НУБіП України

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

УДК 004.94-025.12

«ПОГОДЖЕНО»
Декан факультету
інформаційних технологій
Глазунова О.Г., д.п.н., професор
____
____ __________________ **2021 р.**

«ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ»
Завідувач кафедри комп'ютерної інженерії
Лахно В.А, д.т.н., професор
____
____ _________________ **2021 р.**

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему «Дослідження засобів автоматизації проєктування комп'ютерної системи із використанням формальних методів»

Спеціальність 123 «Комп'ютерна інженерія»

Освітня програма «Комп'ютерні системи і мережі»

Орієнтація освітньої програми освітньо-професійна

Керівник магістерської кваліфікаційної роботи
к.п.н., доцент ________________ Касаткін Д.Ю.
Виконав ________________ Єгоров Є.О

КИЇВ-2021

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерної інженерії

Лахно В.А., д.т.н., професор

"____" ________________ 2021 року

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Єгоров Єгор Олександрович
(прізвище, ім'я, по батькові)

Спеціальність Комп'ютерна інженерія

Освітня програма Комп'ютерні системи і мережі

Орієнтація освітньої програми освітньо-професійна

Тема магістерської кваліфікаційної роботи «Дослідження засобів автоматизації проєктування комп'ютерної системи із використанням формальних методів»
затверджена наказом ректора НУБіП України від "23" жовтень 2020р. №1578»С» Термін подання завершеної роботи на кафедру 30 листопада 2021р.
Вихідні дані до магістерської кваліфікаційної роботи

1.Формальні методи
2.Верифікація комп'ютерних систем
Перелік питань, що підлягають дослідженню:

№ з/п	Питання, що підлягає дослідженню	Строк виконання	Примітка
1.	Аналіз предметної області.	01.09.2021	
2.	Моделювання системи	10.09.2021	
3.	Дослідження засобів автоматизації	17.09.2021	
4.	Результати дослідження	22.10.2021	
5.	Попередній захист	30.11.2021	
6.	Захист	15.12.2021	

Дата видачі завдання "23" жовтня 2020 р.
Керівник магістерської кваліфікаційної роботи ____________ Касаткін Д.Ю
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання ____________ Єгоров Є.О
(підпис) (прізвище та ініціали студента)

НУБіП України

ЗМІСТ

ВСТУП 5
1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 10
1.1 Опис предметної області 10
1.2 Проблематика предметної області 12
1.3 Походження формальних методів 14
1.4 Розширення сфери застосування 15
1.5 Обмежений промисловий вплив 18
1.6 Сприятливий час для формальних методів 21
1.7 Використання формальних методів 23
2 МОДЕЛЮВАННЯ СИСТЕМИ 24
2.1 Завдання формального методу у розробці програм 24
2.2 Призначення специфікації 26
2.3 Проблеми формальних специфікацій 28
2.4 Критерії оцінки методів специфікацій 29
2.5 Формальний метод Model Checking 30
2.6 Метод формальної верифікації UMC 34
2.7 Формальне моделювання систем за допомогою діаграм станів UML 35
2.8 Темпоральна логіка μUCTL 36
3 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ 37
3.1 Верифікація систем 37
3.2 Формальні методи верифікації систем 39
3.3 Перевірка еквівалентності 41
3.4 Перевірка моделі 43

НУБіП України

3.5 Дедуктивний аналіз..46

3.6 Тестування з формальною верифікацією..46

3.7 Практична частина формальних методів..48

ВИСНОВКИ..57

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ..58

ДОДАТОК А..61

НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України

ВСТУП

З моменту появи перших комерційних комп'ютерів у 50-ті роки частина людської діяльності, яка залежить від комп'ютерів, неухильно збільшувалася. Від кінцевих товарів, що використовуються в повсякденному житті (годинник, побутова електроніка, телефони, автомобілі і т. д.) до найбільших національних та міжнародних інфраструктур (енергетика, транспорт і т. д.), багато функцій, які раніше виконувались механічно або електрично, тепер виконуються у цифровому вигляді. Як наслідок, кількість мікроконтролерів та мікропроцесорів нині набагато перевищує (і зростає швидше) загальне населення Землі. Це стало можливим завдяки поєднанню основних досягнень у всіх аспектах інформатики:

- Збільшення обчислювальної потужності, як показано в законах Мура та Кумі, які свідчать, що обчислювальна потужність процесора та кількість операцій, які можуть бути обчислені із заданою кількістю енергії, подвоюється кожні 18–24. місяця;
- Збільшення можливостей зберігання даних, як показано, наприклад, законом Крайдера, що свідчить, що кількість бітів, які можуть зберігатися на магнітних дисках, подвоюється кожні 12 або 18 місяців;
- аналогічно «Інформаційний тиждень» повідомляє, що розмір баз даних найбільших складів з 1998 року зростає надзвичайними темпами;
- Збільшення можливостей підключення, про що свідчить зростання пропускної спроможності електрозв'язку та мобільного трафіку;
- Підвищення продуктивності програмного забезпечення, що дозволило розробити велику кількість програмного забезпечення, зростання якого оцінюється як експоненційне, принаймні у разі програмного забезпечення з відкритим вихідним кодом.

У багатьох випадках комп'ютерна автоматизація забезпечує більш гнучкі та надійні пристрої та інфраструктури, дозволяючи виконувати повторювані

завдання, які раніше виконувались людьми, часто спорадично, з точністю та регулярністю. Однак комп'ютерна автоматизація може також збільшити ризик збоїв або несправностей. Це може мати драматичні наслідки, особливо для двох класів систем:

- Критично важливі для життя системи (звані критично важливими для безпеки системами) - це системи, які у разі відмови або несправності можуть загрожувати життю людей. Типові приклади таких систем можна знайти у транспорті (автомобілі, поїзди, літаки тощо. буд.), в енергетиці (атомні станції тощо. буд.) і медицині (допоміжна хірургія, медичні устрою тощо. буд.).
- Критично важливі системи (також звані критично важливими для бізнесу системами) є ризиками, відмінними від попередніх, оскільки їх відмова або несправність можуть призвести лише до фінансових втрат. Такі ризики зростають для систем, що мають тривалий термін служби, розгорнутих у великій кількості, що інтенсивно використовуються багатьма людьми та/або важких або навіть неможливих для ремонту під час експлуатації. Типовими прикладами є безпілотні космічні кораблі, супутники, банківські програми, системи безпеки тощо.

Порушення безпеки можуть негативно позначитися на здоров'ї та добробуті людини. Крім того, з причин вартості нерідко компоненти, розроблені тільки для критично важливих цілей (наприклад, мікропроцесори, операційні системи, компілятори і т. д.), зрештою використовуються в життєво важливих системах. Існує безліч прикладів відмов комп'ютерних систем. Що стосується апаратних збоїв, можна згадати помилку поділу чисел з плаваючою точкою Pentium (1994) і нестачу набору мікросхем Cougar Point (2011), які коштували Intel 475 мільйонів і один мільярд доларів відповідно. Що стосується програмних збоїв, то у 80-ті роки радіотерапевтичний двигун Therac 25 убив п'ятьох людей через погану розробку програмного забезпечення. Що стосується великомасштабної інфраструктури, то через відмову автоматизованої системи багажу в аеропорту Денвера (1994 р.) відкриття аеропорту затрималося на 16 місяців, а перевитрата коштів перевищила

250 мільйонів доларів. Цей список у жодному разі не є повним, оскільки щотижня форум Risks Digest повідомляє про нові приклади ризиків для населення, пов'язаних з комп'ютерами та комп'ютерними системами.

Існують різні причини відмови або несправності:

- Помилки проектування не дозволяють системі досягти запланованої функціональності. Такі помилки часто виникають на ранніх етапах проектування системи і можуть бути викликані неналежним урахуванням вимог до системи або неточним моделюванням реального середовища, в якому система повинна функціонувати, або математичними помилками у складних рівняннях управління, або помилками у критичних алгоритмах та структурах даних, на які спирається система, або несподівані взаємодії між декількома функціями, які повинні бути забезпечені одночасно, і т. д. і

- Збої обладнання включають фізичні або логічні проблеми в мікропроцесорах, мікроконтролерах, інтегральних схемах, датчиках, виконавчих механізмах і т. д. Деякі проблеми виникають через старіння обладнання, і їх неможливо запобігти; Тому обов'язково щоб системи могли виявляти, справлятися з апаратними збоями і відновлюватися після них.

- Програмні помилки – це логічні помилки під час впровадження програмного забезпечення частина системи. Є багато видів помилок (наприклад, помилки часу виконання, безперервні цикли, взаємоблокування тощо. буд.) залежно від цього, програмне забезпечення буває послідовним, паралельним чи розподіленим.

- Проблеми безпеки виникають, коли система недостатньо надійна, щоб протистояти зловмисникам та/або умисним зловмисникам. В даний час це стає критичною темою, оскільки все більше і більше систем працюють у відкритому світі підключено до Інтернету.

- Проблеми з продуктивністю виникають, коли система не може забезпечити очікувану кількісну продуктивність, наприклад, через те, що вона

виконується занадто повільно або через те, що вона споживає занадто багато енергії або інших ресурсів. Існує безліч систем (наприклад, пристрої обробки зображень, радіомовні мережі, побутова електроніка тощо. буд.), котрим правильна функціональність має лише помірковане значення, але чия додана вартість і зручність використання критично залежить від критеріїв продуктивності.

В ідеально простому світі проектування та впровадження правильних та надійних комп'ютерних систем не повинно бути величезним завданням. Але є практичні причини, які ускладнюють це завдання, чим воно має бути. На додаток до постійної потреби у зниженні витрат та скороченні часу виходу на ринок, п'ять ключових факторів сприяють ускладненню проектування системи:

1. Певні проблеми в апаратному, програмному забезпеченні та проектуванні системи за своєю суттю є складними. Це стосується відмовостійких систем, які повинні відновлюватися після фізичних або логічних збоїв, і паралельних систем, які покладаються на співпрацю та координацію кількох агентів, що виконуються одночасно.

2. Через економічну конкуренцію до систем постійно додаються нові функціональні можливості, щоб забезпечити більш високу цінність для клієнтів. Ця гонка за розширенням функціональності (розповзання функцій) є основною причиною різкого збільшення розміру програмного забезпечення (роздування програмного забезпечення).

3. Складність системи також виникає з економічної конкуренції, оскільки системи часто мають підтримувати чи взаємодіяти з кількома платформами (наприклад, апаратними архітектурами, процесорами, операційними системами, проміжним програмним забезпеченням, комп'ютерними мовами тощо. буд.) й у обробки успадкованих додатків.

4. Прагнення продуктивності спонукає розробників і розробників систем винаходити оптимізовані алгоритми, які забезпечують підвищену продуктивність з допомогою збільшення складності.

НУБіП України

5. Нарешті, потреба в безпеці, яка приходить разом із зростаючою роллю комп'ютерів, змушує розробників систем вводити нові функції (наприклад, процедури автентифікації та авторизації), які збільшують складність і можуть викликати нові проблеми, такі як проблеми конфіденційності.

Отже, найважливішим питанням є забезпечення того, щоб комп'ютерні системи функціонували відповідно до їхніх очікувань. Ця проблема виявлялася давно принаймні з кінця 60-х років. Існують різні підходи до цієї проблеми; ми можемо поділити їх на організаційні та технічні.

- Організаційні підходи розглядають проблему як окремий випадок більш загальної проблеми якості продукції: як побудувати комп'ютерні системи без дефектів? Для підвищення якості були запропоновані різні методології та стандарти, такі як ISO 9001 (Системи менеджменту якості - Вимоги), CMMI (Інтеграція моделі зрілості можливостей) та ISO 15504 (Поліпшення програмного процесу та визначення можливостей).
- Технічні підходи вирішують цю проблему, приділяючи основну увагу самій системі та аспектам інформатики. Багато таких методів були розроблені для створення, тестування та перевірки комп'ютерних систем. Багато хто з них (найчастіше менш дорогі і менш руйнівні) вже прийняті промисловістю та інтегровані в методології розробки продуктів. Ці методи дозволяють запобігти або виявити та усунути більшість помилок у даному продукті. Проте деякі помилки досі залишаються непоміченими, особливо у разі складних систем. Наявність таких залишкових помилок (іноді званих помилками високої якості) є серйозною проблемою для життєво важливих чи критично важливих систем. З цієї причини необхідно вивчити альтернативні та/або додаткові методи.

Ця робота присвячена формальним методам, які вважаються найкращими кандидатами для виходу за рамки тих методів, які зазвичай використовуються в галузі, та які є багатообіцяючим кроком до бездефектних комп'ютерних систем.

