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:

ПОБУДОВА АВТОМАТА БЮХІ ТА АЛГОРИТМИ ПЕРЕВІРКИ ПУСТОТИ МОВИ, ЯКА АКЦЕПТУЄТЬСЯ АВТОМАТОМ, НА ОСНОВІ BFS АЛГОРИТМУ

Виконав студент 2-го курсу магістратури
Михайленко Максим Олександрович ________
(підпис)

Науковий керівник:
професор, доктор фіз.-мат. наук
Кривий Сергій Лук'янович ________
(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
О. І. Провотар ________
(підпис)

Київ – 2021

РЕФЕРАТ

Кваліфікаційна робота на здобуття ступеня магістра складається із вступу, чотирьох розділів, висновку, списку використаних джерел (25 найменувань) та трьох додатків.

Робота містить 4 рисунки. Загальний обсяг роботи становить 66 сторінок, основний текст роботи викладено на 42 сторінках.

Об’єктом дослідження є верифікація моделі за заданими специфікаціями

Мета дослідження – розробити програмне забезпечення побудови автомата Бюхі за ЛТЛ формулами та перевірити пустоти мови, яку акцептує автомат, що є прямим добутком моделі Кріпке та автомата побудованого за ЛТЛ формулою.

Для розв’язання поставлених завдань використовуються алгоритми пошуку в ширину на графі автомата. Програма, що реалізує алгоритми, написана на мові C++.

Ключові слова: ПЕРЕВІРКА МОДЕЛІ, ЛІНІЙНА ТЕМПОРАЛЬНА ЛОГІКА, ПОСЛІДОВНІСТЬ ГІНТІККИ, АВТОМАТ БЮХІ, АЛГОРИТМ, ПОШУК В ШИРИНУ, АЛГОРИТМ ЕМЕРСОНА-ЛЕЯ, МОДИФІКОВАНИЙ АЛГОРИТМ ЕМЕРСОНА-ЛЕЯ

ЗМІСТ

ВСТУП 4

1 ПЕРЕВІРКА МОДЕЛІ 6

1.1 Загальні відомості 6

1.2 Характеристика перевірки моделі 10

1.2.1 Процес перевірки моделі 10

1.2.2 Сильні та слабкі сторони 15

2 ЛІНІЙНА ТЕМПОРАЛЬНА ЛОГІКА 18

2.1 Загальні відомості 18

2.2 Від ЛТЛ формул до узагальнених автоматів Бюхі 20

2.2.1 Виконувана послідовність та послідовність Гінтікки 20

2.2.2 Побудова NGA за формулою ЛТЛ 24

2.2.3 Розмір NGA 29

3 ПЕРЕВІРКА МОВИ НА ПУСТОТУ АЛГОРИТМАМИ НА ОСНОВІ ПОШУКУ В ШИРИНУ 32

3.1 Загальні відомості 33

3.2 Алгоритм Емерсона-Лея 35

3.3 Модифікований алгоритм Емерсона-Лея 39

3.4 Порівняння алгоритмів 41

3.4.1 Хороший випадок для MEL. 41

3.4.2 Хороший випадок для алгоритму Емерсона-Лея 42

4 ВЕРИФІКАЦІЯ ЛТЛ ФОРМУЛ 43

ВИСНОВКИ 45

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 46

ДОДАТОК А 49

ДОДАТОК Б 52

ДОДАТОК В 56

ДОДАТОК Г 60

ВСТУП

Оцінка сучасного стану об'єкта розробки. В інформатиці перевірка моделі являє собою метод, за допомогою якого встановлюється виконуваність на моделі системи заданих специфікацій. Зазвичай це пов'язано з апаратними або програмними системами Специфікації можуть містити вимоги до живучості (наприклад, уникнення livelock), а також вимоги безпеки (наприклад, уникнення станів, що призводять до аварії системи).

Для алгоритмічного розв'язання такої задачі необхідно побудувати модель системи і її специфікацію сформульовану в якійсь логічній математичній мові. В даній роботі використовуються мова лінійної темпоральної логіки і ω-автомати Бюхі. Задача перевірки на моделі полягає у перевірці виконуваності заданих специфікацій.

Актуальність роботи та підстави для її виконання. Складність програмного забезпечення росте з кожним роком і чим довше розробляється застосунок і більше завдань він вирішує, тим виникає більша кількість помилок. Через ці помилки компанія може нести не тільки фінансові втрати, а й може поставити під загрозу людське життя. Звідси випливає необхідність у формальних методах перевірки системи.

Мета й завдання роботи. Метою кваліфікаційної роботи є створення програмного застосунку для побудови автомата Бюхі за заданими формулами і його верифікація . Для досягнення цієї мети поставлено такі завдання.

- Дослідити застосування алгоритмів перевірки виконуваності специфікації на моделі.
- Реалізувати алгоритм побудови автомата Бюхі за заданими ЛТЛ формулами.

- Реалізувати та порівняти алгоритми перевірки пустоти мови, яка акцептується прямим добутком автоматів моделі Кріпке і відповідного автомата Бюхі.

Об'єкт, методи й засоби розроблення. Об'єктом розробки програмного засобу є програмна реалізація процесу верифікації побудованої моделі відносно заданих специфікацій та встановлення наявності помилок і шляхів їх виникнення.

Під час розробки програмного продукту використана каскадна модель, заснована на таких принципах. Спочатку збираються вимоги до системи та програмного забезпечення. Потім йде створення моделей та дизайн архітектури застосунку. Після цих кроків починається етап написання коду та подальше його тестування.

В якості інструменту створення програмного засобу було обрано CLion – інтегроване середовище розробки (IDE) мовою програмування C++. Було обрано саме цю мову програмування через її швидкість та мультиплатформенність, а CLion дозволяє зручно шукати по проекту та керувати версіями.

Можливі сфери застосування. Програмний продукт для побудови автоматів Бюхі за ЛТЛ формулами та перевірки пустоти мови яку акцептує результуючий автомат використовується для перевірки відповідності моделі заданій специфікації. Такий метод одержав назву метод перевірки на моделі (model checking (MC)).

1 ПЕРЕВІРКА МОДЕЛІ

1.1 Загальні відомості

У програмному та апаратному проектуванні складних систем більше часу і сил витрачається на перевірку, ніж на будівництво. Методи прагнуть зменшити та полегшити зусилля з перевірки, збільшивши їх охоплення. Формальні методи пропонують великий потенціал для отримання ранньої інтеграції перевірки в процесі проектування, забезпечення більш ефективних методів верифікації та скорочення часу перевірки[1].

Давайте спочатку коротко обговоримо роль формальних методів. Коротше кажучи, формальні методи можна розглядати як "прикладну математику для моделювання та аналізу ІКТ-систем". Їх мета - встановити правильність системи з математичною суворістю. Їх великий потенціал призвів до збільшення використання інженерами формальних методів верифікації складних програмних та апаратних систем. Крім того, формальні методи є однією з "настійно рекомендованих" методів верифікації програмного забезпечення для розробки критичних систем безпеки відповідно до, наприклад, стандарту найкращих практик IEC (Міжнародної електротехнічної комісії) та стандартам ESA (Європейського космічного агентства)[2]. У результаті звіту розслідування ФАА (Федерального управління авіації) та НАСА (Національної аеронавтики та космічної адміністрації) щодо використання формальних методів робиться висновок, що:

Формальні методи повинні бути частиною освіти кожного комп'ютерного вченого та інженера-програмного забезпечення, так як відповідна галузь прикладної математики є необхідною частиною освіти всіх інших інженерів[3].

Протягом останніх двох десятиліть дослідження формальних методів призвели до розробки деяких дуже перспективних методик перевірки, які сприяють ранньому виявленню дефектів. Ці методи супроводжуються потужними програмними засобами, які можна використовувати для автоматизації різних етапів перевірки. Дослідження показали, що формальні процедури перевірки виявили б дефекти, наприклад, ракети "Аріан-5", "Марса Патфіндера", процесора Intel Pentium II та терапевтичного апарату терапії "Терак-25"[1].

Методи перевірки на основі моделей базуються на моделях, що описують можливу поведінку системи математично точно та однозначно. Виявляється, що - перед будь-якою формою перевірки - точне моделювання систем часто призводить до виявлення неповноти, неоднозначності та невідповідності неформальних специфікацій системи. Такі проблеми зазвичай виявляються лише на значно пізнішому етапі проектування. Моделі системи супроводжуються алгоритмами, які систематично досліджують усі стани системної моделі. Це дає основу для цілого спектру методів верифікації, починаючи від вичерпного дослідження (перевірка моделі) до експериментів із обмежувальним набором сценаріїв в моделі (моделювання) або насправді (тестування).

Завдяки невпинним вдосконаленням алгоритмів та структур даних, а також наявності більш швидких комп'ютерів та більшої пам'яті комп'ютера, на сьогоднішній день застосовні для реальної конструкції методи на основі моделей, які десятиліття тому працювали лише для дуже простих прикладів.

Означення 1. Перевірка моделі - це техніка верифікації, яка досліджує всі можливі стани системи на грубій основі[3].

Подібно до комп'ютерної шахової програми, яка перевіряє можливі рухи, програма перевірки моделей, програмний інструмент, який виконує перевірку моделі, систематично вивчає всі можливі системні сценарії. Таким чином можна показати, що в даній моделі системи справді виконується певна властивість. Справжній виклик - дослідити найбільші можливі простори стану, які можна обробити поточними засобами, тобто процесорами та пам'яттю. Сучасні моделі можуть обробляти приблизно від 10^8 до 10^9 станів. Використовуючи розумні алгоритми та спеціалізовані структури даних, для конкретних проблем можна обробити більшу кількість станів (від 10^{20} до навіть 10^{476} станів)[4]. Навіть тонкі помилки, які залишаються нерозкритими за допомогою емуляції, тестування та моделювання, потенційно можуть бути виявлені за допомогою перевірки моделі.

Типові властивості, які можна перевірити за допомогою перевірки моделі, мають якісний характер: Чи добре генерований результат? Чи може система досягти тупикової ситуації, наприклад, коли дві одночасні програми чекають одна одну і тим самим зупиняють всю систему? Але також можна перевірити властивості синхронізації: Чи може статися тупиковий стан протягом 1 години після скидання системи?, або Чи завжди відповідь отримана протягом 8 хвилин? Перевірка моделі вимагає точного та однозначного твердження властивостей, що підлягають дослідженню. Як і у

випадку складання точної системної моделі, цей крок часто призводить до виявлення кількох неоднозначностей та невідповідностей у неофіційній документації. Наприклад, формалізація всіх системних властивостей для частини ISDN протоколу користувача показала, що 55% (!) неофіційних системних вимог були несумісними[5].

