    const automates::buchi &automat) noexcept
{
    S[q].set(0);
    P.insert(q);

    if (auto iter = automat.m_trans_table.find(q); iter !=
        automat.m_trans_table.end())
    {
        for (const auto &r : iter->second)
            if (const auto &it_bits = S.find(r);
                    it_bits == S.end() || !it_bits->second.
                        test(0))
            {
                if (!dfs1(r, S, P, automat))
                    return false;
            }
    }
    /// \note: better to add 0 due to NBA
    if (automat.is_final(q))
        if (!dfs2(q, S, P, automat))
            return false;
```

```
    P.erase(q);

    return true;
}

} // namespace anonymous

bool is_empty(const automates::buchi &automat) noexcept
{
    assert(!automat.is_generalized() && "NGA unsupported");

    um S; us P;
    return dfs1(automates::buchi::INITIAL_STATE, S, P,
        automat);
}

} // namespace emptiness_check::dfs::nested
```

Лістинг B.2: Реалізація Two-stack алгоритму

```
#include "dfs/two_stack.hpp"

#include <stack>
#include <algorithm>

#include <iostream>

namespace emptiness_check::dfs::two_stack
{

/// \typedef to storing DFS visiting info: <state <V
```

```
    entrance, discovery time>>
using um = std::unordered_map<automates::buchi::atm_size,
    std::pair<bool, automates::buchi::atm_size>>;
/// \typedef to storing set of candidates: <state, final
    indexes set for the state>
using si = std::stack<std::pair<automates::buchi::atm_size,
    automates::buchi::indexes_set>>;

/// \namespace Anonymous namespace. Helpers with DFS steps
namespace
{

/// \brief DFS search with improvements. Will notify
    NONEMPTY
/// \param q: the state in which we are now
/// \param[in,out] S: DFS state visiting info: <state, <bit
    whether state in V, state discovery time>>
/// \param[in,out] C: set of candidates, containing the
    states for which it is not yet known whether they belong
    to some
/// cycle that denotes the set of all indices i ∈ K such
    that q ∈ Fi for NGA
/// \param[in,out] V: when a state is discovered (greyed),
    it is pushed into the stack (so states are always ordered
/// in V by increasing discovery time); and when a root is
    blackened, all states of V above it (including the root
/// itself) are popped. Note: V ⊆ S holds at all times
/// \param[in,out] t: timestamps for the states
/// \param automat: investigated automat
```

```
/// \return true if we have to continue investigation
bool dfs(const automates::buchi::atm_size q, um &S, si &C,
   std::stack<automates::buchi::atm_size> &V,
         automates::buchi::atm_size& t, const automates::
            buchi &automat) noexcept
{
    const bool is_nga = automat.is_generalized();
    // To not calculate without a reason indexes set
    C.push({ q, is_nga ? automat.indexes_final_sets(q) :
       automates::buchi::indexes_set{} });
    V.push(q);
    S[q] = { true, ++t };

    if (auto iter = automat.m_trans_table.find(q); iter !=
       automat.m_trans_table.end())
    {
        for (const auto &r : iter->second)
            if (const auto &it_bits = S.find(r); it_bits ==
               S.end())
            {
                if (!dfs(r, S, C, V, t, automat))
                    return false;
            }
            else if (it_bits->second.first)
            {
                automates::buchi::indexes_set I{};
                automates::buchi::atm_size s;
                do {
                    const auto& [s_top, J] = C.top();
```

```
                    s = s_top;
                    if (is_nga)
                    {
                        automates::buchi::indexes_set new_I
                            {};
                        std::set_union(I.begin(), I.end(), J
                            .begin(), J.end(), std::
                            back_inserter(new_I));
                        I = new_I;
                        if (I.size() == automat.
                            get_final_num_sets())
                            return false; // NONEMPTY NGA
                    }
                    else
                    {
                        /// \note: may add 0 due to the NBA
                        if (automat.is_final(s))
                            return false; // NONEMPTY NBA
                    }

                    C.pop();
                } while (S[s].second > S[r].second); //
                    lifetime comparing
                C.push({ s, I });
            }
        }

        if (const auto& [c_q, _] = C.top(); c_q == q)
        {
```

```
            C.pop();
            automates::buchi::atm_size s;
            do {
                s = V.top();
                V.pop();
                S[s].first = false;
            } while (s != q);
        }

        ++t;
        return true;
    }

} // namespace anonymous

bool is_empty(const automates::buchi &automat) noexcept
{
    um S;
    si C;
    std::stack<automates::buchi::atm_size> V;
    automates::buchi::atm_size t = 0;

    return dfs(automates::buchi::INITIAL_STATE, S, C, V, t,
        automat);
}

} // namespace emptiness_check::dfs::two_stack
```