Актуальність роботи:

1. Вона спрямований на надання всебічного та неупередженого опису сучасного стану формальних методів, мов, інструментів та методологій.
2. Вона спрямований на те, щоб дати єдине бачення формальних методів шляхом визначення концептуальної основи, де всі основні підходи, запропоновані досі, можуть бути поміщені та зіставлені один з одним.
3. Вона спрямований на забезпечення точної оцінки сильних сторін та обмежень формальних методів: хоча формальні методи мають значні переваги для певних проектів, вони ні в якому разі не є срібною кулею для всіх типів складних продуктів. В цій роботі буде чітко пояснено, що можна зробити, а що не слід робити, використовуючи різні види існуючих формальних методів.
4. Вона спрямований на встановлення методологічних керівних принципів для розгортання та ефективного використання формальних методів у проектах реального масштабу. Зокрема, велика увага приділятиметься включенню формальних методів у потоки промислових зразків разом із рекомендаціями про те, як і де слід використовувати формальні методи, щоб уникнути помилок та підводного каміння, які часто спостерігаються при першому застосуванні формальних методів на нетривіальні проекти.

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

Формальні методи можна розглядати як реакцію вченого на емпіричні підходи, а саме організаційні підходи, які іноді більше зосереджуються на процесі проектування, ніж на самому продукті, та технічні підходи, які значною мірою покладаються на тестування для виявлення (певних, але не всіх) помилок проектування та програмування.

НУБіП України

Формальні методи численні та різноманітні, тому важко дати однозначне визначення, яке однозначно включає та характеризує формальні методи. Ми пропонуємо таке визначення: формальні методи в широкому значенні - це математично добре обґрунтовані методи, призначені для допомоги у розробці складних комп'ютерних систем; у принципі, формальні методи націлені побудова систем без дефектів, або виявлення дефектів у існуючих системах, чи встановлення те, що існуючі системи немає дефектів. Щоб бути більш конкретним, ми можемо згадати три загальні риси, загальні для більшості формальних методів:

- Мови: формальні методи часто пов'язані з математичними позначеннями або комп'ютерними мовами з формальною семантикою, яка може описувати властивості, що очікуютьcя від системи, або конкретні способи якою спроектована система (наприклад, архітектура, алгоритми тощо). Залежно від розглянутого формального методу такі описи можуть належати до різних етапів розробки системи, від вимог, специфікації та проектування до реалізації та виконання під час виконання. Незалежно від аналізованої фази, основна ідея формальних методів полягає в тому, щоб розглядати системи, обладнання або програмне забезпечення як математичні об'єкти, які можуть бути описані та проаналізовані суворо.

- Інструменти: формальні методи часто супроводжуються програмними інструментами, які гарантують, що система, що розробляється, функціонуватиме належним чином (очевидно, за певних припущень). Це можна зробити, спрямовуючи і підтримуючи розробку таким чином, щоб система, що вийшла, функціонувала належним чином (підхід «коригування на основі конструкції»), або перевіряючи на різних етапах, що отримана система не відхиляється від своїх початкових очікувань, так що щоб якнайшвидше виявити будь-яку помилку проектування чи реалізації (формальний підхід до верифікації, який є гілкою верифікації та валідації). Важлива відмінність між формальними методами та традиційними методами тестування полягає в тому, що формальні методи

акцентують увагу на аналізі (в ідеалі) всіх можливих виконань системи, а не лише кількох. Це важливо, якщо необхідно математично продемонструвати правильне функціонування системи, а не лише оцінити його за допомогою ймовірностей.

- Методології: щоб бути ефективними, формальні методи мають бути добре інтегровані у виробничу практику. З цієї причини більшість формальних методів мають методологічні вказівки для правильного використання при розробці систем реального розміру.

1.2 Проблематика предметної області

Будучи більш амбітними, ніж традиційні підходи, формальні методи, природно, складніші, і пов'язані з ними інструменти також складніше створити. Але формальним методам притаманні глибші перешкоди. Ці перешкоди виникають із фундаментальних результатів теорії складності обчислень, які стверджують, що за своєю природою найцікавіші завдання перевірки або неможливо, або дуже важко вирішити автоматично. Головна перешкода – це нерозв'язність результатів. У загальному випадку не існує процедури прийняття рішення (тобто алгоритму), яка могла б вирішити, чи може будь-яка програма P завершитися чи ні (це відомо як проблема зупинки). Точно так само не існує процедури прийняття рішення, яка могла б вирішити, чи буде фактично виконана дана інструкція програми P, чи викличе P помилку часу виконання, чи стане деяка задана змінна X програми P колись нульової і т.д. це проблеми, як відомо, нерозв'язні. Звичайно, якщо проблема нерозв'язна, неможливо створити інструмент перевірки, який завжди вирішує цю проблему для будь-якої системи.

Щоб уникнути проблеми нерозв'язності, потрібно знизити початкові очікування і розглянути менш амбітні цілі. Ми класифікуємо пропоновані стратегії за трьома категоріями, які є ортогональними і можуть бути об'єднані разом:

НУБіП України

- Обмеження на виразність: замість того, щоб розглядати будь-яку систему, можна визначити класи систем, для яких проблема перевірки вирішувана. Наприклад, якщо система, що перевіряється, кінцева або може розглядатися як така (це часто має місце з апаратним забезпеченням і телекомунікаційними протоколами), проблеми перевірки стають вирішуваними, принаймні в принципі (тобто з теоретичної точки зору).
- Обмеження точності: замість того, щоб розглядати проблему верифікації у всій її спільності, можна шукати слабші формулювання тієї ж проблеми, які одночасно вирішуються і становлять практичний інтерес. Основна ідея полягає у обчисленні наближень замість точних рішень. Наприклад, якщо неможливо передбачити точне значення деякої змінної X, можна натомість обчислити область якомога меншого розміру, якій належить значення X. Крім того, якщо неможливо передбачити, чи виконає виконання програми P конкретну помилку часу виконання, можна натомість ідентифікувати певні класи програм P, які ніколи не викличуть таку помилку, і відхилити всі інші програми, правильні чи правильні. ні.

Обмеження автоматизації: замість того, щоб вимагати повністю автоматичної перевірки, можна допустити напівавтоматичну (або частково автоматичну) перевірку, коли в певні моменти потрібне втручання людини. Крім того, можна прийняти процедури напів-рішення, які можуть або завершитися шляхом надання правильного рішення або ніколи не завершитися взагалі. Навіть із зазначеними вище обмеженнями, навіть якщо проблема зроблена розв'язаною або напіврозв'язною, все одно залишаються перешкоди. У багатьох випадках обчислювальна складність залишається високою. Наприклад, багато корисних проблем перевірки (наприклад, проблема логічної здійсненності) є NP-повними і таким чином вимагають експоненційного часу виконання для вирішення. Інші корисні завдання мають ще більшу теоретичну складність, наприклад, рішення в

арифметиці Пресбургера, час вирішення найгіршого випадку якої є двічі експоненційним.

На практиці така висока складність (часто звана комбінаторним вибухом або вибухом складності) може бути такою ж обмеженою, як і нерозв'язність. Навіть якщо комбінаторний вибух не відбувається систематично (оскільки це лише найгірший випадок складності), він забороняє існування інструментів перевірки, які будуть працювати для будь-якої системи будь-якого розміру.

Щоб уникнути комбінаторного вибуху, потрібні творчий підхід і кмітливість як від розробників, так і від користувачів інструментів, і це залишається серйозною проблемою для аналізу великих комп'ютерних систем.

1.3 Походження формальних методів

Важко простежити походження формальних методів; можливо, варто повернутися до організованої НАТО конференції з кризи програмного забезпечення, що проходила у Гарміш-Партенкірхені у 1969 році.

З того часу формальні методи еволюціонували у багатьох напрямках, і сьогодні важко дати вичерпний огляд ситуації. У жовтні 2011 року на сайті Formal Methods Wiki, створеному Джонатаном Боуеном, було перераховано понад сотню різних мов формальних методів; Бібліографія DBLP Computer Science Bibliography повідомила про 1334 наукові статті, у заголовку яких міститься слово «формальний метод»; бази бібліографічних даних Citeseer-beta та Google Scholar повідомили, відповідно, про 12 036 та 223 000 публікацій, що містять ключове слово «формальні методи». Тому ясно, що наукова продукція велика і різноманітна, навіть якщо її точний обсяг не так просто виміряти.

Чому набір формальних методів та пов'язаних з ними інструментів настільки фрагментований?

По-перше, це в жодному разі не відноситься до формальних методів: така ж різноманітність вже спостерігалося для мов програмування та компіляторів.

НУБіП України

По-друге, не можна недооцінювати непередбачені обставини академічної кар'єри та небажані наслідки політики «публікувати чи загинути»: часто легше опублікувати про власний винахід, ніж порівнювати себе з багатьма конкурентами на основі загального формалізму.

Однак у разі формальних методів є також вагомі причини для такого розмаїття підходів. Через вищезазначені проблеми складності необхідно йти на компроміси при розробці мов формальних методів та алгоритмів перевірки.

У багатьох випадках немає єдиного рішення, продиктованого науковими міркуваннями; натомість багато дизайнерських рішень повинні прийматися як суб'єктивні людські рішення, і різні вчені вигадують різні рішення.

Що стосується трьох ортогональних компромісів (обмеження виразності, точності та/або автоматизації), які можна зробити, щоб уникнути проблем нерозв'язності, було неминуче – і навіть бажано – щоб вчені досліджували всі можливості, пробуючи різні обмеження та ретельно вивчаючи кожну конкретну підклас проблем.

Крім того, формальні методи можуть бути спеціалізовані для конкретної області застосування (наприклад, обладнання, програмного забезпечення, телекомунікацій і т. д.), і це четвертий вимір, в якому формальні методи можуть відрізнятися.

1.4 Розширення сфери застосування

З моменту появи формальних методів їх сфера застосування постійно розширювалася:

- Спочатку дослідження в основному були націлені на послідовні програми з упором на семантику програм та використання математичної логіки для формального доказу правильності програми.
- У той же час - і, можливо, раніше, фактично відразу після визначення мереж Петрі в 1962 році - були зроблені зусилля по формалізації паралельних

систем. Було досліджено різні парадигми паралельного програмування, особливо моделі із загальною пам'яттю та передачею повідомлень. Потім ці дослідження розширилися на протоколи зв'язку, розподілені та мобільні системи, поступово призвівши до теорії паралелізму, яку ми знаємо сьогодні.

- Формальні методи також поширилися на апаратні та програмні системи, для яких критичний час відгуку. Було запропоновано різні математичні моделі та алгоритми перевірки для реактивних систем, у яких час обробляється дискретно (т. е. як цокання годин), й у систем жорсткого реального часу, у яких час обробляється безупинно (т. е.).
- Потім були розроблені формальні методи для вирішення питань якості обслуговування та оцінки ефективності. Це була важлива зміна парадигми з переходом від точних до наближених моделей, що дозволило працювати з такими концепціями, як системи м'якого реального часу, в яких час відгуку не критичний, але все ж таки важливо для продуктивності, імовірнісні системи, переходи яких підкоряються імовірнісні закони та стохастичні системи, поведінка яких (наприклад, час відгуку) недетермінована, але може бути передбачена за допомогою розподілу ймовірностей.
- За останні два десятиліття формальні методи розширилися до нових областей додатків, серед яких комп'ютерна безпека, а також теорія управління/гібридні системи/кіберфізика, багатоагентні системи та біоінформатика, і це лише деякі з них.

Паралельно з таким розширенням масштабів змінюється рівень абстракції. Спочатку формальні методи були пов'язані з дуже простими, часто ідеалізованими алгоритмічними мовами (наприклад, команди Дейкстри, що охороняються) або високорівневими дуже абстрактними моделями систем (наприклад, мережі Петрі).

Згодом формальні методи стають дедалі ближчими до деталей і складностям нижчого рівня реальних систем. Недавні підходи навіть зводяться до того, що код складання, код C із задіяними функціями, такими як покажчики та потоки, та

реалістичне моделювання параметрів платформи (апаратне забезпечення, операційна система, проміжне програмне забезпечення тощо).