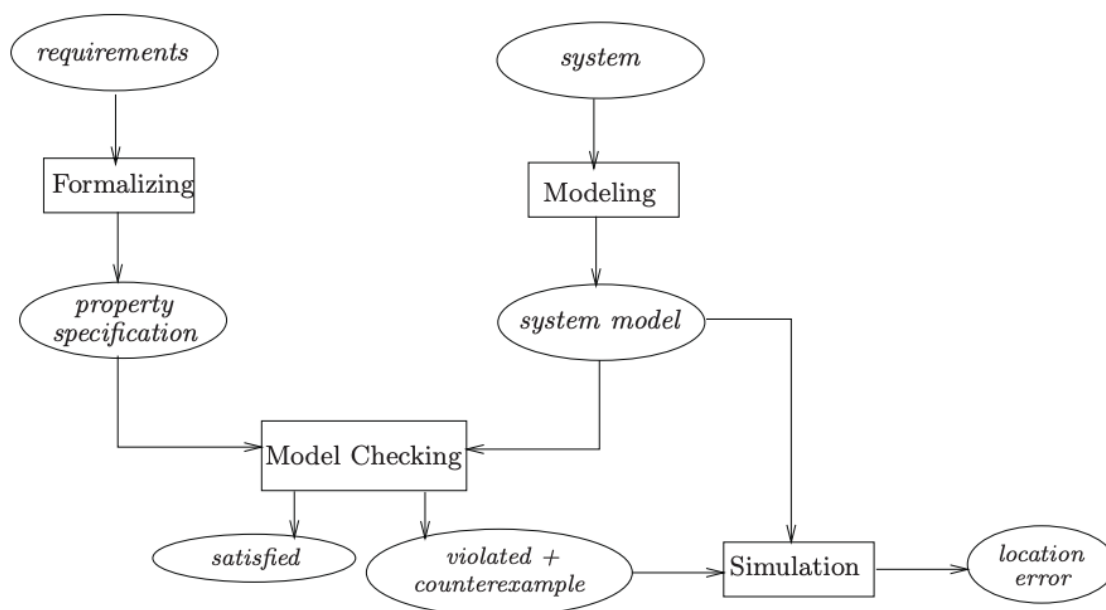

Рисунок. 1.1 - Схематичний вигляд підходу перевірки моделі.

Модель системи, як правило, автоматично генерується з опису моделі, яка вказана на якомусь відповідному діалекті мов програмування, таких як C або Java, або мов опису апаратних засобів, таких як Verilog або VHDL. Специфікація властивості визначає, що повинна робити система, а що вона не повинна робити, тоді як опис моделі стосується поведінки системи. Засіб перевірки моделей вивчає всі відповідні стани системи, щоб перевірити, чи виконується бажана властивість. Якщо виникає стан, який порушує розглянуту властивість, перевіряючи модель надає контрприклад, який

вказує, як модель могла досягти небажаного стану. За допомогою симулятора користувач може відтворити сценарій, що порушує, таким чином отримуючи корисну інформацію про налагодження і відповідно адаптувати модель (або властивість) (див. Рисунок. 1.1)[4].

1.2 Характеристика перевірки моделі

1.2.1 Процес перевірки моделі

При застосуванні методу перевірки на моделі виділяються наступні фази[5]:

- Фаза моделювання:
  - моделювати реальну систему, використовуючи автоматні методи побудови такої моделі;
  - в якості першої перевірки обгрунтованості та швидкої оцінки моделі будують моделі високого рівня;
  - формалізувати властивості, що підлягають перевірці, використовуючи мову специфікацій.
- Фаза побудови результуючого автомата: побудувати прямий добуток автомата реальної моделі і автомата Бюхі, що побудований за ЛТЛ специфікацією
- Фаза запуску: запустити перевірку моделі, щоб перевірити коректність властивості в системній моделі.
- Етап аналізу:
  - властивість виконується? → перевірити наступну властивість (якщо вона є);
  - властивість порушено? →

- проаналізувати генерований контрприклад за допомогою моделювання;
- вдосконалити модель, дизайн або властивість;
- повторити всю процедуру.

- недостатньо пам'яті? → спробуйте зменшити модель і повторіть спробу.

Крім цих кроків, всю перевірку слід планувати, адмініструвати та організовувати. Це називається організацією перевірки. Ці фази перевірки моделі будуть описані дещо детальніше нижче.

Моделювання. Необхідними умовами для перевірки моделі є модель розглянутої системи та формальна характеристика властивості, що перевіряється[6].

Моделі систем точно і однозначно описують поведінку систем. Вони здебільшого виражаються за допомогою скінченних автоматів, що складаються з кінцевого набору станів і набору переходів. Стани містять інформацію про поточні значення змінних, раніше виконаний оператор (наприклад, лічильник програми) тощо. Переходи описують, як система еволюціонує з одного стану в інший. Для реалістичних систем скінченні автомати описуються за допомогою мови опису моделі, наприклад відповідного діалекту/розширення C, Java, VHDL тощо. Системи моделювання, зокрема паралельні, на правильному рівні абстрагування досить складні і справді є мистецтвом;

Для покращення якості моделі може бути проведено моделювання до перевірки моделі. Моделювання можна ефективно використовувати для позбавлення від простішої категорії помилок моделювання. Усунення цих

простіших помилок до того, як відбудеться будь-яка форма ретельного перевірки, може зменшити дорогі та трудомісткі зусилля з перевірки.

Щоб зробити можливою строгу перевірку, властивості повинні бути описані точно й однозначно. Зазвичай це робиться за допомогою мови специфікації властивості. Зокрема, ми зосередимо увагу на використанні часової логіки як мови специфікації властивостей, форми модальної логіки, яка доречна для визначення відповідних властивостей систем ІКТ. З точки зору математичної логіки можна перевірити, чи опис системи є моделлю темпоральної логічної формули. Це пояснює термін "перевірка моделі"[2].

Означення 2. Темпоральна логіка - це в основному розширення традиційної логіки пропозицій з операторами, які посилаються на поведінку систем у часі.

Це дозволяє визначити широкий спектр відповідних властивостей системи, таких як функціональна коректність (чи система робить те, що належить зробити?), доступність (чи можна опинитися в тупиковому стані?), безпека ("щось поганого ніколи не буває »), живучості (« зрештою станеться щось добре »), справедливості (чи за певних умов подія трапляється неодноразово?) та властивостей у режимі реального часу (чи діє система в часі?)[6].

Хоча вищезазначені кроки часто добре зрозумілі, на практиці може бути серйозною проблемою судити про те, чи формалізована заява проблеми (модель + властивості) є адекватним описом фактичної проблеми верифікації. Це також відоме як проблема перевірки. Складність залученої системи, а також недостатня точність неофіційного конкретизації функціональності системи можуть ускладнити відповідь на це питання. Перевірку та підтвердження не слід плутати. Підтвердження полягає в тому, щоб

перевірити, чи відповідає проект визначеним вимогам, тобто перевірка - це "перевірити, чи ми будуємо річ правильно". У валідації перевіряється, чи відповідає формальна модель неформальній концепції дизайну, тобто перевірка - це "перевірити, чи ми перевіряємо правильність".

Запуск перевірки моделі Спочатку перевірку моделі потрібно ініціалізувати, встановивши різні параметри та директиви, які можуть бути використані для проведення вичерпної перевірки. Згодом відбувається фактична перевірка моделі. Це в основному виключно алгоритмічний підхід, при якому обгрунтованість властивості, що розглядається, перевіряється у всіх станах системної моделі[6].

Аналіз результатів В основному є три можливі результати: вказана властивість чи дійсна у даній моделі, чи ні, або модель виявляється занадто великою, щоб вміститись у фізичні межі пам'яті комп'ютера.

Якщо властивість є дійсною, можна перевірити наступне властивість, або, якщо всі властивості перевірені, модель укладається, що вона має всі бажані властивості.

Щоразу, коли властивість порушується, негативний результат може мати різні причини. Може виникнути помилка моделювання, тобто, вивчивши помилку, виявляється, що модель не відображає конструкцію системи. Це означає корекцію моделі, і перевірку потрібно перезапустити з покращеною моделлю. Це повторне підтвердження включає перевірку тих властивостей, які раніше були перевірені на помилковій моделі та перевірка яких може бути визнана недійсною виправленням моделі! Якщо аналіз помилок показує, що між дизайном та його моделлю немає необґрунтованої невідповідності, то або

була виявлена помилка проектування, або помилка властивості. У разі помилки проектування перевірка завершується негативним результатом, і дизайн (разом із його моделлю) має бути вдосконалений. Може статися так, що при вивченні виявленої помилки виявляється, що властивість не відображає неофіційну вимогу, яку потрібно було перевірити. Це передбачає модифікацію властивості, і потрібно провести нову перевірку моделі. Оскільки модель не змінюється, не повинно відбуватися повторне підтвердження властивостей, які були перевірені раніше. Дизайн перевіряється, якщо і тільки якщо всі властивості були перевірені щодо дійсної моделі.

Кожен раз, коли модель занадто велика, щоб її обробляти - кількість станів у системі в реальному житті можуть бути на багато порядків більшими, ніж те, що можна зберігати за наявними на даний момент пам'яттю - існують різні способи з ними працювати. Можливість полягає в застосуванні прийомів, які намагаються використовувати неявні закономірності в структурі моделі. Прикладами цих прийомів є представлення станів за допомогою символічних прийомів, таких як двійкові діаграми рішення або часткове скорочення порядку. Як варіант, використовуються жорсткі абстракції повної моделі системи. Ці абстракції повинні зберігати (не) валідність властивостей, які потрібно перевірити. Часто можна отримати абстракції, які є достатньо малими щодо однієї властивості. У такому випадку для моделі, яку потрібно використовувати, необхідно зробити різні абстракції. Інший спосіб поводження зі станами, які занадто великі, - це відмовитися від точності результату верифікації. Імовірнісні підходи до верифікації досліджують лише частину станів, роблячи (часто мізерно) жертву в покритті верифікації[3].

Організація верифікації Весь процес перевірки моделі повинен бути добре організованим, добре структурованим та добре сплanованим. Промислові програми перевірки моделей надали докази того, що використання керування версіями та конфігурацією має особливе значення. Наприклад, під час верифікації виробляються різні описи моделей, що описують різні частини системи, доступні різні версії моделей верифікації (наприклад, через абстракцію) та велика кількість параметрів верифікації (наприклад, параметри перевірки моделі) та результати (діагностичні дані, статистика) доступні. Цю інформацію потрібно дуже ретельно задокументувати та зберігати, щоб керувати практичним процесом перевірки моделі та дозволяти відтворювати проведені експерименти[6].

1.2.2 Сильні та слабкі сторони

Сильні сторони перевірки моделі:

- Це загальний підхід до верифікації, який має дуже широкий спектр застосувань, наприклад вбудовані системи, інженерія програмного забезпечення та розробка обладнання.
- Підтримує часткову перевірку, тобто властивості можна перевірити окремо, таким чином дозволяючи спершу зосередитись на суттєвих властивостях. Повна специфікація вимог не потрібна.
- Не є вразливим до ймовірності викриття помилки; це контрастує з тестуванням та моделюванням, які спрямовані на відстеження найбільш ймовірних дефектів.

- Надає діагностичну інформацію в разі визнання властивості недійсним; це дуже корисно для налагодження.
- Потенційна технологія "кнопка"; використання перевірки моделі не вимагає ані високого ступеня взаємодії з користувачем, ані високого рівня знань.
- Користується швидко зростаючим інтересом промисловості; кілька апаратних компаній розпочали роботу з власними лабораторіями перевірки, часто з'являються пропозиції щодо роботи з необхідними навичками перевірки моделей, а також з'являються комерційні шашки.
- Можна легко інтегрувати в існуючі цикли розвитку; його крива навчання не дуже крута, а емпіричні дослідження показують, що це може призвести до скорочення часу розвитку.
- Має обгрунтовану та математичну основу; вона заснована на теорії графічних алгоритмів, структур даних та логіки.