З моменту появи формальних методів їх сфера застосування постійно розширювалася:

- Спочатку дослідження в основному були націлені на послідовні програми з упором на семантику програм та використання математичної логіки для формального доказу правильності програми.
- У той же час - і, можливо, раніше, фактично відразу після визначення мереж Петрі в 1962 році - були зроблені зусилля щодо формалізації паралельних систем. Було досліджено різні парадигми паралельного програмування, особливо моделі із загальною пам'яттю та передачею повідомлень. Потім ці дослідження розширилися на протоколи зв'язку, розподілені та мобільні системи, поступово призвели до теорії паралелізму, яку ми знаємо сьогодні.
- Формальні методи також поширилися на апаратні та програмні системи для яких час відгуку має вирішальне значення. Були запропоновані різні математичні моделі та алгоритми перевірки для реактивних систем, в яких час обробляється дискретно (тобто як такти годинника), і для систем жорсткого реального часу, в яких час обробляється безперервно.
- Потім були розроблені формальні методи для вирішення питань якості обслуговування та оцінки ефективності Це була важлива зміна парадигми з переходом від точних до наближених моделей, що дозволило працювати з такими концепціями, як системи м'якого реального часу, в яких час відгуку не критичний, але все ж таки важливо для продуктивності, ймовірнісні системи, переходи яких підкоряються ймовірні закони і стохастичні системи, поведінка яких (наприклад, час відгуку) недетермінована, але може бути передбачена за допомогою розподілу ймовірностей
- За останні два десятиліття формальні методи розширилися до нових областей додатків, серед яких комп'ютерна безпека, а також теорія управління /

НУБіП України

гібридні системи / кіберфізика , багатоагентні системи ми і біоінформатика, і це лише деякі з них. Паралельно з таким розширенням масштабів змінюється рівень абстракції. Спочатку формальні методи були пов'язані з дуже простими, часто ідеалізованими алгоритмічними мовами (наприклад, команди Дейкстри, що охороняються) або високорівневими дуже абстрактними моделями систем (наприклад, мережі Петрі). Згодом формальні методи стають дедалі ближчими до деталей і складностям нижчого рівня реальних систем. Недавні підходи навіть зводяться до того, що код складання, код С із задіяними функціями, такими як покажчики і потоки, і реалістичне моделювання параметрів платформи (апаратне забезпечення, операційна система, проміжне і т. д.).

1.5 Обмежений промисловий вплив

Незважаючи на ці успіхи, формальні методи зазвичай не використовуються в промисловості (ні в академічних колах), за помітним винятком двох класів областей додатків, в яких формальні методи відіграють значну роль:

- критичні системи, для яких помилки особливо дорогі і їх важко чи неможливо виправити після випуску системи: це випадок апаратних схем та архітектур, до яких метод програмних виправлень зазвичай не застосовується. Великі компанії, що займаються розробкою апаратного забезпечення, наймають експертів за формальними методами та використовують формальні інструменти перевірки (наприклад, засоби перевірки моделей та засоби доказу теорем) як частину своїх виробничих процесів.

- Ті життєво важливі системи, для яких формальні методи потрібні за законом технічними стандартами або органами сертифікації: наприклад, випадок цивільної авіоніки, залізниць та ядерної енергетики.

Те ж зауваження про успіх формальних методів у проектуванні критичних систем та обладнання також з'явилося у нещодавньому звіті [KTVW11]. У період з 1985 по 1995 рік формальні методи також інтенсивно використовувалися для

специфікації протоколів та сервісів OSI (Open System Interconnection), але це використання зменшилося, коли стандарти OSI були залишені на користь TCP/IP, який не вимагає формальних методів, а просто вимагає наявності двох різних реалізацій протоколу.

Інформаційні системи з високим рівнем безпеки, навіть якщо вони не завжди є життєво важливими, також підпорядковуються строгим обмеженням сертифікації, таким як стандарт ISO 15408 (Загальні критерії оцінки безпеки інформаційних технологій) та його рівні оцінки.

У глобальному масштабі використання формальних методів у промислових проектах залишається пунктуальним, переважно призначеним на вирішення конкретних завдань. Таке використання відбувається швидше з індивідуальних ініціатив («героїчні зусилля», у термінології CMMI), ніж з усталених методологій.

Фактично, немає загального консенсусу щодо того, які формальні методи слід використовувати або для якої частини діяльності з розробки слід запроваджувати формальні методи. Цей обмежений промисловий вплив також відображається у поточній ситуації з програмними інструментами для формальних методів.

Такі інструменти дороги у розробці та адаптації до конкретних областей додатків. В даний час інструменти, які продаються найкраще (наприклад, Simulink та UML), не є формальними. Ринок «справді формальних» методів в даний час є дуже вузькою нішою і страждає від добре відомих.

Ефект "петлі негативного зворотного зв'язку": постачальники програмного забезпечення не наважуються вкладати кошти в інструменти, тому що ринок занадто малий, і до тих пір, поки відсутні промислові інструменти, попит користувачів на формальні методи залишається низьким.

У деяких випадках ситуація погіршується, оскільки комерційні інструменти зникають із ринку без заміни еквівалентними інструментами. Виходячи з нашого великого досвіду, ми можемо згадати три такі випадки технологічно просунутих

інструментів, які більше не доступні, хоча вони фактично використовувалися як у великих промислових проектах, так і для навчання студентів на лекціях в університеті:

- QNAP2 (Пакет 2 для аналізу масової мережі) обслуговування було програмне середовище, що надає мову для моделювання мереж масового обслуговування, а також набір алгоритмів для моделювання дискретних подій та точного вирішення цих моделей. Спочатку розроблений INRIA та Bull [VP84], QNAP2 потім поширювався і покращувався Simulog, але зник після того, як Simulog був куплений Astek у 2003 році.
- ObjectGEODE було програмним середовищем для мови SDL [ITU02]. Він включає розширені функції перевірки, які вже розроблені для мови Estelle [ACD + 93]. Компанія, що розробляє ObjectGEODE, одержала назву Verilog; він був куплений Telelogic у 1999 році, а сама була придбана IBM у 2007 році. Інструмент ObjectGEODE більше не комерціалізується, і, наскільки нам відомо, його функції перевірки не були збережені в жодному іншому продукті IBM.

Esterel [Ber05] - комп'ютерна мова для формального опису реактивних систем та синхронних апаратних схем, яка може бути описана на Esterel на високому рівні абстракції, що дозволяє як формальну перевірку, так і ефективний синтез схем. Ґрунтуючись на дослідженні, спочатку проведеному в INRIA, програмне середовище Esterel Studio було розроблено Esterel Technologies, потім передано Synfora у 2009 році, сама придбана Synopsis у 2010 році. В даний час Esterel Studio більше не доступна, незважаючи на її високу технічну значущість.

На жаль, економічні умови знищують цінні наукові та технологічні результати, перешкоджаючи подальшому поширенню формальних методів із тих місць, де вони вже були прийняті.

Однак, незважаючи на економічні труднощі постачальників комерційних інструментів, технічний вплив формальних методів залишається позитивним. У більшості проектів формальні методи показали покращення якості продуктів за

НУБіП України

рахунок кращої формалізації вихідних вимог та зменшення помилок проектування та програмування.

Формальні методи також скорочують час виходу ринку, дозволяючи раніше виявляти помилки, цим усуваючи основну причину непередбачених затримок у великих промислових проектах.

Нарешті, формальні методи можуть знизити витрати, хоча часто не вистачає даних про створення одного й того самого продукту з використанням формальних методів і без них. Такі результати підтверджуються британським дослідженням [WLBF09], яке є «найповнішим оглядом з будь-коли опублікованих» з промислового використання формальних методів. Згідно з даними, зібраними в ході цього дослідження, переваги формальних методів можна кількісно оцінити наступним чином:

- Вплив на якість: покращення на 92%, відсутність ефекту на 8%
- Вплив на час: покращення на 35%, відсутність ефекту на 53%, погіршення на 12%
- Вплив на вартість: покращення на 37%, відсутність ефекту на 56%, погіршення на 7%.

Найцікавіше те, що надзвичайно більшість респондентів дослідження погодилися з тим, що використання формальних методів було успішним (повністю згодні: 61%, згодні: 34%, змішані думки: 5%).

1.6 Сприятливий час для формальних методів

Формальні методи - це довгострокове колективне підприємство, яке розпочалося кілька десятиліть тому. Оскільки проблема за своєю суттю складна справжній прогрес був важким, і також було досліджено безплідні напрями - наприклад, більшість наукового співтовариства довгий час просувала формальні методи як суто математичні позначення. , у своїй мало уваги приділяється програмним засобам підтримки цих позначень.

НУБіП України

Занадто часто прихильники формальних методів надто обіцяли і не виправдовували очікувань, перетворюючи ентузіазм та очікування на розчарування, розчарування, скептицизм та різку критику. Тим не менш, з часом постійні зусилля спільноти формальних методів дозволили розробити більш досконалі мови, які враховують передісторію та потреби передбачуваних користувачів, більш досконалі інструменти, що забезпечують можливості аналізу, що виходять за межі людського мозку, та більш досконалі методології. , які легше інтегруються у існуючу виробничу практику.

Як згадувалося вище, формальні методи в даний час широко застосовуються для проектування критично важливих систем та обладнання, і їх використання часто рекомендується або навіть наказується технічними стандартами .

Але результати дослідження формальних методів також більш прихованим чином використовуються в сучасних компіляторах з функціями перевірки коду, які стали частиною повсякденного життя дизайнерів і програмістів.

У глобальному масштабі промислова значимість формальних методів швидко зростає, оскільки якість і безпека стають все більш диференціюючими факторами для комп'ютерних систем.

Це залишає місце для поширення формальних методів при проектуванні та побудові складних систем, для яких формальні методи повинні стати стандартною практикою, як і в будь-якій іншій інженерній науці.

Отримати чітке уявлення про формальні методи дуже легко. Як згадувалося вище, набір формальних методів обширний та фрагментований. Те саме і з програмними інструментами, які або призначені для вирішення дуже конкретної проблеми, або якщо вони загального призначення, призначені для певної мови введення.

Крім того , список проблем, до яких можуть бути використані формальні методи, розширюється. Також, деякі підходи є формальними методами, але це не так, тоді як інші підходи не претендують на те, щоб бути формальними, хоча це так.Таким

чином, будь-хто, хто хоче вивчити формальні методи, швидше за все, заплутається, якщо не загубиться, через безліч несумісних підходів та суперечливих визначень.

Наукова література за формальними методами страждає від того ж стану. Існують тисячі статей на конференціях та журнальних статей, більшість з яких зазвичай присвячені конкретним темам, але лише кілька статей про формальні методи загалом.

1.7 Використання формальних методів

Формальні методи поступово стають реальністю, яку ігнорувати професіонали не можуть , для кого ці методи необхідні :

- Менеджерам, які доручають, укладають субпідряд та/або контролюють складні комп'ютерні проекти, успіху яких можуть сприяти формальні методи.
- Розробники, які проектують складне обладнання, програмне забезпечення чи системи, особливо якщо ці системи є життєво важливими або критично важливими. Це включає продукти, представлені на сертифікацію, наприклад продукти з високим рівнем безпеки.
- Компанії, які планують запровадити чи надалі впроваджувати формальні методи для власних розробок продуктів чи відносин із підрядниками.
- Особи, які оцінюють продукти відповідно до стандартизованих критеріїв і, отже, повинні проводити аналіз продуктів та аудит розробки на основі формальних методів.
- Творці формальних інструментів, які прагнуть інтегрувати свої конкретні інструменти у методології розробки та процедури сертифікації.

У більш загальному плані, для підвищення якості та безпеки складних продуктів важливо, щоб достатня кількість людей розуміла формальні методи та знала, як їх застосовувати ефективно та з прибутком.

НУБіП України

2 МОДЕЛЮВАННЯ СИСТЕМИ

2.1 Завдання формального методу у розробці програм

Щоб вивчати методи, що встановлюють правильність програми, можна розглянути модель того, що означає правильність програми. Ми розшукуємо якийсь процес, що встановлює правильну реалізацію програмою певної концепції, яка існує у думках деякого суб'єкта. Концепція зазвичай може бути реалізована багатьма програмами, але лише невелика кількість із них має практичне значення. Ця концепція зображена на Рис. 2.1.

Рис.2.1 Концепція неформальних методів

У поточній практиці концепція зазвичай формулюється неформально (природною мовою) і тому незалежно від використовуваного методу перевірки правильності реалізує її програми (зазвичай використовується тестування) результат застосування цього методу може формулюватися тільки в неформальних термінах.

При формальних методах між концепцією та програмою запроваджується проміжна ланка – специфікація. Мета цієї ланки – забезпечити математичний опис концепції та дозволити встановити правильність програми шляхом доказу еквівалентності програми цієї специфікації. Ця концепція зображена на Рис. 2.2.