Слабкі сторони перевірки моделі:

- Доцільно для інтенсивного контролю програм і менш підходить для подальших програм, оскільки дані, як правило, знаходяться в нескінченних доменах.
- Застосовність підлягає вирішенню проблем; для систем з нескінченним станом або міркувань про абстрактні типи даних (для яких потрібна логіка, що не визначена або напіврозв'язна), перевірка моделі, як правило, не піддається обчисленню.
- Перевіряє модель системи, а не саму систему (продукт чи прототип); будь-який отриманий результат таким чином хороший, як і системна

модель. Додаткові методи, такі як тестування, потрібні для пошуку помилок виготовлення (для апаратних засобів) або помилок кодування (для програмного забезпечення).

- Перевіряє лише заявлені вимоги, тобто немає гарантії повноти. Дійсність властивостей, які не перевіряються, неможливо оцінити.
- Страждає від проблеми вибуху в просторі стану, тобто кількість станів, необхідних для точного моделювання системи, може легко перевищувати об'єм доступної комп'ютерної пам'яті. Незважаючи на розробку декількох дуже ефективних методів боротьби з цією проблемою, моделі реалістичних систем все ще можуть бути занадто великими для неї в пам'яті.
- Використання вимагає певного досвіду у пошуку відповідних абстракцій для отримання менших системних моделей та визначення властивостей використовуваного логічного формалізму.
- Не гарантує правильних результатів: як і будь-який інструмент, перевірка моделі може містити дефекти програмного забезпечення.
- Не дозволяє перевіряти узагальнення: загалом перевірка систем з довільною кількістю компонентів або параметризованих систем не може бути оброблена. Проте перевірка моделі може запропонувати результати для довільних параметрів, які можуть бути перевірені за допомогою допоміжних помічників.

2 ЛІНІЙНА ТЕМПОРАЛЬНА ЛОГІКА

У цьому розділі буде представлено нову мову для визначення властивостей безпеки і живучості, який називається лінійна темпоральна логіка (ЛТЛ). ЛТЛ близька до природної мови, але все ж має формальну семантику.

2.1 Загальні відомості

Формули ЛТЛ побудовані з множини AP атомарних пропозицій. Інтуїтивно атомарні пропозиції – це абстрактні назви основних властивостей конфігурацій, значення яких фіксується тільки після розгляду конкретної системи. Формально, для даної системи з набором конфігурацій C значення атомарних пропозицій фіксується оціночної функцією $V: AP \rightarrow 2^{C}$, яка присвоює кожному абстрактному імені набір конфігурацій, в яких воно виконується. Позначимо ЛТЛ (AP) – безліч формул ЛТЛ над AP.

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

Означення 3. Нехай AP - скінченна множина атомарних пропозиційних формул. ЛТЛ(AP) - це набір виразів, породжених граматикою

$$\varphi := true \mid p \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid X\varphi_1 \mid \varphi_1\ U\ \varphi_2$$

Формули інтерпретуються на послідовностях $\sigma = \sigma_0\sigma_1\sigma_2...$, де $\sigma_i \subseteq AP$ для кожного $i \geq 0$. Ці послідовності називаються обчисленнями. Сукупність усіх обчислень за AP позначається C(AP). Виконавчими обчисленнями системи є обчислення σ, для яких існує ω-виконання $c_0c_1c_2...$ таке, що для кожного $i \geq 0$ множина атомарних пропозицій, де виконується c_i, рівна σ_i. Тепер ми формально визначимо, коли формула виконується[7].

Означення 4. Дано обчислення $\sigma \in C(AP)$, нехай σ^j позначає суфікс $\sigma_j\sigma_{j+1}\sigma_{j+2}...$ з σ. Співвідношення виконуваності $\sigma \vDash \varphi$(читати "φ виконується в σ") індуктивно визначається таким чином:

- $\sigma \vDash true$
- $\sigma \vDash p$ якщо $p \in \varphi(0)$
- $\sigma \vDash \neg\varphi$ якщо $\sigma \nvDash \varphi$
- $\sigma \vDash \varphi_1 \wedge \varphi_2$ якщо $\sigma \vDash \varphi_1$ та $\sigma \vDash \varphi_2$
- $\sigma \vDash X\varphi$ якщо $\sigma^1 \vDash \varphi$
- $\sigma \vDash \varphi_1\, U\, \varphi_2$ якщо існує $k \geq 0$ такий, що $\sigma^k \vDash \varphi_2$ та $\sigma^i \vDash \varphi_1$для кожного $0 \leq i \leq k$

Ми використовуємо такі скорочення:

- false, ∨, → та ↔ інтерпретуються звичайним способом
- $F\varphi = true\, U\varphi$ ("зрештою φ"). Відповідно до семантики вище, $\sigma \vDash F\varphi$ якщо існує $k \geq 0$ такий, що $\sigma^k \vDash \varphi$
- $G\varphi \vDash \neg F\neg\varphi$ ("завжди φ"). Відповідно до семантики вище, $\sigma \vDash G\varphi$, якщо $\sigma^k \vDash \varphi$ для кожного $k \geq 0$

Множина обчислень, при якій виконується формула φ, позначається L(φ). Формула φ виконується в системі, якщо φ виконується для всіх її виконуваних обчислень.[7,8]

2.2 Від ЛТЛ формул до узагальнених автоматів Бюхі

Представляємо алгоритм, який, враховуючи формулу $\varphi \in ЛТЛ(AP)$, повертає $NGA\ A_{\varphi}$ над алфавітом 2^{AP}, що розпізнає L(φ), а потім виводимо повністю автоматичну процедуру, яка, враховуючи систему та формулу ЛТЛ, вирішує, чи виконується формула для виконуваних обчислень системи.

2.2.1 Виконувана послідовність та послідовність Гінтікки

Визначимо виконувану послідовність та послідовність Гінтікки для обчислення σ та формули φ. Спочатку потрібно ввести поняття замикання формули та атом замикання.

Означення 5. Для формули φ, запереченням φ буде формула ψ, якщо $\varphi = \neg\psi$і формула ¬φ в іншому випадку. Замикання cl(φ) формули φ - це множина, що містить усі підформули φ та їх заперечення. Непуста множина $\alpha \subseteq cl\ (\varphi)$ є атомом cl(φ), якщо виконуються наступні властивості:

a0. Якщо $true\ \in cl(\varphi)$, тоді $true\ \in\ \alpha$

a1. Для кожного $\varphi_1\ \wedge\ \varphi_2\ \in cl(\varphi)$: $\varphi_1\ \wedge\ \varphi_2\ \in\ \alpha$ тоді і тільки тоді, коли $\varphi_1\ \in\ \alpha$ та $\varphi\ \in\ \alpha$

a2. Для кожного $\neg\varphi_1 \in cl(\varphi)$: $\neg\varphi_1 \in \alpha$ тоді і тільки тоді, коли $\varphi_1 \notin \alpha$

Сукупність усіх атомів cl(φ) позначається через at(φ).

Варто звернути увагу, що якщо α - набір усіх формул cl(φ), які виконуються для обчислення σ, то α обов'язково є атомом. Справді, **true** виконується для кожного обчислення; якщо сполучення двох формул виконується для обчислення, то кожен із сполучників також виконується; тому, якщо формула виконується для обчислення, то її заперечення не виконується, і навпаки. Також слід зауважити, що через (a2), якщо cl(φ) містить k формул, кожен атом cl(φ) містить рівно k/2 формул[9].

Означення 6. Виконувана послідовність для обчислення σ і формули φ є нескінченною послідовністю атомів

$$sats(\sigma, \varphi) = sats(\sigma, \varphi, 0)\, sats(\sigma, \varphi, 1)\, sats(\sigma, \varphi, 2). . .$$

де sats (σ, φ, i) - атом, що містить формули cl(φ), які виконуються в σ^i .

Інтуїтивно зрозуміло, що виконувана послідовність обчислення σ отримується шляхом "завершення" σ: хоча σ лише вказує, які атомарні пропозиції виконуються в кожний момент часу, виконувана послідовність також вказує, який атом утримується в кожен момент.[9,10]

Означення 7. Послідовність до-Гінтікки для φ - це нескінченна послідовність $\alpha_1, \alpha_2, \alpha_3$... атомів, що виконуються при наступних умовах для кожного i ≥ 0:

l1. Для кожного $X\varphi \in cl(\varphi)$: $X\varphi \in \alpha_i$ тоді і тільки тоді, коли $\varphi \in \alpha_{i+1}$

l2. Для кожного $\varphi_1 U \varphi_2 \in cl(\varphi)$: $\varphi_1 U \varphi_2 \in \alpha_i$ тоді і тільки тоді, коли $\varphi_2 \in \alpha_i$ або $\varphi_2 \in \alpha_i$ та $\varphi_1 U \varphi_2 \in \alpha_{i+1}$

Послідовність до-Гінтікки - це послідовність Гінтікки, якщо вона також виконується:

g. Для кожного $\varphi_1 U \varphi_2 \in \alpha_i$ існує такий j ≥ i, що $\varphi_2 \in \alpha_j$

Послідовність до-Гінтікки або Гінтікки α відповідає обчисленням σ, якщо $\sigma_i \subseteq \alpha_i$ для кожного i ≥ 0.

Слід звернути увагу, що умови (l1) та (l2) є локальними: для того, щоб визначити, чи виконується α, нам потрібно лише перевірити кожну пару α_i, α_{i+1} послідовних атомів. Навпаки, умова (g) є глобальною, оскільки відстань між індексами i та j може бути довільно великою.

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

Теорема 1. Нехай σ - обчислення, а φ - формула. Єдиною послідовністю Гінтікки для φ відповідає σ є виконувана послідовність sats (σ, φ)[11].

Доведення:

Як зазначено вище, з означено одразу випливає, що $sats(\sigma, \varphi)$ є послідовністю Гінтікки для φ, що відповідає σ. Щоб показати, що жодна інша послідовність Гінтікки не відповідає sats (σ, φ), нехай $\alpha = \alpha_0\alpha_1\alpha_2...$ є послідовністю Гінтікки для φ відповідає σ і нехай ψ - це довільна формула

cl(φ). Доведемо, що для кожного і ≥ 0: $\psi \in \alpha_i$ тоді і лише тоді, коли $\psi \in sats(\sigma, \varphi, i)$.

Доведення за індукцією

- ψ = true. Тоді $true \in sats(\sigma, \varphi, i)$ і, оскільки α_i є атомом, $true \in \alpha$
- ψ = p для атомарної пропозиції p. Оскільки α відповідає σ, ми маємо $p \in \alpha_i$ тоді і лише тоді, коли $p \in \sigma_i$. За визначенням виконуваної послідовності, $p \in \sigma_i$ тоді і лише тоді, коли $p \in sats(\sigma, \varphi, i)$. Тож $p \in \alpha_i$ тоді і лише тоді, коли $p \in sats(\sigma, \varphi, i)$.
- $\psi = \varphi_1 \wedge \varphi_2$. Маємо:

  $\varphi_1 \wedge \varphi_2 \in \alpha_i$

  $\Leftrightarrow \varphi_1 \in \alpha_i$ та $\varphi_2 \in \alpha_i$ (умова (a1))

  $\Leftrightarrow \varphi_1 \in sats(\sigma, \varphi, i)$ та $\varphi_2 \in sats(\sigma, \varphi, i)$ (гіпотеза індукції)

  $\Leftrightarrow \varphi_1 \wedge \varphi_2 \in sats(\sigma, \varphi, i)$ (визначення sats (σ, φ))

- $\psi = \neg\varphi_1$ або $\psi = X\varphi_1$. Доведення дуже схоже на попереднє.
- $\psi = \varphi_1 U \varphi_2$. Доведемо:

  a. Якщо $\varphi_1 U \varphi_2 \in \alpha_i$, то $\varphi_1 U \varphi_2 \in sats(\sigma, \varphi, i)$.

     За умовою (l2) визначення послідовності Гінтікки ми повинні розглянути два випадки:

     – $\varphi_2 \in \alpha_i$. За математичною індукцією, $\varphi_2 \in sats(\sigma, \varphi, i)$, а отже, $\varphi_1 U \varphi_2 \in sats(\sigma, \varphi, i)$.

     – $\varphi_1 \in \alpha_i$ та $\varphi_1 U \varphi_2 \in \alpha_{i+1}$. За умовою (g) існує принаймні один індекс j ≥ i такий, що $\varphi_2 \in \alpha_j$. Нехай j_m є найменшим

із цих індексів. Результат ми доведемо за індукцією по $j_m - i$. Якщо $i = j_m$, то $\varphi_2 \in \alpha_j$ і ми будемо діяти, як у випадку $\varphi_2 \in \alpha_i$. Якщо $i < j_m$, то оскільки $\varphi_1 \in \alpha_i$, маємо $\varphi_1 \in sats(\sigma, \varphi, i)$ (індукція по ψ). Оскільки $\varphi_1 U \varphi_2 \in \alpha_{i+1}$, ми маємо або $\varphi_2 \in \alpha_{i+1}$, або $\varphi_1 \in \alpha_{i+1}$. У першому випадку ми маємо $\varphi_2 \in sats(\sigma, \varphi, i + 1)$, і тому $\varphi_1 U \varphi_2 \in sats(\sigma, \varphi, i)$. У другому випадку, за індукцією (індукція по $j_m - i$), маємо $\varphi_1 U \varphi_2 \in sats(\sigma, \varphi, i + 1)$, а отже $\varphi_1 U \varphi_2 \in sats(\sigma, \varphi, i)$.

b. Якщо $\varphi_1 U \varphi_2 \in sats(\sigma, \varphi, i)$, то $\varphi_1 U \varphi_2 \in \alpha_i$.

Ми знову розглянемо два випадки

- $\varphi_2 \in sats(\sigma, \varphi, i)$. За математичною індукцією $\varphi_2 \in \alpha_i$, а отже $\varphi_1 U \varphi_2 \in \alpha_i$.
- $\varphi_1 \in sats(\sigma, \varphi, i)$ та $\varphi_1 U \varphi_2 \in sats(\sigma, \varphi, i + 1)$. Згідно з визначенням виконуваної послідовності, існує принаймні один індекс $j \geq i$ такий, що $\varphi_2 \in sats(\sigma, \varphi, j)$. Продовжити, як у випадку (a)

2.2.2 Побудова NGA за формулою ЛТЛ

Дано формула φ ми будуємо узагальнений автомат Бюхі A_φ, що розпізнає $L(\varphi)$. За визначенням виконуваної послідовності, формула φ

виконується в σ тоді і лише тоді, коли $\varphi \in sats(\sigma, \varphi, 0)$[12]. Більше того, за теоремою 1 $sats(\sigma, \varphi)$ є (унікальною) послідовністю Гінтікки для φ відповідає σ. Тож A_φ повинен розпізнавати обчислення σ, в якому виконується: перший атом унікальної послідовності Гінтікки для φвідповідає σ містить φ.

Для цього застосуємо таку стратегію:

a. Визначимо стани та переходи автомата так, щоб проходи по A_φ були послідовностями

$$\alpha_0 \xrightarrow{\sigma_0} \alpha_1 \xrightarrow{\sigma_1} \alpha_2 \xrightarrow{\sigma_2} \ldots$$

такими, що $\sigma = \sigma_0\sigma_1 \ldots$ є обчисленням, а $\alpha = \alpha_0\alpha_1 \ldots$ є послідовністю до-Гінтікки φвідповідає σ.

b. Визначимо набори акцептуючих станів автомата (нагадаємо, що $A_\varphi \in NGA$), так що прохід акцептується тоді і тільки тоді, коли відповідна йому послідовність до-Гінтікки також є послідовністю Гінтікки.

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

- Алфавіт $A_\varphi - 2^{AP}$.
- Стани A_φ є атомами φ
- Початковими станами є атоми α такі, що $\varphi \in \alpha$.
- Вихідними переходами стану α (де α - атом) є трійки $\alpha \xrightarrow{\sigma} \beta$такі, що σ відповідає α, а пара α, β виконується для умов (l1) та (l2) (де α та β відповідаютьаα_i та α_{i+1} відповідно).

Набори акцептуючих станів $A\varphi$ визначаються умовою (b). Визначаючи послідовність Гінтікки, ми повинні гарантувати, що в кожному проході $\alpha_0 \xrightarrow{\sigma_0} \alpha_1 \xrightarrow{\sigma_1} \alpha_2 \xrightarrow{\sigma_2} ....$, якщо будь-який α_i містить підформулу $\varphi_1 U \varphi_2$, то існує $j \geq i$такий, що $\varphi_2 \in \alpha_j$. За умовою (l2) це гарантує, що кожен прохід містить нескінченно багато індексів i таких, що $\varphi_2 \in \alpha_i$, або нескінченно багато індексів j таких, що $\neg (\varphi_1 U \varphi_2) \in \alpha_j$. Отже, обираємо набори станів прийняття наступним чином:

Умова акцептації містить набір $F_{\varphi_1 U \varphi_2}$акцептуючих станів для кожної підформули $\varphi_1 U \varphi_2$з φ. Атом належить $F_{\varphi_1 U \varphi_2}$, якщо він не містить $\varphi_1 U \varphi_2$ або якщо він містить φ_2[13].

LTLtoNGA(φ)

Input: формула φ з AP

Output: NGA $A_\varphi = (Q, 2^{AP}, Q_0, \delta, F)$ з $L(A_\varphi) = L(\varphi)$

1. $Q_0 \leftarrow \{\alpha \in at(\varphi) \mid \varphi \in \alpha\};\ Q \leftarrow \emptyset; \delta \leftarrow \emptyset$
2. $W \leftarrow Q_0$
3. $while\ W \neq \emptyset\ do$
4. $\quad pick\ \alpha\ from\ W$
5. $\quad add\ \alpha\ to\ Q$
6. $\quad for\ all\ \varphi_1 U \varphi_2 \in cl(\varphi)\ do$
7. $\quad\quad if\ \varphi_1 U \varphi_2 \notin \alpha\ or\ \varphi_2 \in \alpha\ then\ add\ \alpha\ to\ F_{\varphi_1 U \varphi_2}$

8. $for\ all\ \beta \in at(\varphi)\ do$

9. $if\ \alpha,\ \beta\ satisfies\ (l1)\ and\ (l2)\ then$

10. $add\ (\alpha,\ \alpha \cap AP,\ \beta)\ to\ \delta$

11. $if\ \beta\ <\ Q\ then\ add\ \beta\ to\ W$

12. $F \leftarrow \emptyset$

13. $for\ all\ \varphi_1\ U\ \varphi_2 \in cl(\varphi)\ do\ F \leftarrow F \cup \{F_{\varphi_1\ U\ \varphi_2}\}$

14. $return\ (Q,\ 2^{AP},\ Q_0,\ \delta,\ F)$

Приклад

Побудуємо автомат A_φ для формули $\varphi\ =\ p\ U\ q$. Замикання $cl\ (\varphi)$ має вісім атомів, що відповідає всім можливим способам вибору між p і ¬p, q і ¬q та $p\ U\ q$ і $\neg\ (p\ U\ q)$. Однак легко бачити, що атоми $\{p,\ q,\ \neg\ (p\ U\ q)\}$, $\{\neg p,\ q,\ \neg\ (p\ U\ q)\}$ та $\{\neg p,\ \neg q,\ p\ U\ q\}$ не мають вихідних переходів, оскільки ці переходи порушили б умову (l2). Отже, ці стани можна видалити, і ми залишимо п’ять атомів, показаних на рисунку 1. Три атоми зліва містять $p\ U\ q$, і тому вони стають початковими станами. Варто звернути увагу, що кожен перехідA_φ, що залишає атом α, позначаєтьсяα $\cap\ AP$. Наприклад, усі переходи, що виходять із стану $\{\neg p,\ q,\ p\ U\ q\}$, позначаються {q}, а всі переходи, що залишають $\{\neg p,\ \neg q,\ \neg\ (p\ U\ q)\}$, позначаються ∅. Більше того, оскільки φмає лише одну підформулу виду $\varphi_1\ U\ \varphi_2$, NGA насправді є NBA, і ми можемо представляти акцептуючі стани як для NBA. Акцептуючими станами $F_{p\ U\ q}$є атоми, які не містять $p\ U\ q$- два атоми праворуч - і атоми, що містять q - крайній лівий атом і атом вгорі.

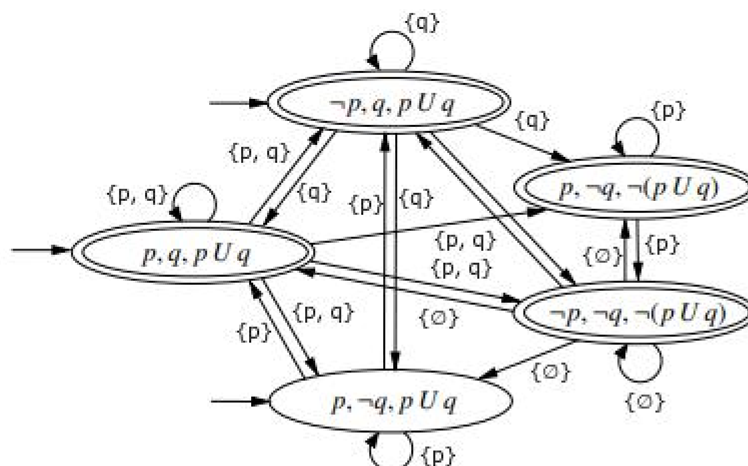

Рисунок 2.1- Автомат побудований за ЛТЛ формулою

Розглянемо, наприклад, атоми $\alpha = \{\neg p,\ \neg q,\ \neg\,(p\ U\ q)\}$та $\beta = \{p,\ \neg q,\ p\ U\ q\}$. A_{φ} містить перехід$\alpha \xrightarrow{\{p\}} \beta$, оскільки {p} відповідає β, і α, β виконуються для умов (l1) та (l2). Умова (l1) виконується вакуумно, оскільки φ не містить підформул виду$X\psi$, тоді як умова (l2) виконується, оскільки $p\ U\ q \notin \alpha$ та$q \notin \beta$і $p \notin \alpha$. З іншого боку, немає переходу від β до α, оскільки це порушило б умову (l2):$p\ U\ q \in \beta$, але ні $q \in \beta$, ні $p\ U\ q \in \alpha$.