НУБіП України

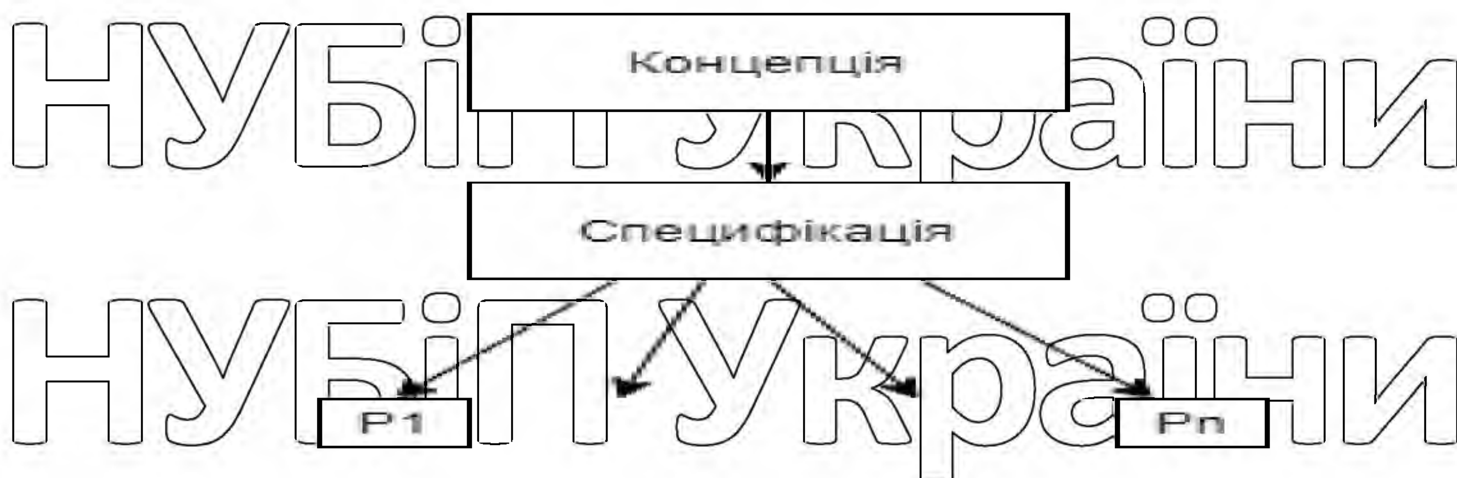

Рис.2.2 Концепція формальних методів

Формальні специфікації також відіграють важливу роль під час побудови програми. Широко визнано, що специфікацію того, що повинна робити програма, слід мати до того, як програма кодується, оскільки специфікація допомагає краще зрозуміти концепцію та підвищує ймовірність того, що програма дійсно реалізовуватиме саме її. Далі формальна специфікація винятково важлива під час використання та модифікації програми.

Таким чином, використання формальної специфікації залучає до розгляду кілька прагматичних аспектів: хто використовує її, для чого вона використовується, коли вона використовується і як вона використовується

Основними користувачами формальної специфікації є:

1) специфікатор – особа, яка становить специфікацію;

2) реалізатор – особа, яка пише програму;

3) верифікатор – особа, яка доводить відповідність програми специфікації; 4) клієнт – особа, яка використовує програму.

НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України

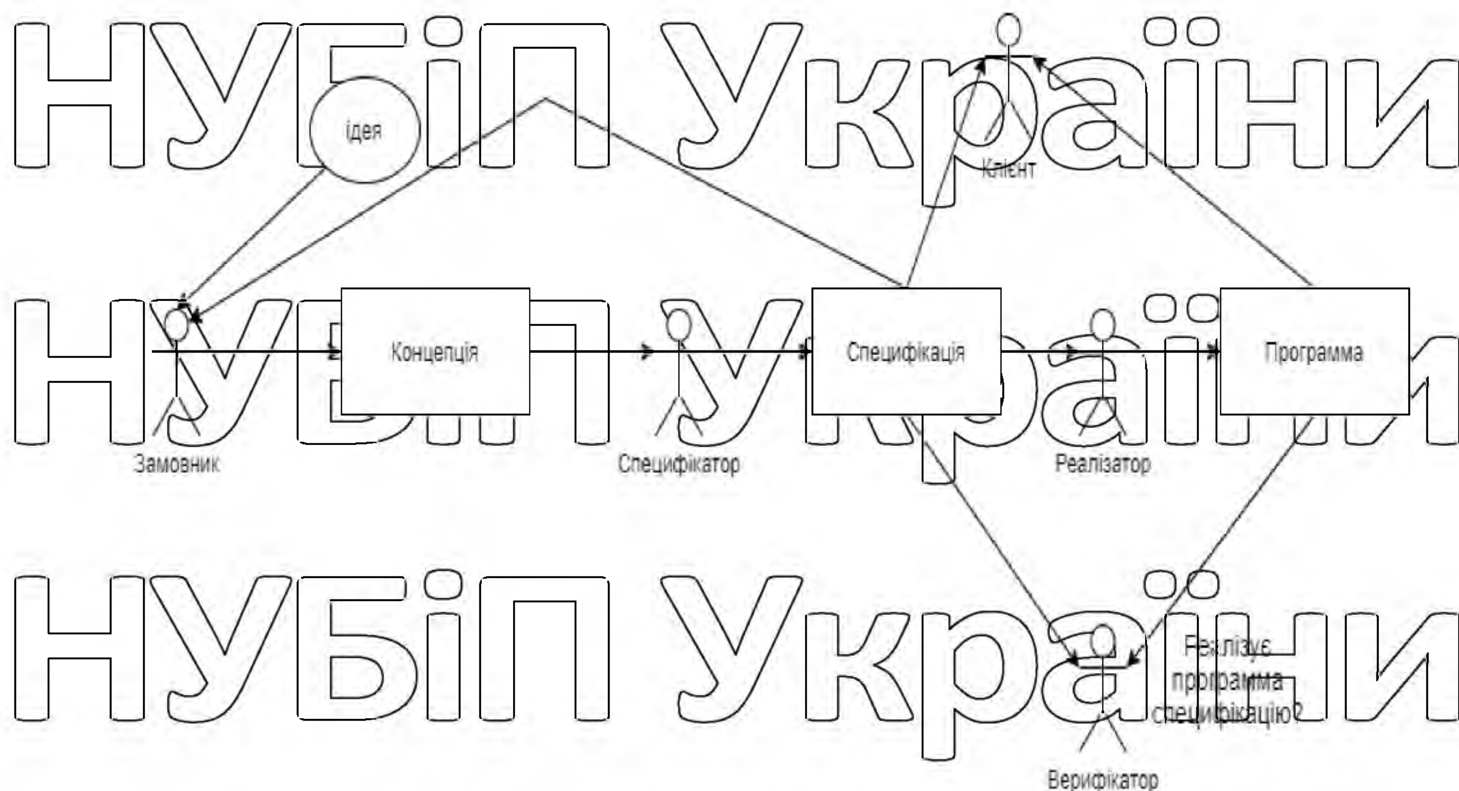

Рис.2.3 Взаємодія осіб формальної специфікації

2.2 Призначення специфікації

1. Опис концепції влучним і недвозначним чином. Складаючи формальний опис концепції, специфікатор змушений аналізувати її до найдрібніших деталей, які ігноруються при неформальному описі.

2. Залучення інструментів виявлення двозначностей і протиріч у описі концепції. Неформальний опис може бути формально проаналізовано, як наслідок, програма найчастіше реалізує концепцію неправильно. Формальна специфікація служить засобом спілкування між замовником та реалізатором програми для того, щоб бути впевненим, що реалізатор правильно розуміє бажання замовника

Формальна специфікація служить також засобом спілкування між кількома реалізаторами, якщо програма реалізується у вигляді кількох модулів, що кодуються різними особами.

НУБіП України

3. Досягнення однозначного розуміння програми її реалізатором та користувачами. Таким чином усуваються суперечки між користувачем та реалізатором щодо правильності виконання програмою її функцій.

4. Визначення правильності реалізації програми та еквівалентності різних реалізацій. При цьому не обов'язково залучення аналітичних методів перевірки. Навіть якщо перевірка правильності програми здійснюється методом тестування, можливе застосування методик, що ґрунтуються на порівнянні формальної специфікації та реалізації.

Результатом такої методики може бути безліч критичних тестових варіантів, які встановлюють, що програма правильно реалізує цю специфікацію. Формальність специфікації означає можливість залучення обчислювальної машини, наприклад для перевірки кроків доказу правильності програми або для генерування тестових варіантів.

5. Підготовка документації програми, яка потрібна на експлуатації та модифікації програми. І тут формальна специфікація сприяє розумінню тексту програми, написаного іншою особою. За відсутності формальної специфікації єдиний шлях програміста, модифікує програму, – порівняти текст програми зі своїми інтуїтивним уявленням у тому, що вона має робити.

Однак інтуїція часто виявляється ненадійним помічником. За наявності формальної специфікації читання тексту програми набуває вигляду неформального доказу, кожен крок якого ґрунтується на розумінні формального опису.

6. Забезпечення засобу спілкування між клієнтом, реалізатором та специфікатором. Специфікація, отримана у процесі проектування програми, служить передачі наміру замовника програми реалізатору і готової програми – користувачеві.

НУБіП України

2.3 Проблеми формальних специфікацій

Формальна специфікація є лише математичним поданням вимог замовника програми. Складаючи її, специфікатор інтерпретує неформальну постановку завдання замовником і записує її деякою формальною мовою специфікацій. У цьому вся процесі виникають кілька проблем, які можуть призвести до неправильної специфікації.

По-перше, специфікатор може неправильно зрозуміти концепцію замовника. Немає формального способу встановлення те, що специфікація охоплює неформальну концепцію. Тому якщо в процесі спілкування із замовником специфікатор неправильно зрозумів його вимоги і тим самим склав специфікацію концепції, яка відрізняється від необхідної, то будь-яка програма, що реалізує специфікацію, не відповідатиме вимогам замовника.

Інтенсивне спілкування специфікатора із замовником під час складання специфікації – єдиний спосіб подолання цієї проблеми. У цьому процесі специфікатор дає початковий опис концепції, знайомить з ним замовника, модифікує опис відповідно до зауважень замовника і так далі, поки замовник не повністю згоден зі складеною специфікацією.

По-друге, специфікатор може припуститися помилок у специфікації. У цьому відношенні основними є дві такі проблеми:

а) повнота - специфікатор може "забути" описати деякі частини програми;

б) несуперечність – специфікатор може описати взаємно суперечливі властивості програми.

Щоб усунути можливі помилки такого характеру, специфікацію необхідно верифікувати. Цю роботу виконує верифікатор (людина чи програма).

НУБіП України

2.4 Критерії оцінки методів специфікацій

Щоб стати корисним, метод специфікації повинен задовольняти низку вимог. Одні їх зачіпають теоретичні аспекти специфікації, інші – практичні аспекти.

1. Формальність. Як уже згадувалося, метод специфікації має бути формальним, тобто специфікації повинні бути написані математично обґрунтованою мовою. Цей критерій є обов'язковим, коли специфікація має використовуватися разом із доказом правильності програми. Крім того, формальні специфікації можуть вивчатися самостійно із залученням математичних засобів, так що можуть шукатися відповіді на різні цікаві питання типу того, чи еквівалентни дві дані специфікації. Нарешті, формальні специфікації повинні розумітися обчислювальною машиною їхньої обробки.

2. Конструктивність. Мова специфікацій має бути простою у вживанні та відповідати концепції, яку необхідно описати. У цьому плані цікаві два аспекти конструювання специфікації: по-перше, труднощі самого конструювання специфікації і, по-друге, труднощі визначення те, що специфікація правильно передає концепцію.

3. Зрозумілість. Людина, яка привчається до нотації, повинна бути в змозі без особливих зусиль читати специфікацію і з мінімальними труднощами відновлювати концепцію, що описується нею.

Тут, як й у попередньому пункті, мають на увазі суб'єктивна міра, через яку складність конструювання і читання специфікації порівнюється зі складністю описуваної концепції (як і інтуїтивно відчувається).

Властивості специфікації, що впливають на її зрозумілість, є розмір і прозорість. Зазвичай коротку специфікацію легше зрозуміти, аніж довгу. Тому добре, якщо специфікація значно менша за реалізуючу її програму. Однак навіть довга специфікація може бути легшою для розуміння, якщо вона ясніше описує концепцію порівняно з програмою.

4. Мінімальність. Специфікація повинна описувати тільки ті властивості концепції, які цікавлять замовника. Ці властивості повинні описуватися точно і недвозначно, але так, щоб при цьому додавали якнайменше сторонньої інформації.

Зокрема, специфікація повинна повідомляти про те, яку функцію повинна виконувати програма, але якнайменше вказувати на те, як ця функція виконується. Однією з причин бажаності цього критерію є те, що властивість мінімальності мінімізує доказ правильності програми шляхом зменшення кількості властивостей, які мають бути доведені.