NGA, отримані з формул ЛТЛ 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 цикл прийому.

Згідно з визначенням NGA, якщо $\alpha_0 \xrightarrow{\sigma_0} \alpha_1 \xrightarrow{\sigma_1} \cdots$ є акцептуючим проходом, то $\alpha_0 \alpha_1 \alpha_2 \ldots$ - виконувана послідовність $\sigma_0 \sigma_1 \sigma_2 \ldots$. Оскільки виконувана послідовність даного обчислення за визначенням є унікальною, може бути лише один акцептуючий прохід.

- Набори обчислень, розпізнані будь-якими двома різними станами NGA, несумісні. Нехай σ - обчислення, і нехай $sats(\sigma, \varphi) = sats(\sigma, \varphi, 0)\, sats(\sigma, \varphi, 1) \ldots$ буде його виконуваною послідовністю. Тоді σ акцептується лише із стану $sats(\sigma, \varphi, 0)$.

2.2.3 Розмір NGA

Нехай n - довжина формули φ. Легко помітити, що множина $cl(\varphi)$має розмір O(n). Отже, NGA A_φмає не більше $O(2^n)$станів. Оскільки φ містить не більше n підформул виду $\varphi_1 U \varphi_2$, то автомат має нA_φе більше n наборів кінцевих станів[14].

Тепер ми доведемо відповідність нижньої межі кількості станів. Продемонструємо сімейство формул $\{\varphi_n\}\, n\geq 1$таке, що φ_n має довжину O(n), і кожна NGA, що розпізнає $L_\omega(\varphi_n)$, має принаймні 2^nстанів. Для цього продемонструємо сімейство $\{D_n\}\, n\geq 1$ω-мов над алфавітом Σ таке, що для кожного n ≥ 0:

1. кожна NGA, що розпізнає D_n, має щонайменше 2^nстанів

2. існує формула $\varphi_n \in LTL\ (\Sigma)$довжини O (n) така, що$L_\omega\ (\varphi_n) = D_n$.

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

Припустимо $\Sigma = \{0,\ 1,\ \#\}$ і виберемо мову D_n наступним чином:

$$D_n = \{w\,w\,\#^\omega \mid w \in \{0,\ 1\}^n\}$$

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

   Припустимо, що узагальнений автомат Бюхі $A = (Q,\ \{0,\ 1,\ \#\},\ \delta,\ q_0,\ \{F_1,\ ...,\ F_k\})$ з $|Q| < 2^n$розпізнає D_n. Тоді для кожного слова $w \in \{0, 1\}^n$існує такий стан , що A приймає $w\,\#^\omega$з 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 $w1\ w2\ \#^\omega$, що не належить D_n, що суперечить гіпотезі

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

   Спочатку побудуємо наступні допоміжні формули:

   – $\varphi 1 = G\ (0 \vee 1 \vee \#) \wedge \neg\ (0 \wedge 1) \wedge \neg\ (0 \wedge \#) \wedge \neg\ (1 \wedge \#)$.

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

– $\varphi_n 2 = \neg \# \wedge (\wedge_{i=1}^{2n-1} X^i \neg \#) \wedge X^{2n} G \#$

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

– $\varphi_n 3 = G((0 \rightarrow X^n (0 \vee \#)) \wedge (1 \rightarrow X^n (1 \vee \#))$

Ця формула виражає, що якщо атомарна пропозиція утримується в позиції 0 або 1, то n позицій пізніше атомарна пропозиція має те саме, або #

Очевидно, що $\varphi_n = \varphi_n 1 \wedge \varphi_n 2 \wedge \varphi_n 3$- це формула, яку ми шукаємо. Варто звернути увагу, що φ_n містить O (n) символів

3 ПЕРЕВІРКА МОВИ НА ПУСТОТУ АЛГОРИТМАМИ НА ОСНОВІ ПОШУКУ В ШИРИНУ

Оскільки в багатьох додатках нам доводиться мати справу з дуже великими автоматами Бюхі, нас цікавлять алгоритми, які не вимагають знати автомат Бюхі заздалегідь, але перевіряють на пустоту при його конструюванні. Знадобиться кілька графо-теоретичних понять. Якщо $(q,\ r) \in \delta$, то r є наступником a і q є попередником r.

Шлях - це послідовність $q_0, q_1, . . . , q_n$ таких станів, що q_{i+1} є наступником q_i для кожного $i \in \{0,\ ...,\ n\ -\ 1\}$; 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.

Зрозуміло, що A не порожній, якщо він має приймаюче ласо, тобто шлях $q_0 q_1 ... q_{n-1} q_n$ таким, що $q_n = q_i$ для деякого i $\in$ {0, ..., n - 1}, і принаймні один з станів $\{q_i, q_{i+1}, ..., q_{n-1}\}$ акцептуючий. Ласо складається з шляху $q_0 ... q_i$, за яким слідує порожній цикл $q_i q_{i+1} ... q_{n-1} q_i$. Нас цікавить перевірки на пустоту, що на вході має автомат і на виході EMPTY або NONEMPTY, і в останньому випадку повернути приймаюче ласо як признаку непустоти.

Алгоритми на основі BFS можуть бути відповідним чином описані за допомогою операцій та перевірок наборів, що дозволяє нам реалізовувати їх, використовуючи автомати як структури даних. У багатьох випадках виграш, отриманий в результаті використання структури даних, компенсує

квадратичну складність в найгіршому випадку, роблячи алгоритми конкурентоспроможними.

3.1 Загальні відомості

Пошук в ширину (BFS) зберігає набір станів, які були виявлені, але ще не досліджені, які часто називають кордоном. Пошук в ширину з набору початкових станів Q_0 ініціалізує як набір виявлених станів, так і його кордон, а потім проходить по кроках циклу. Під час пошуку вперед, кожен крок досліджує вихідні переходи зі станів на поточному кордоні; нові стани, знайдені під час кроку, додаються до набору виявлених станів, і вони стають наступним кордоном. Зворотний пошук в ширину протікає аналогічно, але досліджує вхідні замість вихідних переходів. Реалізації псевдокоду обох варіантів пошуку в ширину, показані нижче, використовуючи дві змінні S і B для зберігання набору виявлених станів та кордонів відповідно. Припустимо існування оракулу, який по поточному кордону B, повертає $\delta(B) = \bigcup_{q \in B} \delta(q)$ або $\delta^{-1}(B) = \bigcup_{q \in B} \delta^{-1}(q)$ [15, 16].

Обидва варіанти пошуку в ширину обчислюють наступників або попередників стану рівно один раз, тобто, якщо в ході алгоритму оракул викликається двічі з аргументами B_i і B_j відповідно, тоді $B_i \cap B_j = \emptyset$. Щоб довести це у випадку пошуку вперед (зворотний пошук аналогічний), зауважте, що $B \subseteq S$ - інваріант повторюваного циклу, і що значення S ніколи не зменшується. Нехай B_1, S_1, B_2, S_2... послідовність значень змінних B і S прямо перед і-м виконанням рядка 3. Маємо $B_i \subseteq$ S за інваріантом, $S_i \subseteq S_j$

для кожного $j \geq i$ $B_{j+1} \cap S_j = \emptyset$ до рядку 3. Отже, $B_j \cap B_i = \varnothing$ для кожного $j > i$.

ForwardBFS[A](Q_0)

Input:

NBA A = (Q, Σ, δ, Q_0, F)

$Q_0 \subseteq Q$

1. $S, B \leftarrow Q_0$;
2. repeat
3. $B \leftarrow \delta(B) \setminus S$
4. $S \leftarrow S \cup B$
5. until $B = \emptyset$

BackwardBFS[A](Q_0)

Input:

NBA A = (Q, Σ, δ, Q_0, F)

$Q_0 \subseteq Q$

1. $S, B \leftarrow Q_0$;
2. repeat
3. $B \leftarrow \delta^{-1}(B) \setminus S$
4. $S \leftarrow S \cup B$
5. until $B = \emptyset$

Як структури даних для множин S і B можна використовувати хеш-таблицю і чергу, відповідно. Але також можна прийняти множину станів Q автомата A як кінцеву всесвіт і використовувати автомати для мов фіксованої довжини для представлення як S, так і B. Крім того, можна представити $\delta \subseteq Q \times Q$ скінченним перетворювачем T_δ і звести обчислення $\delta(B)$ та $\delta^{-1}(B)$ у рядку 3 для обчислень $Post(B, \delta)$ та $Pre(B, \delta)$ відповідно[15].

3.2 Алгоритм Емерсона-Лея

Стан q автомата A є живим, якщо існує нескінченний шлях який починається з q і відвідує акцептуючі стани нескінченно часто. Зрозуміло, що автомат A не порожній тоді і тільки тоді, коли його початковий стан є живим[17]. Опишемо алгоритм Емерсона-Лея для обчислення набору станів живого стану. Для кожного n ≥ 0 n-живих станів A індуктивно визначають так:

- кожен стан 0-живий;
- стан q є (n + 1) -живим, якщо деякий шлях, що містить принаймні один перехід, веде від q до an прийняття n-живого стану.

Якщо говорити просто, стан є n-живим, якщо, починаючи з нього, можна відвідати акцептуючі стани n-разів. Нехай $L\,[n]$ множина n-живих станів автомата A . Маємо:

Лема. 1

1. $L[n] \supseteq L[n\ +\ 1]$для кожного $n\ \geq\ 0$
2. Послідовність $L\,[0] \supseteq L\,[1] \supseteq L\,[2]. . .$ досягає точки фіксування $L\,[i]$ (тобто є найменший індекс$i \geq 0$, такий, що $L\,[i\ +\ 1]\ =\ L\,[i]$), а $L\,[i]$- набір живих станів[18].

Доведення: Доведемо (1) за індукцією на n. Випадок $n\ =\ 0$ тривіальний. Припустимо, $n\ >\ 0$, і $q \in L\,[n\ +\ 1]$. Існує шлях, що містить щонайменше один перехід, який веде від q до акцептуючого стану $r \in L\,[n]$. За індукційною гіпотезою, $r \in L\,[n\ -\ 1]$, і так $q \in L\,[n]$.

Щоб довести (2), спочатку зауважте, що оскільки Q скінченне, то точка фіксування $L[i]$ існує. Нехай L - множина живих станів. Зрозуміло, що $L \subseteq L[i]$ для кожного $i \geq 0$. Більше того, оскільки $L[i] = L[i + 1]$, кожен стан $L[i]$ має власного нащадка, який акцептує і належить до $L[i]$. Отже $L[i] \subseteq L$. ■

Алгоритм Емерсона-Лея обчислює точку фіксування $L[i]$послідовності $L[0] \supseteq L[1] \supseteq L[2]. . .$.Для обчислення $L[n + 1]$, заданого L [n], можна спостерігати, що стан n + 1 - живий, якщо якийсь непустий шлях веде з нього до приймаючого стану n-живого, і маємо:

$$L[n + 1] = BackwardBFS(Pre(L[n] \cap F))$$

Псевдокод алгоритму наведено нижче ліворуч; змінна L використовується для зберігання елементів послідовності $L[0]$, $L[1]$, $L[2]$,. . ..

EmersonLei(A)

Input: NBA A = $(Q, \Sigma, \delta, Q_0, F)$

Output: EMP if $L_\omega(A) = \emptyset$,

NEMP otherwise

1. $L \leftarrow Q$
2. repeat
3. $OldL \leftarrow L$
4. $L \leftarrow Pre(OldL \cap F)$

EmersonLei2(A)

Input: NBA A = $(Q, \Sigma, \delta, Q_0, F)$

Output: EMP if $L_\omega(A) = \emptyset$,

NEMP otherwise

1. $L \leftarrow Q$
2. repeat
3. $OldL \leftarrow L$
4. $L \leftarrow Pre(OldL \cap F) \setminus OldL$

5. $L \leftarrow BackwardBFS(L)$	5. $L \leftarrow BackwardBFS(L) \cup OldL$
6. until $L = OldL$	6. until $L = OldL$
7. if $q_0 \in L$ then report NEMP	7. if $q_0 \in L$ then report NEMP
8. else report EMP	8. else report EMP

Цикл повторень виконується не більше $|Q| + 1$- разів, оскільки кожна ітерація, але остання, видаляє принаймні один стан з L. Оскільки кожна ітерація займає $O(|Q| + |\delta|)$ часу, алгоритм працює за $O(|Q| \cdot (|Q| + |\delta|))$.

Алгоритм може двічі обчислити попередників стану. Наприклад, якщо $q \in F$і є перехід (q, q), то після виконання рядка 4 стан все ще належить L. Версія праворуч враховує таку можливість і уникає цієї проблеми.

Алгоритм Емерсона-Лея можна легко узагальнити до роботи з NGA(приведено узагальнення лише для першої версії):

GenEmersonLei(A)

Input: NGA A = $(Q, \Sigma, \delta, q_0, \{F_0, \ldots, F_{m-1}\})$

Output: EMP if $L_\omega(A) = \emptyset$,

NEMP otherwise

1. $L \leftarrow Q$
2. repeat
3. $OldL \leftarrow L$
4. for $i = 0$ to $m - 1$
5. $L \leftarrow Pre(OldL \cap F)\backslash OldL$
6. $L \leftarrow BackwardBFS(L) \cup OldL$

7. until $L = OldL$
8. if $q_0 \in L$ then report NEMP
9. else report EMP

GenEmersonLei (A) повертає $NEMP$ якщо автомат A не порожній.

Доведення: Для кожного $k \geq 0$визначимо n-живих станів автомата А таким чином: кожен стан є 0-живим, а q є $(n + 1)$- живим, якщо якийсь шлях, що має принаймні один перехід, веде від q до n -живого стану $F_{(n \bmod m)}$. Нехай $L[n]$ безліч n-живих станів. Виходячи з леми 1 можна легко показати, що $L[(n + 1) \cdot m] \supseteq L[n \cdot m]$ виконується для кожного $n \geq 0$.

Нехай, маємо послідовність $L[0] \supseteq L[m] \supseteq L[2 \cdot m]. . .$ що досягає точки фіксування $L[i \cdot m]$ (тобто є найменший показник $i \geq 0$, такий що $L[(i + 1) \cdot m] = L[i \cdot m]$), і $L[i \cdot m]$ - набір живих станів. Оскільки Q скінченне, то точки фіксування L [i · m] існує. Нехай q - живий стан. Існує шлях, починаючи з q, який відвідує F_j нескінченно часто для кожного $j \in \{0, ..., m - 1\}$. На цьому шляху кожне виникнення стану F_j завжди супроводжується деяким пізнішим виникненням стану $F_{(j+1) \bmod m}$, для кожного $j \in \{0,. . . , m - 1\}$. Отже $q \in L[i \cdot m]$. Тепер покажемо, що кожен стан $L[i \cdot m]$ живий. Для кожного стану $q \in L[(i + 1) \cdot m]$ існує шлях $\pi = \pi_{m-1}\pi_m - 2\pi_0$ такий, що для кожного $j \in \{0, ..., m - 1\}$ відрізок π_j містить щонайменше один перехід і призводить до стану $L[i \cdot m + j] \cap F_j$. Зокрема, π відвідує стани $F_0,. . . , F_{m-1}$, і, оскільки $L[(i + 1) \cdot m] = L[i \cdot m]$, він веде зі стану $L[(i + 1) \cdot m]$ в інший стан

$L[(i + 1) \cdot m]$. Отже, кожен стан $L[(i + 1) \cdot m] = L[i \cdot m]$ живий, що підтверджує твердження.

Оскільки GenEmersonLei(A) обчислює послідовність $L[0] \supseteq L[m] \supseteq L[2 \cdot m]$..., після припинення L містить набір живих станів[15, 19, 20].

3.3 Модифікований алгоритм Емерсона-Лея

Існує багато варіантів алгоритму Емерсона-Лея, які мають однакову складність, але намагаються підвищити ефективність, принаймні в деяких випадках, за допомогою евристики. Представимо тут один із таких варіантів, який називаємо модифікованим алгоритмом Емерсона-Лея (MEL).

Маємо множину станів $S \subseteq Q$, нехай $inf(S)$ позначає стани $q \in S$ такими, що деякий нескінченний шлях, що починається з q, містить лише стани з S. Замість обчислення $Pre(OldL \cap F)$ на кожному кроці ітерації MEL обчислює $Pre(inf(OldL) \cap Fi)$[15].

MEL(A)

Input: NGA A = ($Q, \Sigma, \delta, q_0, \{F_0, \ldots, F_{m-1}\}$)

Output: EMP if $L_\omega(A) = \emptyset$,

NEMP otherwise

1. $L \leftarrow Q$
2. repeat
3. $OldL \leftarrow L$
4. $L \leftarrow inf(OldL)$
5. $L \leftarrow Pre(OldL \cap F)$
6. $L \leftarrow BackwardBFS(L)$
7. until $L = OldL$
8. if $q_0 \in L$ then report NEMP
9. else report EMP

function inf(S)

1. repeat
2. $OldS \leftarrow S$
3. $S \leftarrow S \cap Pre(S)$
4. *until S = OldS*
5. *return S*

Як можна буде побачити, хоча MEL вводить накладні витрати на багаторазові обчислювальні операції, в багатьох випадках це все-таки має сенс, оскільки зменшує кількість кроків циклу.

Для підтвердження правильності ми стверджуємо, що після закінчення L містить множину живих станів. Нагадаємо, що множина живих станів є точкою фіксування $L[i]$ послідовності $L[0] \supseteq L[1] \supseteq L[2]. . ..$ За визначенням живості ми маємо inf $(L[i]) = L[i]$. Визначте тепер

$L'[0] = Q$, а $L'[n + 1] = inf\,(pre^{+}(L'[i] \cap \alpha))$. Зрозуміло, що MEL обчислює послідовність $L'[0] \supseteq L'[1] \supseteq L'[2]. . ..$ Оскільки $L[n] \supseteq L'[n] \supseteq L[i]$ для кожного $n > 0$, то маємо, що $L[i]$ також є точкою послідовності $L'[0] \supseteq L'[1] \supseteq L'[2]. . .$, і так MEL обчислює $L[i]$. Оскільки $inf\,(S)$ можна обчислити за час $O(|Q| + |\delta|)$ для будь-якого набору S, MEL працює за $O(|Q| \cdot (|Q| + |\delta|))$ [15, 21, 22].

3.4 Порівняння алгоритмів

Наведемо два приклади, які показують, що MEL може перевершити алгоритм Емерсона-Лея, але не завжди.

3.4.1 Хороший випадок для MEL

Розглянемо автомат на рисунку 3.1. I-та ітерація алгоритму Емерсона-Лей видаляє q_{n-i+1}. Кількість викликів BackwardBFS дорівнює $(n + 1)$, хоча проста модифікація дозволяє алгоритму зупинитися, якщо $L = \emptyset$ заряджає (n + 1) - операція. З іншого боку, перше операція *inf* в MEL вже встановлює змінну L в порожній набір станів, і тому, при тій же простій модифікації, алгоритм зупиняється після цієї ітерації.

Рисунок 3.1

3.4.2 Хороший випадок для алгоритму Емерсона-Лея

Розглянемо автомат на рисунку 3.2. I-та ітерація алгоритму Емерсона-Лея видаляє $q_{n-i+1,1}$ і $q_{n-i+1,2}$, і алгоритм викликає BackwardBFS (n+1) разів. I-та ітерація алгоритму MEL не видаляє жодного стану в результаті дії inf операції, і стани $q_{n-i+1,1}$ і $q_{n-i+1,2}$ як результат виклику BackwardBFS. Тож у цьому випадку всі операції є зайвими.

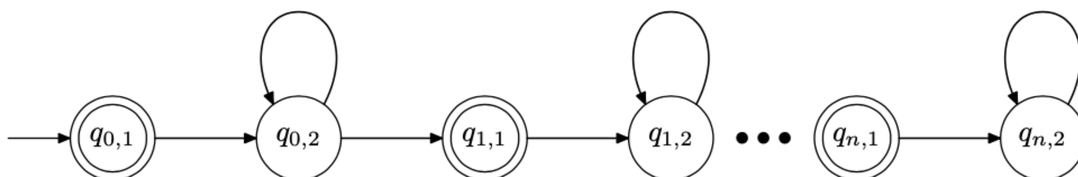

Рисунок 3.2

4 ВЕРИФІКАЦІЯ ЛТЛ ФОРМУЛ

Тепер маємо можливість описати процедуру автоматичної верифікації властивостей, виражених формулами LTL. Вхідні дані до процедури

- система NBA A_s, отримана або безпосередньо від системи, або шляхом обчислення асинхронного добутку автоматів;
- формула φз LTL для набору атомарних пропозицій AP
- оцінка ν: $AP \to 2^C$, де C - набір конфігурацій A_s, що описує для кожної атомарної пропозиції набір конфігурацій, у яких має місце пропозиція[23].

Процедура складається з наступних кроків:

1. Обчислити NGA A_v для заперечення формули φ. A_v розпізнає всі обчислення, що порушують φ.
2. Обчислити NGA $A_v \cap A_s$, розпізнаючи виконувані обчислення системи, які порушують формулу.
3. Перевірити на пустоту $A_v \cap A_s$

Крок (1) можна здійснити, застосувавши LTLtoNGA, а крок (3), скажімо, MEL алгоритм. Для кроку (2) спочатку слід зауважити, що алфавіти A_v і A_s відрізняються: алфавіт A_v - це 2^{AP}, тоді як алфавіт A_s - це набір C конфігурацій. Застосовуючи оцінку ν, ми перетворюємо A_v на автомат із алфавітом C. Оскільки всі стани системи NBA приймаються, автомат $A_v \cap A_s$ може бути обчислений за допомогою interNFA.

Важливо зауважити, що три етапи можуть виконуватися одночасно. Стани $A_v \cap A_s$ - пари [α, c], де α - атом φ, c - конфігурація. Наступний алгоритм приймає пару [α, c] як вхід і повертає своїх наступників в NGA $A_v \cap A_s$. Алгоритм спочатку обчислює наступників c в A_s. Потім для кожного наступника c' він обчислює спочатку набір P атомарних пропозицій, що виконуються в c' згідно з оцінкою, а потім набір атомів β такий, що (a) β відповідає P і (b) пара α, β виконується для умов (l1) та (l2). Наступниками [α, c] є пари [β, c'] [24, 25].