5. Застосовність. З кожним методом специфікації може бути пов'язаний клас концепцій, які можуть бути ним описані найбільш природним і зрозумілим чином, що веде до специфікацій, що задовольняють критеріям 2 і 3.

Концепції, що не належать цьому класу, можуть бути описані тільки з великими труднощами, або не описані взагалі (наприклад, концепції, що містять паралелізм, не можуть бути описані методом, що не містить відповідних конструкцій).

Зрозуміло, що більше клас концепцій, легко описуваних даним методом, тим паче він корисний. Універсального методу специфікацій, що однаково добре застосовується для опису широкого кола різноманітних концепцій, однак, не існує.

6. Еластичність. Бажано, щоб мінімальні зміни концепції призводили до мінімальних змін специфікації. Цей критерій особливо впливає конструктивність специфікацій.

2.5 Формальний метод Model Checking

В даний час спостерігається стійкий інтерес у застосуванні формальних методів для автоматичного доказу коректності систем, що розробляються. У цьому напрямку використовується два основних підходи:

– автоматичний доказ теорем (Automated Theorem Proving), який базується на принципах дедуктивного аналізу та апарату перед- та пост-умов і

НУБіП України

полягає у використанні набору логічних аксіом та правил виведення для доказу правильності (несуперечності) наявного опису системи;

– верифікація моделі (Model Checking), у межах якого виконується верифікація поведінкових властивостей системи з урахуванням повного розгляду (обходу) всіх можливих станів, у яких система може опинитися у процесі функціонування.

Model Checking – це набір ідей та методів для побудови поведінкових моделей працюючих систем, протоколів чи програм, математичної формулювання вимог до них, що відбивають правильність їхньої роботи, тобто формальної специфікації властивостей, та алгоритмів автоматичної перевірки (доказу чи спростування) цих вимог (властивостей) по всій множині станів моделі.

Якщо модель задовольняє зазначеним вимогам, то програма-верифікатор повідомляє про це. Якщо ж виявляється помилка, то вона надає контрприклад, який показує, за яких умов могла виникнути ця невідповідність.

Контрприклад являє собою сценарій, в якому модель веде себе небажаним чином і порушує одну або кілька властивостей, що верифікуються. Це означає, зазвичай, що модель помилкова і підлягає перегляду. Однак у деяких випадках це може означати, що неправильні формальні вимоги або модель системи побудована некоректно.

Зазначимо, що модель програми не завжди повно відображає її поведінку. Розробник при побудові моделі зазвичай абстрагується від несуттєвих її властивостей. Така концепція дає змогу зменшити розмір самої моделі та прискорити процес її перевірки.

Перевірка моделі дозволяє розробнику виявити помилку та виправити модель чи вимоги. Якщо не знайдено жодної помилки, розробник може удосконалити опис моделі (зробити модель більш реалістичною, взявши до уваги більший набір властивостей), як правило, збільшивши її розмір і перезапустити процес верифікації.

Нубіп україни

Основна складність моделювання – не втратити важливі деталі програми, а складність завдання вимог – сформулювати їх коректно та вичерпно. Для побудови моделі системи в рамках методу формальної верифікації Model Checking використовується структура Крипке – автомат із кінцевим числом станів, що характеризується лише самими станами. Для формулювання вимог до системи застосовуються темпоральні логіки (LTL, CTL та інших.), дозволяють визначити її динамічні характеристики.

Алгоритми для Model checking зазвичай базуються на повному перегляді простору станів моделі: для кожного стану перевіряється, чи він задовольняє сформульованим вимогам. Алгоритми гарантовано завершуються, оскільки кількість станів моделі конечно.

НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України

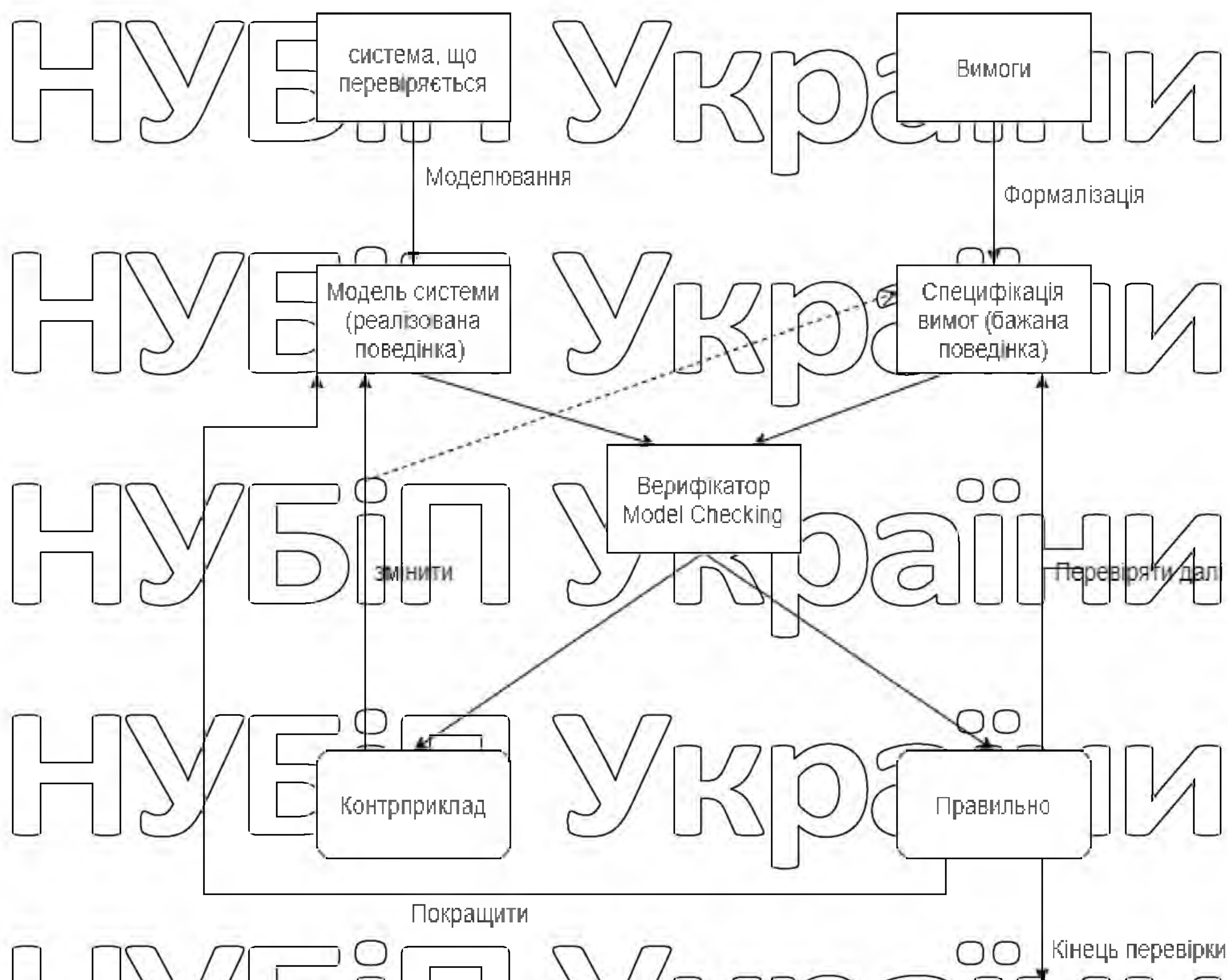

Рис 2.4 Загальний алгоритм формальної верифікації Model Checking

В якості браку традиційних формальних методів, у тому числі і Model Checking, слід зазначити дуже обмежену підтримку сучасних технологій та парадигм програмування. Зокрема для опису моделі системи традиційним є використання абстрактної мови специфікації алгоритмів Promela.

Опис мовою Promela потім може бути перетворено на структуру Крипке. Однак, розроблені на мові Promela моделі суттєво відрізняються від програм, що верифікуються, зазвичай написаних мовами програмування високого рівня, наприклад, Java або C.

НУБіП України

Програми на Promela не мають класів, не використовують покажчиків і не підтримують складні структури даних. Вони представляють плоску структуру взаємодіючих паралельних процесів, мають мінімум конструкцій, що управляють, а всі змінні мають кінцеві області визначення.

Тому на сьогодні одним із перспективних напрямів розвитку методів формальної розробки та верифікації програм є підтримка об'єктно-орієнтованої моделі взаємодії та засобів візуального моделювання.

У зв'язку з цим можна виділити метод формальної верифікації UMC (UML Model Checking), який як вихідну модель системи використовує набір діаграм станів (State chart diagrams) об'єктів, що взаємодіють, розроблених з використанням специфікації уніфікованої мови моделювання UML 2.0.

2.6 Метод формальної верифікації UMC

UMC – метод та інструментальний засіб для формальної верифікації специфікацій, описаних за допомогою темпоральної логіки μUCTL, для перевірки коректності UML-діаграм станів (див. рис. 4).

UMC приймає в якості вхідних даних набір описів станів (що описують динамічну поведінку системи), діаграму об'єктів, що описує початкову конфігурацію системи, набір «спостерігаються критеріїв», який включає атрибути об'єкта і події зв'язку, які ми хочемо спостерігати, а також μUCTLформулу, що представляє властивість , яке необхідно верифікувати.

Формула μUCTL перевіряється «на льоту», інкрементно генеруючи систему станів та переходів L2TS.

НУБіП України

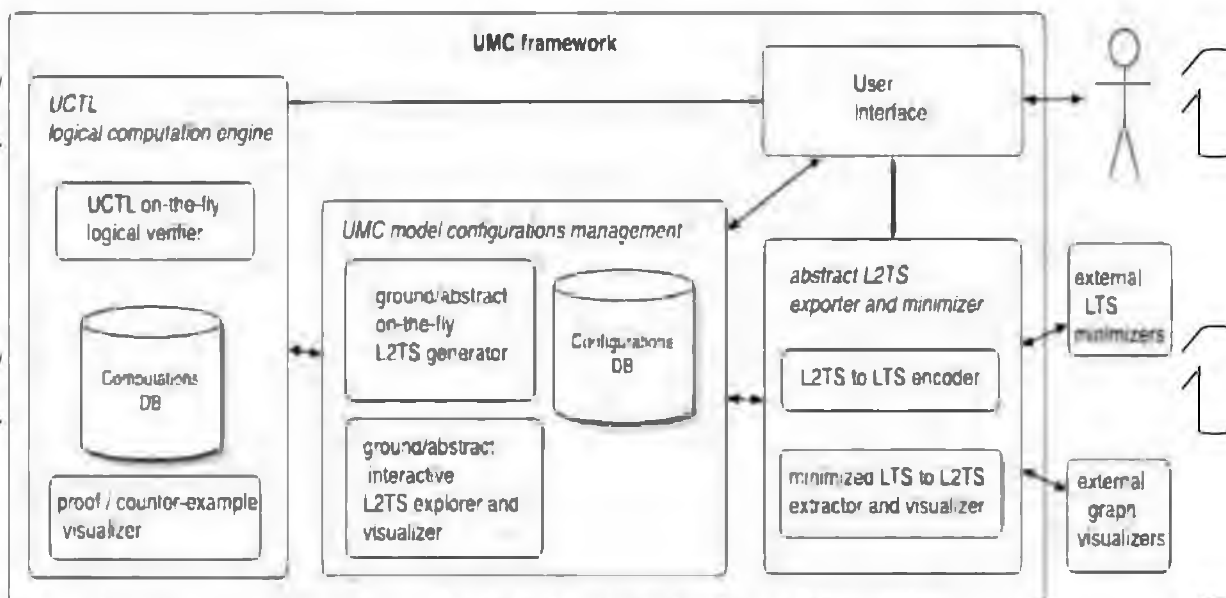

Рис.2.5 Структура методу UMC

2.7 Формальне моделювання систем за допомогою діаграм станів UML

UML (Unified Modeling Language) є уніфікованою мовою графічного моделювання об'єктно-орієнтованих систем [7]. UML підтримує безліч діаграм різного типу та призначення, серед яких діаграми станів, призначені для опису динамічних властивостей систем. UML дозволяє асоціювати діаграму станів з кожним об'єктом, що є системою, а також визначити семантику діаграми станів з точки зору автоматів.

Всі можливі поведінки системи можуть бути описані як можливі еволюції набору взаємопов'язаних автоматів – діаграм станів UML, які можуть бути формально представлені у вигляді системи переходу з подвійною позначкою L2TS (Doubly Labeled Transition Systems) у яких стани становлять змінні системні конфігурації та межі можливих еволюцій системної конфігурації. LTS – це четвірка (Q, q0, Evt, R), де Q – це безліч станів; q0 ∈ Q – початковий стан;

Evt - кінцевий набір подій, що спостерігаються;

R ⊆ Q × 2Evt × Q – відношення переходів.