Succ([α, c])

1. $S \leftarrow \emptyset$
2. $for\ all\ c' \in \delta_s(c)\ do$
3. $P \leftarrow \emptyset$
4. $for\ all\ p \in AP\ do$
5. $if\ c' \in \nu(p)\ then\ add\ p\ to\ P$
6. $for\ all\ \beta \in at(\varphi)\ matching\ P\ do$
7. $if\ \alpha,\ \beta\ satisfies\ (l1)\ and\ (l2)\ then\ add\ c\ 0\ to\ S$
8. $return\ S$

ВИСНОВКИ

В ході виконання кваліфікаційної роботи було сформульовано загальні задачі і положення перевірки моделей, їх застосування в сучасній науці і техніці. Описано мову для для визначення властивостей безпеки і живучості, її загальні положення та алгоритм побудови узагальненого автомата Бюхі за ЛТЛ формулою. Введено основні поняття з перевірки моделей і перевірки автоматів на пустоту. Описано основні переваги та недоліки перевірки моделей.

Впродовж виконання кваліфікаційної роботи було проведено аналіз існуючих алгоритмів для перевірки на пустоту, а саме алгоритми Емерсона-Лея та модифікований алгоритм Емерсона-Лея та алгоритми побудови автомата Бюхі за заданими специфікаціями.

Розроблено програмний продукт на основі технології С++, для верифікації моделі. Виконано порівняння ефективності алгоритмів, описано їх переваги і недоліки на різних вхідних даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Clarke E.M., Grumberg Jr. O., Peled D. Model Checking. The MIT Press: England, 2001. 356 p.
2. C. Baier and E. Clarke and V. Hartonas-Garmhausen and M. Kwiatkowska and M. Ryan. Symbolic model checking for probabilistic processes. In 24th International Colloqium on Automata, Languages and Programming (ICALP), volume 1256 of Lecture Notes in Computer Science. Springer-Verlag, 1997.
3. C. Baier and B. R. Haverkort and H. Hermanns and J.-P. Katoen. Model checking algorithms for continuous time Markov chains. IEEE Transactions on Software Engineering, 29(6), 2003
4. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L.Dill, J. Hwang, Symbolic model checking: 10^{20} states and beyond, Information and Computation, 98, 1992.
5. B. Berard and M. Bidoit and A. Finkel and F. Laroussinie and A. Petit and L. Petrucci and Ph. Schnoebelen. Systems and Software Verification: Model-Checking Techniques and Tools. Springer-Verlag, 2001.
6. Christel Baier, Joost-Pieter Katoen. Principles of Model Checking. The MIT Press. 2008. 975p
7. A Pnueli, The Temporal Logic of Programs, Theoretical Computer Science 13, 1977.
8. Z. Manna and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, 1992.

9. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Proc. of the 15th Workshop on Protocol Specification, Testing, and Verification,, Warsaw, Poland, June 1995. Chapman Hall.
10. P. Gastin and D. Oddoux. Fast LTL to Buchi automata translation. In G. Berry, H. Comon, and A. Finkel, eds., Computer Aided Verification. 13th International Conference, CAV 2001, vol. 2102 of LNCS. Springer-Verlag, 2001
11. Tauriainen, H., Heljanko, K.: Testing LTL formula translation into Büchi automata. International Journal on Software Tools for Technology Transfer 4(1), 2002
12. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, Springer, Heidelberg, 1999
13. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102. Springer, Heidelberg, 2001
14. Fritz, C.: Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In: Ibarra, O.H., Dang, Z. (eds.) CIAA 2003. LNCS, vol. 2759. Springer, Heidelberg, 2003
15. Esparza J. Automata Theory: An algorithmic approach. // Lecture notes. August 26. 2017. 321 p.
16. Perrin D. Finite automata. Handbook of Theoretical Computer Science. v. B. Elsevier, 1990. P. 1 58.
17. Кривий С.Л. Скінченні автомати: теорія, алгоритми, складність. Київ-Чернівці: Букрек. 2020. 427 с.

18. Gurumurthy, S., Kupferman, O., Somenzi, F., Vardi, M.Y.: On complementing nondeterministic Büchi automata. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860. Springer, Heidelberg, 2003
19. Comon H., Dauchet M., Gilleron R., Jacquemand F., Lugies D., Tison S., Tommasi M. Tree Automata: Techniques and Applications. November 18. 2008. 262 p.
20. Kupferman, O., Vardi, M.: Complementation constructions for nondeterministic automata on infinite words. In: Halbwachs, N., Zuck, L. (eds.) TACAS 2005. LNCS, vol. 3440. Springer, Heidelberg, 2005
21. Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Y. Vardi, Zijiang Yang: Is There a Best Symbolic Cycle-Detection Algorithm? TACAS 2001
22. Jean-Michel Couvreur, Alexandre Duret-Lutz, Denis Poitrenaud: On-the-Fly Emptiness Checks for Generalized Buchi Automata. SPIN 2005
23. Jean-Michel Couvreur: On-the-Fly Verification of Linear Temporal Logic. World Congress on Formal Methods 1999
24. Jaco Geldenhuys, Antti Valmari: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theor. Comput. Sci. (TCS) 345(1)
25. Stefan Schwoon, Javier Esparza: A Note on On-the-Fly Verification Algorithms. TACAS 2005

ДОДАТОК А

(довідниковий)

Текст програми для алгоритму Емерсона-Лея

```
#include "bfs/emerson.hpp"
#include<iostream>

namespace emptiness_check::bfs::emerson
{

/// \namespace Anonymous namespace. Helpers with Emerson-Lei algorithm
namespace
{

std::vector<uint32_t> pre(const automates::inv_buchi &automat,
std::vector<uint32_t> states) noexcept
{
  std::vector<uint32_t> result{};
  for(auto state : states)
  {
     const auto accepted_transitions = automat.acceptable_inv_transitions(state);
     if(accepted_transitions)
        result.insert(result.end(), accepted_transitions.value()->second.begin(),
accepted_transitions.value()->second.end());
  }

  std::sort(result.begin(), result.end() );
  result.erase(unique(result.begin(), result.end()), result.end());

  return result;
}
```

```
std::vector<uint32_t> backward_bfs(const automates::inv_buchi &automat,
std::vector<uint32_t> states) noexcept
{
  std::vector<uint32_t> S = states, B = states, temp_value{};

  do {
     B = pre(automat, B);
     std::set_difference(B.begin(), B.end(), S.begin(), S.end(),
std::back_inserter(temp_value));
     B = temp_value;
     temp_value.clear();
     std::set_union(B.begin(), B.end(), S.begin(), S.end(),
std::back_inserter(temp_value));
     S = temp_value;
     temp_value.clear();
  }while(!B.empty());

  return S;
}

}

bool is_empty(const automates::inv_buchi &automat) noexcept
{
  std::vector<std::unordered_set<uint32_t>> final_states =
automat.get_final_sets();
  std::vector<std::vector<uint32_t>> F;
  std::vector<uint32_t> OldL{}, temp_value{}, newL{};
  std::vector<uint32_t> L(automat.m_set_of_states);
  std::sort(L.begin(), L.end());

  for(auto i = 0; i < final_states.size(); i++){
     std::vector<uint32_t> temp_value;
     temp_value.assign(final_states[i].begin(), final_states[i].end());
```

```
      std::sort(temp_value.begin(), temp_value.end());
      F.push_back(temp_value);
   }

   do {
      OldL = L;
      for(auto Fi : F)
      {
         std::set_intersection(OldL.begin(), OldL.end(), Fi.begin(), Fi.end(),
std::back_inserter(temp_value));
         L = pre(automat, temp_value);
         temp_value.clear();
         std::set_difference(L.begin(), L.end(), OldL.begin(), OldL.end(),
std::back_inserter(temp_value));
         L = temp_value;
         temp_value.clear();
         L = backward_bfs(automat, L);
         std::set_union(L.begin(), L.end(), OldL.begin(), OldL.end(),
std::back_inserter(temp_value));
         L = temp_value;
         temp_value.clear();
      }
   }while( OldL != L);

   if (std::find(L.begin(), L.end(), automates::inv_buchi::INITIAL_STATE) !=
L.end())
      return false;
   else
      return true;
}

} // namespace emptiness_check::dfs::emerson
```

ДОДАТОК Б

(довідниковий)

Текст програми для модифікованого алгоритму Емерсона-Лея

```
#include "bfs/modified_emerson.hpp"
#include<iostream>

namespace emptiness_check::bfs::modified_emerson
{

/// \namespace Anonymous namespace. Helpers with Modified Emerson-Lei
algorithm
namespace
{

std::vector<uint32_t> pre(const automates::inv_buchi &automat,
std::vector<uint32_t> states) noexcept
{
  std::vector<uint32_t> result{};
  for(auto state : states)
  {
     const auto accepted_transitions = automat.acceptable_inv_transitions(state);
     if(accepted_transitions)
        result.insert(result.end(), accepted_transitions.value()->second.begin(),
accepted_transitions.value()->second.end());
  }

  std::sort(result.begin(), result.end() );
  result.erase(unique(result.begin(), result.end()), result.end());

  return result;
}
```

```
std::vector<uint32_t> backward_bfs(const automates::inv_buchi &automat,
std::vector<uint32_t> states) noexcept
{
  std::vector<uint32_t> S = states, B = states, temp_value{};

  do {
     B = pre(automat, B);
     std::set_difference(B.begin(), B.end(), S.begin(), S.end(),
std::back_inserter(temp_value));
     B = temp_value;
     temp_value.clear();
     std::set_union(B.begin(), B.end(), S.begin(), S.end(),
std::back_inserter(temp_value));
     S = temp_value;
     temp_value.clear();
  }while(!B.empty());

  return S;
}

std::vector<uint32_t> inf(const automates::inv_buchi &automat,
std::vector<uint32_t> S) noexcept
{
  std::vector<uint32_t> OldS{}, temp_value{}, pre_s{};

  do {
     OldS = S;
     pre_s = pre(automat, S);
     std::set_intersection(S.begin(), S.end(), pre_s.begin(), pre_s.end(),
std::back_inserter(temp_value));
     S = temp_value;
     temp_value.clear();
  }while(S != OldS);
```

```
  return S;
}

}

bool is_empty(const automates::inv_buchi &automat) noexcept
{
  std::vector<std::unordered_set<uint32_t>> final_states =
automat.get_final_sets();
  std::vector<std::vector<uint32_t>> F;
  std::vector<uint32_t> OldL{}, temp_value{};
  std::vector<uint32_t> L(automat.m_set_of_states);
  std::sort(L.begin(), L.end());

  for(auto i = 0; i < final_states.size(); i++){
     std::vector<uint32_t> temp_value;
     temp_value.assign(final_states[i].begin(), final_states[i].end());
     std::sort(temp_value.begin(), temp_value.end());
     F.push_back(temp_value);
  }

  do {
     OldL = L;
     for(auto Fi : F)
     {
        L = inf(automat, OldL);
        std::set_intersection(OldL.begin(), OldL.end(), Fi.begin(), Fi.end(),
std::back_inserter(temp_value));
        L = pre(automat, temp_value);
        temp_value.clear();
        L = backward_bfs(automat, L);
     }
  }while( OldL != L);
```

```
  if (std::find(L.begin(), L.end(), automates::inv_buchi::INITIAL_STATE) !=
L.end())
     return false;
  else
     return true;
}

} // namespace emptiness_check::dfs::modified_emerson
```