У свою чергу L2TS – це шістка (Q, q0, Evt, R, AP, L), де

(Q, q0, Evt, R) - те саме, що і в LTS;

AP – набір атомарних речень;

L : Q → 2AP – функція, що позначає кожен стан підмножиною AP.

Тобто L2TS – це, так звана, «структура Крипке з переходами», яка є розширенням структури Крипке шляхом позначки переходів.

2.8 Темпоральна логіка µUCTL

Більшість мов специфікації, що використовуються для опису динамічної поведінки систем, є стан-орієнтованими, або подієво-орієнтованими. У першому випадку опис системи будується довкола внутрішніх властивостей станів; у другому випадку опис будується навколо подій, що відбуваються при переході з одного стану в інший.

Темпоральні логіки є широко визнаним та корисним формалізмом для вираження властивостей живучості та безпеки комплексних систем. Більшість темпоральних логік, що зазвичай використовуються, такі як CTL або LTL, базуються тільки на одній з парадигм (стану/події).

Метод формальної верифікації UMC для специфікації необхідних властивостей UML та L2TS моделей використовує стан/подійно-орієнтовану темпоральну логіку µUCTL .

Формула µUCTL будується над станами Evt.

Семантика подієвої формули. Задовольняюче співвідношення (|=) для подієвої формули форми η |= χ визначено над безліччю подій, як зазначено нижче:

η |= tt (выполняется всегда);

η |= e iff e ∈ η; η |= τ iff η =

∅; η |= ¬χ iff not η |= χ; η

|= χ ∧ χ′ iff η |= χ and η |= χ′.

Семантика подієвої формули вимагає, щоб події e відповідало строго одна подія у η.

Синтаксис µUCTL визначений так:

– формула стану я $\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \wedge \varphi' \mid E\ \pi \mid A\ \pi$;

– формула шляху $\pi ::= X\chi\varphi \mid \varphi\ \chi U\chi'\ \varphi' \mid \varphi\ \chi W\chi'\ \varphi$.

Припустимо, що (Q, q0, Evt, R, AP, L) – это L2TS, $q \in Q$ и $\sigma \in fpath(q')$ для деякого $q' \in Q$. Задовольняюче співвідношення для μUCTL формули описано нижче:

$q \models true$ holds always; $q \models p$ iff p

$\in L(q)$; $q \models \neg\varphi$ iff not $q \models \varphi$; q

$\models \varphi \wedge \varphi'$ iff $q \models \varphi$ and $q \models \varphi'$; $q \models$

$E\ \pi$ iff $\exists\ \sigma \in fpat{=}(q) : \sigma \models \pi$; $q \models$

$A\pi$ iff $\forall\ \sigma \in fpat{=}(q) : \sigma \models \pi$; $\sigma \models$

$X\chi\varphi$ iff $\sigma\{1\} \models \chi$ and $\sigma(2) \models \varphi$;

$\sigma \models \varphi\ \chi U\chi'\ \varphi'$ iff $\exists\ j \geq 1$: $\sigma(j) \models \varphi$, $\sigma\{j\} \models \chi'$ and $\sigma(j + 1) \models \varphi'$ and $\forall\ 1 \leq i < j$: $\sigma(i) \models \varphi$ and $\sigma\{i\} \models \chi$;

$\sigma \models \varphi\ \chi W\chi'\ \varphi'$ iff either $\sigma \models \varphi\ \chi U\chi'\ \varphi'$ or $\forall\ j \geq 1$: $\sigma(j)$

$\models \varphi$ and $\sigma\{j\} \models \chi$.

При верифікації моделей UML може виникнути проблема нескінченності простору станів. Для вирішення цієї проблеми в UMC можна задавати ліміт глибини пошуку у генерації моделі. Якщо правильний результат не знайдено, то збільшується ліміт глибини генерації моделі і процес починається заново.

3 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

3.1 Верифікація систем

Верифікацією називається перевірка відповідності програми (або деякого проміжного результату проектування: прототипу, моделі тощо) вимогам, що висуваються до неї (проектної документації). Якщо програма відповідає вимогам, вона називається коректною (правильною); в іншому випадку програма називається

НУБіП України

некоректною (помилковою), а сам факт невідповідності програми вимогам - помилкою.

Таким чином, верифікація — це аналіз програми щодо наявності чи відсутності у ній помилок. Говорячи перебільшено, результатом верифікації є вердикт: негативний, якщо в програмі знайдено хоча б одну помилку, або позитивний, якщо доведено, що помилок немає. Можливий також невизначений вердикт, коли помилок не знайдено, але їх відсутність не доведена (це поширена ситуація).

Вище було дано формальне визначення — на практиці необхідно враховувати безліч нюансів. По-перше, вимоги, як правило, формулюються неформально, природною мовою, тому не завжди можна однозначно визначити, відповідає їм програма чи ні. По-друге, навіть якщо вимоги формалізовані, довести відсутність помилок у програмі вкрай важко з математичної точки зору (ця робота не цілком піддається автоматизації).

У більшості проектів завдання доказу коректності чи некоректності програмного забезпечення не ставиться7. Типовою метою є розробка якісного продукту, де під «якістю» розуміється складна та розпливчаста характеристика.

Методи верифікації можна поділити на три основні групи (більш детальна класифікація наводиться, наприклад:

1. формальні методи, що використовують математично суворий аналіз моделі програми та моделі вимог;
2. методи тестування, які здійснюють перевірку реальної поведінки програми на деякому наборі сценаріїв;
3. експертизу, що виконується людьми на основі їх знань та досвіду безпосередньо над результатами проектування (наприклад, інспекція коду).

Кожна із зазначених груп методів має свої переваги та недоліки, у кожної своя область застосування. Повноцінна верифікація складних систем

відповідального призначення неможлива без використання різних підходів. Методи верифікації, що поєднують різні техніки, називаються гібридними або синтетичними .

3.2 Формальні методи верифікації систем

Формальна верифікація ґрунтується на математичному (логічному) моделюванні програм та вимог до них. Ідея тут точно така, як при використанні моделей в інших галузях знань: створюється модель — ідеалізоване опис досліджуваного об'єкта чи явища; модель досліджується із застосуванням математичних методів; результати дослідження переносяться на реальний об'єкт чи явище.

Безумовно, застосовність такого підходу визначається моделями, що використовуються — потрібно чітко розуміти умови їх адекватності.

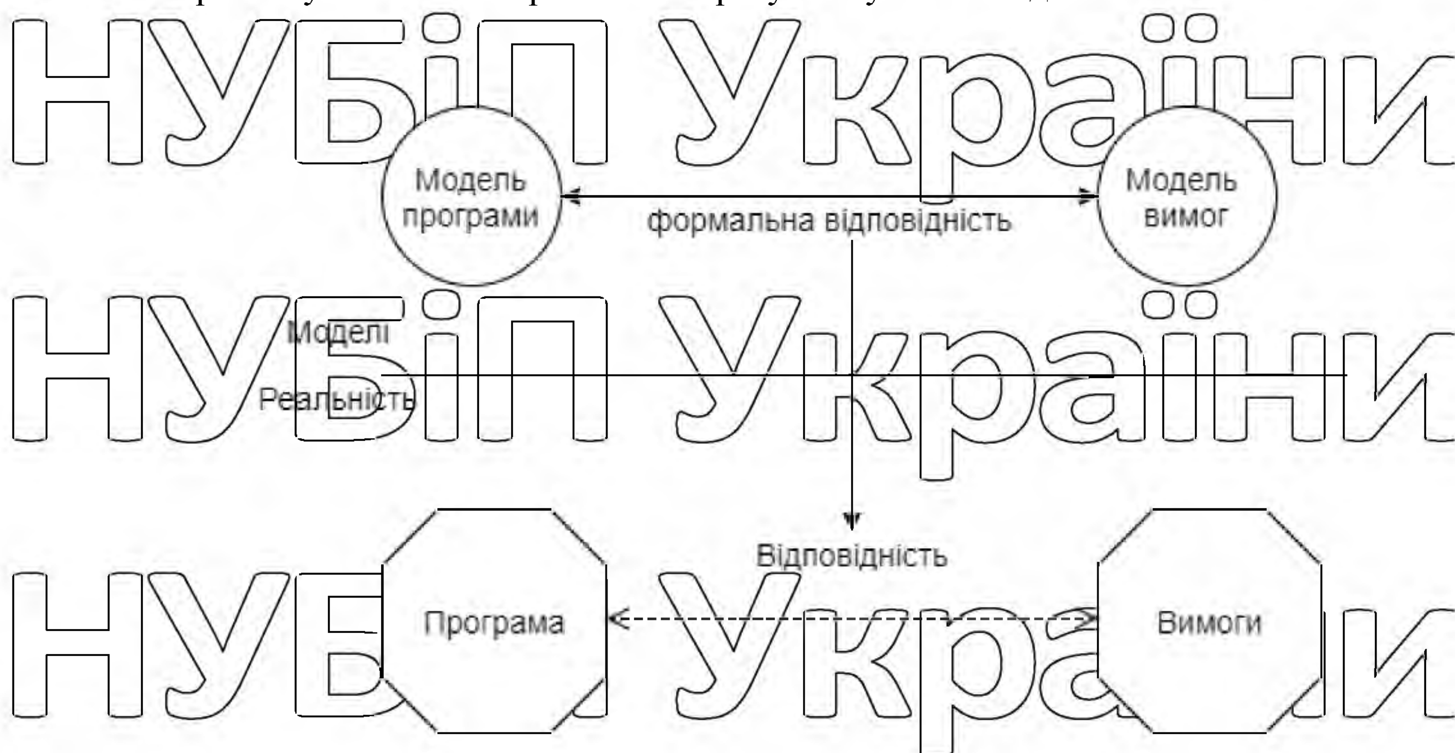

Рис.3.1 Загальна схема формальної верифікації

Загальна схема формальної верифікації показано на рис. 6: створюється формальна модель програми; створюється формальна модель вимог; формально перевіряється відповідність моделі програми моделі вимог; на підставі результатів

перевірки робиться висновок про відповідність чи невідповідність реальної програми реальним вимогам (іншими словами, про відсутність чи наявність помилок у програмі).

Для представлення моделей програм та моделей вимог використовуються відповідно мови формальної специфікації програм (мови моделювання) та мови формальної специфікації вимог;

Методи формальної перевірки дозволяють відслідковувати помилки, які виявляються стандартними методами перевірки. Крім того, у разі помилок, які можуть бути виявлені стандартними методами, формальна перевірка зазвичай виявляє їх значно швидше. Перш ніж проект буде функціонально підтверджений моделюванням та емуляцією, він проходить формальну верифікацію.

Деякі переваги формальної перевірки полягають у наступному:

- Виявляє помилки на ранніх етапах циклу проектування
- Менше часу
- Надійний
- Швидше
- Вичерпний

На наступному малюнку показані різні методи формальної перевірки:

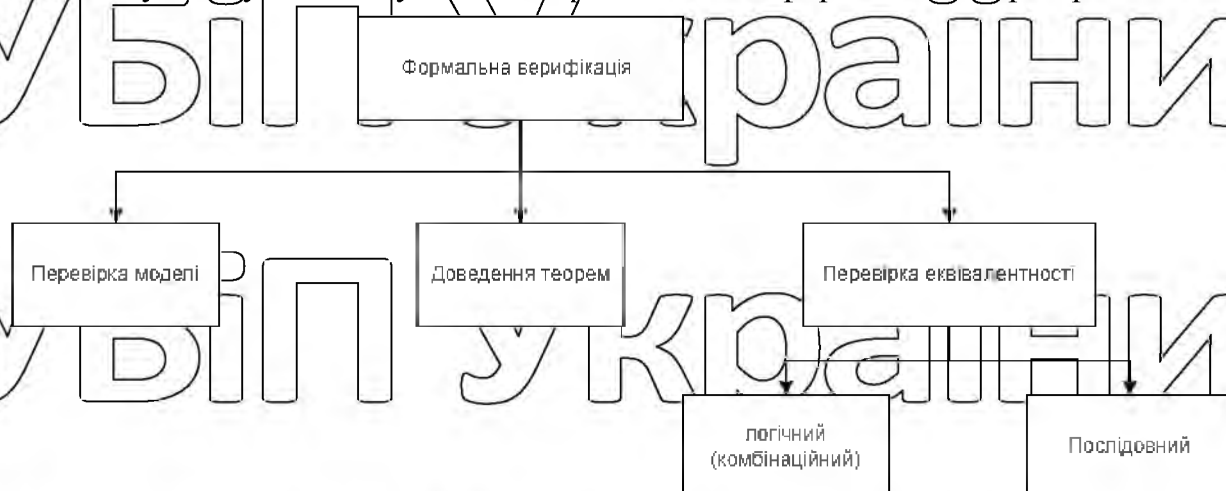

Рис.3.2 Методи формальної перевірки

Нижче ми розглянемо докладніше такі різновиди методів формальної верифікації, як перевірку еквівалентності, перевірку моделей та доведення теорем.

НУБіП України

3.3 Перевірка еквівалентності

Перевірка еквівалентності – це процес перевірки того, що два дизайни функціонально однакові.

Існують два типи методів перевірки еквівалентності: Перевірка логічної еквівалентності (LEC): також відома як перевірка комбінаційної еквівалентності, перевірка логічної еквівалентності – це процес перевірки того, що дві схеми мають однакову комбінаційну логіку між регістрами. Два порівнювані дизайни також повинні мати однакову кількість регістрів.

Послідовна перевірка еквівалентності (SEC): Послідовна перевірка еквівалентності – це процес перевірки того, що два проекти функціонально ідентичні та що вони дають однакові вихідні дані за наявності однакових вхідних даних. SEC порівнює послідовну логіку двох проектів, які можуть мати різні реалізації. Це складний процес і, отже, дуже обмежений розміром дизайну.

Іноді конструкція IC змінюється в останню хвилину, щоб включити деякі функціональні, тимчасові, силові або інші виправлення або включити деяку додаткову логіку, таку як логіка сканування, схеми управління потужністю і так далі. Такі зміни потрібно перевірити. Стандартні процедури перевірки забирають багато часу і, отже, збільшують час виходу ринку.

Цей метод використовується для перевірки функціональної ідентичності двох проектів різних рівнях абстракції; наприклад, список з'єднань на рівні воріт функціонально такий самий, як список з'єднань макета.

У перевірці еквівалентності модель програми і модель вимог однотипні (наприклад, обидві моделі - кінцеві автомати, або обидві моделі - машини Тюрінга), а як відношення відповідності використовується одне із відносин еквівалентності, визначене для моделей типу, що розглядається. При перевірці еквівалентності моделей (автоматів, програм тощо) модель вимог називають еталонною моделлю або еталонною реалізацією.

НУБіП України

Проілюструємо підхід спочатку з прикладу кінцевих автоматів, та був з прикладу послідовних програм.

Кінцевим автоматом Мілі, або просто автоматом, називається шістка(S,X,Y,S0, σ, φ) , в якій S — безліч станів; X — вхідний алфавіт (множина стимулів), , Y-вихідний алфавіт(безліч реакцій), S0∈S – початковий стан , σ: S x X – S – функція виходів (Усі згадані множини вважаються кінцевими та непустими).

Семантика автомата визначається відображенням ланцюжків символів вхідного алфавіту ланцюжка символів вихідного алфавіту. Два автомати над загальними вхідним і вихідним алфавітами називаються еквівалентними, якщо відповідні відображення збігаються.

Розглянемо два автомати і їх діаграми станів показані на Рис.3.3: початкові стани відзначені стрілками, що ведуть у них з «нізвідки».

$$A = \langle \{0,1\}, \{0,1\}, \{0,1\}, 0, \delta_A, \lambda_A \rangle, \text{ где}$$

$$\delta_A(s,x) = x, \text{ для всех } x, s \in \{0,1\}; \qquad \lambda_A(s,x) = s, \text{ для всех } x, s \in \{0,1\};$$

$$B = \langle \{a,b,c,d\}, \{0,1\}, \{0,1\}, a, \delta_B, \lambda_B \rangle, \text{ где}$$

$$\delta_B(s,x) = \begin{cases} a, & \text{если } s \in \{b,d\} \wedge x = 0; \\ b, & \text{если } s \in \{a,c\} \wedge x = 0; \\ c, & \text{если } s \in \{a,d\} \wedge x = 1; \\ d, & \text{если } s \in \{b,c\} \wedge x = 1; \end{cases} \qquad \lambda_B(s,x) = \begin{cases} 0, & \text{если } s \in \{a,b\}; \\ 1, & \text{если } s \in \{c,d\}. \end{cases}$$

Рис.3.3 Діаграми станів автоматів

Очевидно, що автомати A і B еквівалентні - це дві різні реалізації елемента одиничної затримки - D-тригера (обидва автомати видають попередній стимул незалежно від поточного; початкова реакція -0).

НУБіП України

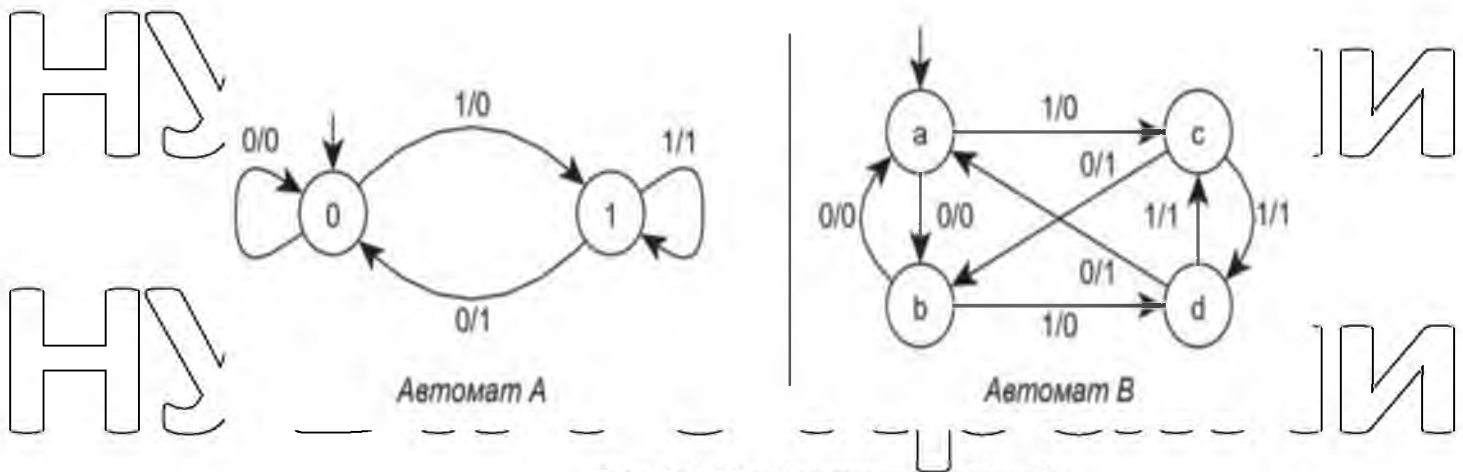

Рис.3.4 Порівняння автоматів

Яким чином можна встановити еквівалентність чи нееквівалентність двох кінцевих автоматів? По-перше, це можна зробити за допомогою методів тестування. Основний результат у цій галузі - теорема Мура про розрізнення станів автоматів, що визначає обмеження на довжину тесту (розрізняючого експерименту) залежно від кількості станів порівнюваних автоматів.

Зауважимо, що тестування можна застосувати, навіть коли автомати є «чорними ящиками», тобто, коли їхня структура не відома. Інший підхід – статичний аналіз відносин переходів. Із цього приводу є інша теорема Мура — теорема про еквівалентність автоматів два кінцеві автомати A і B еквівалентні тоді і тільки тоді, коли в будь-якому досяжному стані їх прямого твору (ssa, $ssss$) для всіх

стимулів xx виконано рівність $\lambda\lambda$a(Sa, xx) = $\lambda\lambda ss$(Sb, xx).

3.4 Перевірка моделі

Перевірка моделі, також відома як перевірка властивостей, є підставою на стані підхід до формальної перевірки.

Наступні кроки пояснюють процедуру перевірки моделі:

1. Змоделюйте систему, щоб отримати модель M. Система моделюється як набір станів із набором переходів між станами, які описують, як система переходить з одного стану до іншого у відповідь на внутрішні чи зовнішні стимули.

2. Створіть властивість для перевірки, використовуючи мову специфікації властивостей, як PSL або SVA, щоб отримати формулу ϕ. Властивість – це опис поведінки проекту.

3. Запустіть програму перевірки моделі, щоб дізнатися, чи модель M формулі ϕ задовольняє.

4. Якщо модель не задовольняє цю властивість, створюється контрприклад. Контрприклад - це стимул, який порушує властивість і зазвичай відображається як хвилі, яку ви можете використовувати при моделюванні.

5. Запустіть контрприклад із моделлю системи у симуляції, щоб знайти місце помилки.

У перевірці моделей вимоги до програм надаються логічними формулами певного виду, а самі програми - структурами, що інтерпретують формули цього виду; відношення відповідності - істинність формули на структурі (програма задовольняє вимогам тоді і лише тоді, коли відповідна структура є моделлю відповідної формули).

Часто для подання вимог використовуються темпоральні, або тимчасові логіки, — логіки, що дозволяють задати взаємозв'язки подій у часі, наприклад, логіка лінійного часу LTL (Linear-time Temporal Logic) або логіка CTL (Computational Tree Logic). Відповідно для представлення програм використовуються структури Крипке та споріднені їм формалізми (розмічені системи переходів).

Нехай задано безліч елементарних висловлювань $AAAA$ (Atomic Propositions). Структурою Крипці над безліччю $AAAA$ називається четвірка (SS, $SS0$, RR, LL), де SS - безліч

станів, $SS0 \leq SS$ - безліч початкових станів, $RR \leq$

LL: $SS \rightarrow 2ÆAA$ — функція розмітки, що позначає кожен стан безліччю істинних у цьому стані .

Структури Крипке використовуються для моделювання реагуючих систем — систем, які працюють у «нескінченному циклі» та взаємодіють зі своїм оточенням. Поведінка системи моделюється обчисленням у структурі Крипке — нескінченною послідовністю станів, перший з яких належить множині S0, а кожна пара послідовних станів входить до RR.

Для опису вимог може використовуватися темпоральна логіка, зокрема LTL. Крім звичайних зв'язок у цій логіці використовуються темпоральні оператори G (Globally), F (in the Future), X (neXt time) та U (Until), які інтерпретуються таким чином:

- G$\varphi\varphi$ - умова $\varphi\varphi$ істинно завжди;
- F$\varphi\varphi$ - умова $\varphi\varphi$ істинно зараз або стане істинною коли-небудь у майбутньому;
- X$\varphi\varphi$ - умова $\varphi\varphi$ істинно в наступний момент часу;
- $\varphi\varphi$ U f — умова $\varphi\varphi$ істинно доти, доки не стане істинною умова f.

Нехай безліч елементарних висловлювань має вигляд $\{llocccceedd$, $llocccc$, $uuuullocccc\}$. Розглянемо структуру Крипке MM = ($\{ss0, ss1, ss2, ss3\}$, $\{ss0\}$, RR, LL), що

моделюють двері із замком (див. рис.7 : початковий стан помічено вхід; істинних у ньому висловлювань).

Стан структури розмічено наступним чином:

- $LL(ss0)= \emptyset$ — двері відчинені; із замком не провадиться жодних дій;
- $LL(ss1)= \{llocccc\}$ — двері відчинені; її закривають;
- $LL(ss2)= \{llocccceedd\}$ — двері зачинені, із замком не виконується жодних дій;
- $LL(ss3) = \{llocccceedd, uuuullocccc\}$ - двері зачинені; її відчиняють.