ДОДАТОК В

(довідниковий)

Текст програми виклику алгоритмів перевірки

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

#include "bfs/emerson.hpp"
#include "bfs/modified_emerson.hpp"

/// \brief Author info
constexpr static const char* AUTHOR_TEXT = "\
\n\
Author: Maksym Mykhailenko\n\
  Github: @karlion\n\
Year: 2021\n\
\n";

/// \brief Program calculation mode description
constexpr static const char* INFO_TEXT = "\
Emptiness check: Algorithms based on breadth-first search \n\
\n\
We present two emptiness algorithms that explore A using breadth-first search
(BFS):\n\
  1. Emerson-Lei -- used by default for both automaton types (NBA/NGA)\n\
  2. Modified Emerson-Lei - used by NBA only\n\
NBA -- nondeterministic Büchi automaton;\n\
NGA -- nondeterministic Generalized Büchi automaton;\n\
This program also provide NGA-to-NBA converter. For greater compatibility.\n\
\n\
'THIS_BINARY' usage:\n\
\n\
```

```
--generator        [number > 0]    enable generation mode. See more info by calling
it with help parameter;\n\
--help             [NONE/bool]     Show info about this binary (programm);\n\
--nba              [NONE/bool]     Works only with NBA (converts NGA if
needed);\n\
--modified         [NONE/bool]     Invokes modified algorithm. \n\
--in_file          [text]          Input file name where we store interested automaton;\n\
--out_file         [text]          Output file name where we will dump converted
automaton (if exists);\n\
***********************\n\
Return true or false for selected algorithm\n\
\n";

/// \brief Handle default usage case: calculation of the input automaton
/// \param opts: parsed command line options
void handle_user_case_call(const emptiness_cmd_helper::options& opts) noexcept
{
  using namespace emptiness_cmd_helper;
  auto automaton = proceed_data(opts.in_file);
  auto nba_automaton = (opts.nba || opts.modified) ?
                   std::move(proceed_conversion(automaton, opts.out_file)) :
                   std::nullopt;

  // Return reference on the work automaton (to prevent redundant copying)
  auto get_worker = [&automaton, &nba_automaton]
     { return automates::inv_buchi(std::move(nba_automaton ? *nba_automaton :
automaton)); };

  using namespace emptiness_check::bfs;
  std::cout << std::boolalpha << "...\n";
  if (opts.modified)
     std::cout << "Modified Emerson: " <<
modified_emerson::is_empty(get_worker()) << "\n";
  else
```

```
      std::cout << "Emerson: " << emerson::is_empty(get_worker()) << "\n";
}

/// \brief Initialize callbacks for generation
/// \param opts: generation options that will be passed for appropriate callback
/// \return callbacks handler object
inline emptiness_check::statistic::callbacks_handler<automates::inv_buchi>
intialize_callbacks(
      const utils::generator::generator_opts &opts) noexcept
{
   return {
          .generation_fn = [&opts]() { return
automates::inv_buchi(utils::generator::generate_automaton(opts)); },
          .conv_fn = [](const automates::inv_buchi& at) ->
std::optional<automates::inv_buchi>
          {
             /// \note Yes, slicing
             auto nba = utils::converters::nga2nba(static_cast<automates::buchi>(at));
             return nba ?
                 std::make_optional(automates::inv_buchi(std::move(*nba))) :
                 std::nullopt;
          },
          .nba_algorithms = {
                 &emptiness_check::bfs::emerson::is_empty,
                 &emptiness_check::bfs::modified_emerson::is_empty
          },
          .nga_algorithms = {
                 &emptiness_check::bfs::emerson::is_empty
          }
   };
}

/// \brief Program entry point: parsing arguments, running selected program mode
/// \param argc: the number of command line arguments
```

```
/// \param argv: list of command-line arguments
int main(int argc, const char *argv[])
{
  emptiness_cmd_helper::command_line<automates::inv_buchi>(argc, argv,
                    { AUTHOR_TEXT, INFO_TEXT,
                     {"NBA EMERSON", "NGA EMERSON" },
                     &handle_user_case_call, &intialize_callbacks });
  return 0;
}
```

ДОДАТОК Г

(довідниковий)

Текст програми переводу ЛТЛ формули в автомат

```
#include "LtlGraph.hpp"
#include <cstdio>

/** Read an LTL formula from standard input
* @param negation flag: negate the formula
* @return    the parsed formula, or NULL on error
*/
Ltl* readFormula(bool negation)
{
  int ch;
  class Ltl *left, *right;
  while ((ch = getchar()) == ' ' || ch == '\n' || ch == '\r' ||
        ch == '\t' || ch == '\v' || ch == '\f');
  switch (ch)
  {
    case 't':
    case 'f':
      return &LtlConstant::construct((ch == 'f') == negation);
    case 'p':
      if (1 == scanf("%u", &ch))
        return &LtlAtom::construct(ch, negation);
      fputs("error in proposition number\n", stderr);
      return 0;
    case '!':
      return readFormula(!negation);
    case '&':
    case '|':
      if ((left = readFormula(negation)) && (right = readFormula(negation)))
        return &LtlJunct::construct((ch == '&') != negation, *left, *right);
      return 0;
```

```
      case 'i':
        if ((left = readFormula(!negation)) && (right = readFormula(negation)))
          return &LtlJunct::construct(negation, *left, *right);
        return 0;
      case 'e':
      case '^' :
        if ((left = readFormula(false)) && (right = readFormula(false)))
          return &LtlIff::construct((ch == 'e') != negation, *left, *right);
        return 0;
      case 'X':
        return (left = readFormula(negation))
              ? &LtlFuture::construct(LtlFuture::next, *left)
              : 0;
      case 'F':
      case 'G':
        return (left = readFormula(negation))
              ? &LtlFuture::construct((ch == 'F') == negation
                                      ? LtlFuture::globally
                                      : LtlFuture::finally, *left
               )
              : 0;
      case 'U':
      case 'V':
        if ((left = readFormula(negation)) && (right = readFormula(negation)))
          return &LtlUntil::construct((ch == 'U') == negation, *left, *right);
      case EOF:
        fputs("unexpected end of file while parsing formula\n", stderr);
        return 0;
      default:
        fprintf(stderr, "unknown character 0x%02x\n", ch);
        return 0;
  }
}
```

```
/** Display the gates for arcs leaving a specific Büchi automaton state
* @param gba     the generalized Büchi automaton
* @param state       the state whose successor arcs are to be displayed
*/
void printGates(const class LtlGraph &gba, unsigned state)
{
  for (LtlGraph::const_iterator n = gba.begin(); n != gba.end(); n++)
  {
    if (n->second.m_incoming.find(state) ==
      const_cast<std::set<unsigned> &>(n->second.m_incoming).end())
      continue;
    const class BitVector &propositions = n->second.m_atomic;
    printf("%u ", n->first);
    /** Previously encountered gate expression */
    const class LtlAtom *gate = 0;

    for (unsigned i = propositions.nonzero(); i;)
    {
      if (gate)
        printf(gate->isNegated() ? "& ! p%u " : "& p%u ",
            gate->getValue());
      gate = &static_cast<const class LtlAtom &>(Ltl::fetch(i - 1));
      if (i) i = propositions.findNext(i);
    }
    if (gate)
      printf(gate->isNegated() ? "! p%u\n" : "p%u\n", gate->getValue());
    else
      puts("t");
  }
  puts("-1");
}

/** Translate a formula and output the automaton to standard output
* @param formula  the formula to be translated
*/
```

```
void translateFormula(const Ltl &formula)
{
  /** The generalized Büchi automaton */
  LtlGraph gba(formula);
  /** acceptance set number and proposition */
  using acceptance_set_t = std::pair<unsigned, const class Ltl *>;
  using acceptance_map_t = std::map<const class Ltl *, acceptance_set_t>;
  acceptance_map_t acceptance_sets;

  /** number of states */
  unsigned numStates = 0;
  /** iterator to states */
  LtlGraph::const_iterator s;

  // construct the acceptance sets
  for (s = gba.begin(); s != gba.end(); s++)
  {
    numStates++;
    const class BitVector &temporal = s->second.m_old;
    for (unsigned i = temporal.nonzero(); i;)
    {
      const class Ltl &f = Ltl::fetch(i - 1);
      switch (f.getKind())
      {
        case Ltl::Until:
          if (!static_cast<const class LtlUntil &>(f).isRelease())
            acceptance_sets.insert(std::pair<const class Ltl *, acceptance_set_t>
                                   (&f, acceptance_set_t
                                        (acceptance_sets.size(),
                                         &static_cast<const class LtlUntil &>
                                         (f).getRight())));
          break;
        case Ltl::Future:
          if (static_cast<const class LtlFuture &>(f).getOp() ==
```

```
              LtlFuture::finally)
              acceptance_sets.insert(std::pair<const class Ltl *, acceptance_set_t>
                                      (&f, acceptance_set_t
                                           (acceptance_sets.size(),
                                            &static_cast<const class LtlFuture &>
                                            (f).getFormula())));
            break;
          default:
            break;
        }
        if (i) i = temporal.findNext(i);
      }
    }

    if (numStates)
    {
      printf("%u %lu", numStates + 1, acceptance_sets.size());
      puts("\n0 1 -1");
     printGates(gba, 0);
    }
    else
      puts("0 0");

    for (s = gba.begin(); s != gba.end(); s++)
    {
      printf("%u 0", s->first);
      const class BitVector &temporal = s->second.m_old;

      // determine and display the acceptance sets the state belongs to
      for (acceptance_map_t::iterator a = acceptance_sets.begin();
           a != acceptance_sets.end(); a++)
      {
        /** flag: does the state belong to the acceptance set? */
        bool accepting = true;
```

```
        for (unsigned i = temporal.nonzero(); i;)
        {
          const class Ltl *f = &Ltl::fetch(i - 1);
          if (f == a->second.second)
          {
            accepting = true;
            break;
          }
          else if (f == a->first)
            accepting = false;
          if (i) i = temporal.findNext(i);
        }

        if (accepting)
          printf(" %u", a->second.first);
      }

      puts(" -1");
      printGates(gba, s->first);
    }
}

int main(int argc, char *argv[])
{
  if (argc > 1)
  {
    fputs("usage: ", stderr);
    fputs(*argv, stderr);
    fputs(" < formula.txt > automaton.txt\n", stderr);
    return 1;
  }

  if (Ltl *formula = readFormula(false))
```

```
  {
    translateFormula(*formula);
    int ch;
    while ((ch = getchar()) == ' ' || ch == '\n' || ch == '\r' ||
        ch == '\t' || ch == '\v' || ch == '\f');
    if (ch != EOF)
    {
      fputs("ignoring extraneous input: '", stderr);
      do putc(ch, stderr); while ((ch = getchar()) != EOF);
      fputs("'\n", stderr);
      return 2;
    }
  }
  else
    return 2;

  return 0;
}
```