НУБіП України

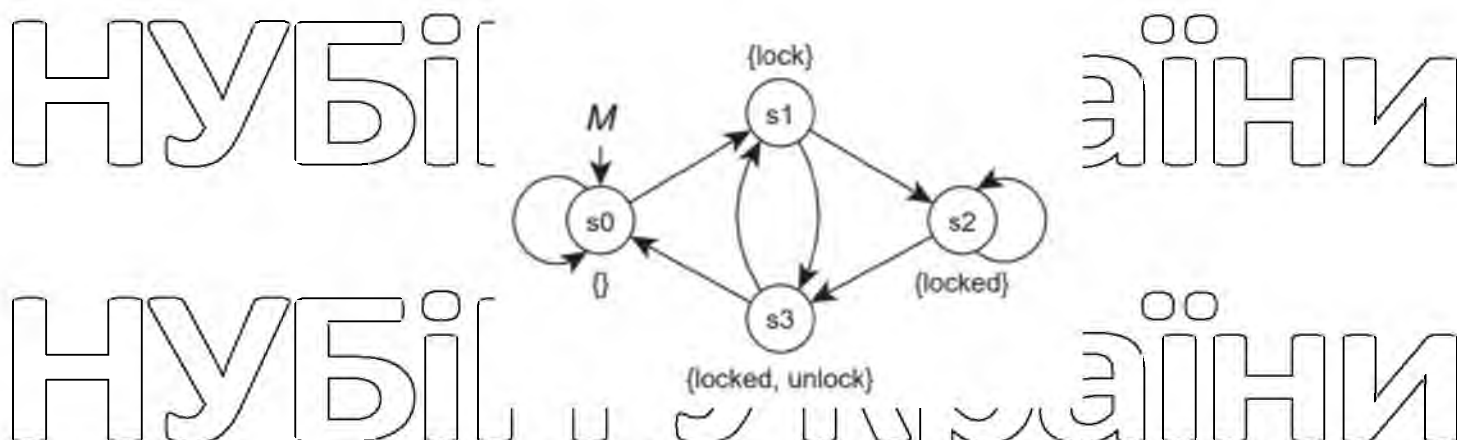

Рис 3.4 Структура Крипке, що моделює дверний замок

3.5 Дедуктивний аналіз

В останньому з підходів, що розглядаються, дедуктивному аналізі, моделлю програми є код мовою з формалізованою семантикою (сама програма, якщо мова програмування формалізована, псевдокод або блок-схема), а моделлю вимог - програмний контракт, тобто. паралогічних формул: перед-і постумова; відношення відповідності - повна або часткова коректність програми щодо контракту.

Для зручності вважатимемо, що змінні програми діляться на три типи: вхідні, внутрішні та вихідні: вхідні змінні містять вихідні дані та не змінюються під час виконання програми; внутрішні змінні використовуються зберігання проміжних результатів обчислень; вихідні змінні призначені для видачі одержаних результатів.

3.6 Тестування з формальною верифікацією

Нагадаємо, що тестуванням називається метод верифікації, що базується на аналізі результатів виконання програми в деяких ситуаціях. Процедури, що описують процес створення цих ситуацій та перевірки, які необхідно виконати над одриманими результатами, називаються тестами .

На відміну від формальних методів, у яких аналіз програми проводиться всім можливих обчислень, тестування має справу з обмеженим безліччю варіантів,

власне, одним із головних завдань тестування є створення репрезентативної вибірки таких варіантів – тестового набору.

З іншого боку, тестування має справу з реальною системою, а не з її моделлю, яка, як відомо, може бути неадекватною . Переваги та недоліки методів тестування та формальної верифікації зведені до Таблиці 3.1.

Методи тестування і формальної верифікації є взаємовиключними, а навпаки, доповнюють одне одного і можуть використовуватися спільно. Можлива наступна організація процесу створення ПЗ, загалом відповідна концепції розробки з урахуванням моделей (model-driven development / engineering):

- створення моделі системи;
- формальна верифікація моделі;
- реалізація системи (з урахуванням моделі);
- тестування системи (з урахуванням моделі).

Таблиця 3.1. Переваги та недоліки методів тестування та формальної верифікації

Тестування	Формальна верифікаця
+Перевіряється реальна система +Перевірка здійснюється у реальних умовах	+Перевіряються всі можливі сценарії +Перевіряється клас можливих реалізацій
-Виконується на пізніх етапах розробки -Перевіряється обмежений набір сценаріїв -Висока трудомісткість розробки тестів -Необхідність великого штату тестувальників	-Перевіряється модель, а не реальна система -Перевіряється обмежений набір властивостей -Висока обчислювальна складність -Високі вимоги до кваліфікації верифікаторів

Більше того, формальні техніки можуть застосовуватися для вирішення завдань тестування: генерації тестових впливів (послідовностей стимулів); перевірки коректності поведінки (реакцій, що видаються у відповідь на стимули); оцінки повноти тестування (репрезентативність набору тестів). Методи тестування, в яких застосовуються формальні моделі та техніки їх аналізу, називаються

методами тестування на основі моделей (MBT, Model-Based Testing) [Utt12]. Перед тим, як розглянути деякі основні практики цього типу, зробимо невеликий екскурс в історію.

Тестування на основі моделей у тому чи іншому вигляді існує з 1970-х рр., проте спочатку застосовувалося лише в таких галузях, як проектування апаратури та телекомунікаційних протоколів [Pet15]. Поступово прийшло усвідомлення, що моделі (формальні специфікації) можна використовувати для тестування ширшого класу систем. Наприкінці 1980-х років. стали з'являтися роботи, які пропонують методи генерації тестів за специфікаціями, на кшталт статті [Ost88].

Це були напівавтоматичні підходи, що дають змогу генерувати описи для подальшої ручної розробки тестів за ними. Повноцінні інструменти почали зароджуватися в середині 1990-х років. Одними з перших технологій тестування на основі моделей стали KVEST [Bur99] і UniTESK [Бур03b], розроблені в ІСП РАН. Обидва підходи базувалися на програмних контрактах та техніках обходу модельних графів станів.

Пізніше з'явився близький до можливостей інструмент SpecExplorer [Bla05] від Microsoft Research, заснований на абстрактних станах машин (ASM, Abstract State Machines).

3.7 Практична частина формальних методів

Зазвичай вимоги до проекту значною мірою пишуться англійською мовою, перемежуючись відповідними таблицями, діаграмами, знімками екрана та описами UML, такими як варіанти використання та діаграми (наприклад, діаграми послідовності, класу та переходу між станами).

Коли ми перевіряємо вимоги до проекту, ми шукаємо відповіді на ряд питань. Ось загальні питання, які ви поставите під час перевірки своїх вимог:

Чи точно вони відображають вимоги користувачів? Чи все, що ви заявили, відповідає тому, що хочуть користувачі, і чи ви включили все, що просили користувачі?

Вимоги чітко написані та недвозначні?

Наскільки вони гнучкі та зрозумілі для інженерів? Наприклад, чи є вимоги відповідними модульними та добре структурованими для полегшення проектування та розробки?

Чи можна використовувати вимоги для простого визначення прикладів випробувань для перевірки відповідності реалізації вимогам?

Чи написані вимоги абстрактно та на високому рівні, далеко від проекту, реалізації, технологічних платформ тощо, щоб дати розробникам достатньо свободи для їх ефективної реалізації?

Знайти відповіді на ці питання – непросте завдання, і немає простого способу зробити це, але якщо ваші вимоги можуть успішно подолати ці труднощі.

Незважаючи на деяку допомогу інструментів моделювання, таких як UML, проблема забезпечення якості вимог залишається. Цей процес вимагає великих витрат часу та вручну, включаючи перевірки та іноді часткове створення прототипу. Використання декількох нотацій (наприклад, у UML) створює додаткові проблеми:

- які позначення використовувати для яких вимог
- як забезпечити відповідність описів у різних позначеннях один одному

Ціна помилок у вимогах часто висока і вимагає як мінімум доопрацювання та обслуговування. Якщо ви реалізуєте неправильні вимоги у тому вигляді, в якому вони є, це може призвести до неправильної поведінки системи в польових умовах і до високих витрат, таких як втрата життя та майна, особливо у вбудованих критично важливих для безпеки вбудованих системах у реальному часі. Аналогічні проблеми існують у забезпеченні якості проектування систем.

Один із способів покращити якість ваших вимог та дизайну –

використовувати автоматизовані інструменти для перевірки якості різних аспектів вимог та дизайну. Але які інструменти? Очевидно, що створення інструментів для перевірки вимог або проектування англійською є надзвичайно складним. Необхідно забезпечити ясну, строгу та недвозначну формальну мову для формулювання вимог. Якщо мова для написання вимог та дизайну має чітко визначену семантику, можливо, буде доцільно розробити інструменти для аналізу тверджень, написаних цією мовою. Ця основна ідея використання строгої мови для написання вимог або проектування тепер є основою для перевірки системи.

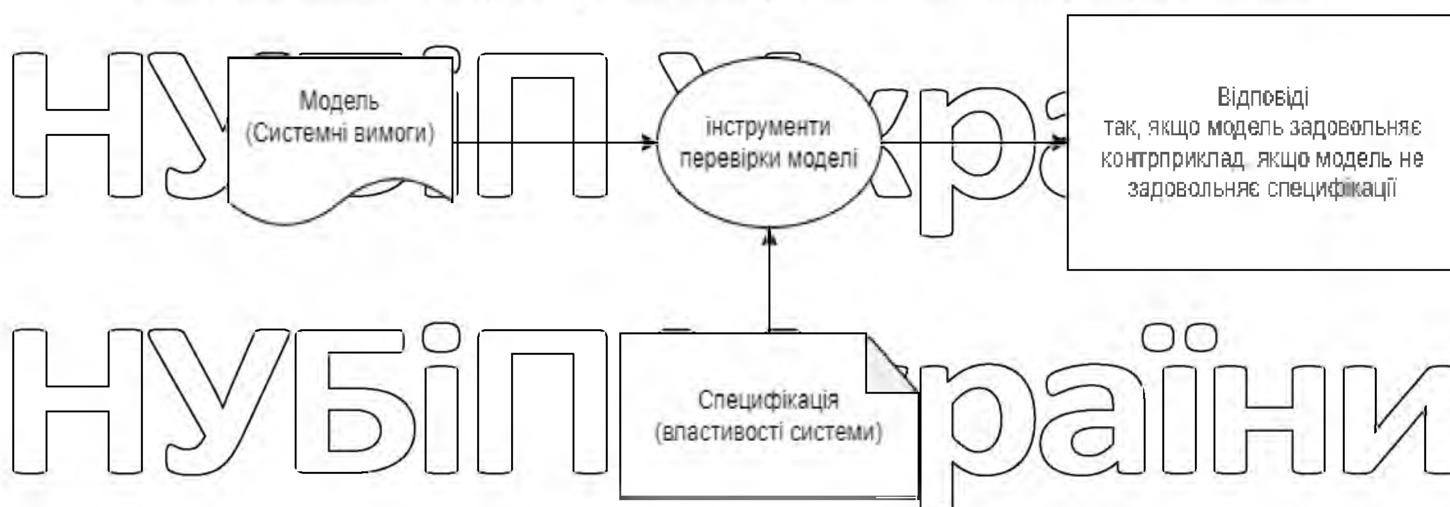

Рис.3.5 Підхід до перевірки моделі

Перевірка моделі – найбільш успішний підхід до перевірки вимог. Основна ідея перевірки моделі показана на малюнку 8. Інструмент перевірки моделі приймає системні вимоги або проект (звані моделями) та властивість (названа специфікацією), якій повинна задовольняти остаточна система.

Потім інструмент виводить так, якщо дана модель відповідає заданим специфікаціям, і генерує контрприклад, в іншому випадку.

Контрприклад докладно визначає, чому модель відповідає специфікації. Вивчаючи контрприклад, можна точно визначити джерело помилки в моделі, виправити модель і повторити спробу. Ідея полягає в тому, що гарантуючи, що модель задовольняє достатнім системним властивостям, ми підвищуємо нашу

впевненість у правильності моделі. Системні вимоги називаються моделями, тому що вони представляють вимоги чи дизайн.

Отже, яка формальна мова працює для визначення моделей? Однозначної відповіді немає, оскільки вимоги (або дизайн) для систем у різних галузях додатків дуже різняться. Наприклад, вимоги банківської системи та аерокосмічної системи розрізняються за розміром, структурою, складністю, характером даних системи та виконуваним операціям.

Навпаки, більшість вбудованих систем реального часу або критично важливих для безпеки систем орієнтовані на управління, а не на дані - це означає, що динамічна поведінка набагато важливіша за бізнес-логіку (структура та операції з внутрішніми даними, що підтримуються системою). Такі орієнтовані управління системи використовуються в різних областях: авіакосмічна промисловість, авіоніка, автомобілебудування, біомедичні прилади, промислова автоматизація і управління технологічними процесами, залізниці, атомні електростанції тощо.

Для систем, орієнтованих на управління, кінцеві автомати широко використовуються як хороша, чиста і абстрактна система позначень для визначення вимог і проектування. Але, звичайно, чистий автомат не підходить для моделювання складних реальних промислових систем. Також нам необхідно:

- мати можливість розбити вимоги на модулі, щоб переглядати їх на різних рівнях деталізації
- мати спосіб комбінувати вимоги (або дизайн) компонентів
- мати можливість вказувати змінні та засоби для їх оновлення, щоб використовувати їх для захисту під час переходів.

Коротше кажучи, нам потрібні розширені кінцеві автомати. Більшість інструментів перевірки моделей мають свою строгу формальну мову для визначення моделей, але більшість з них є різновидами EFSM.

Як можна використовувати перевірку моделі для перевірки властивостей простої вбудованої системи. Для цього ми скористаємося інструментом перевірки моделей символічної моделі (SMV) від Університету Карнегі-Меллона.

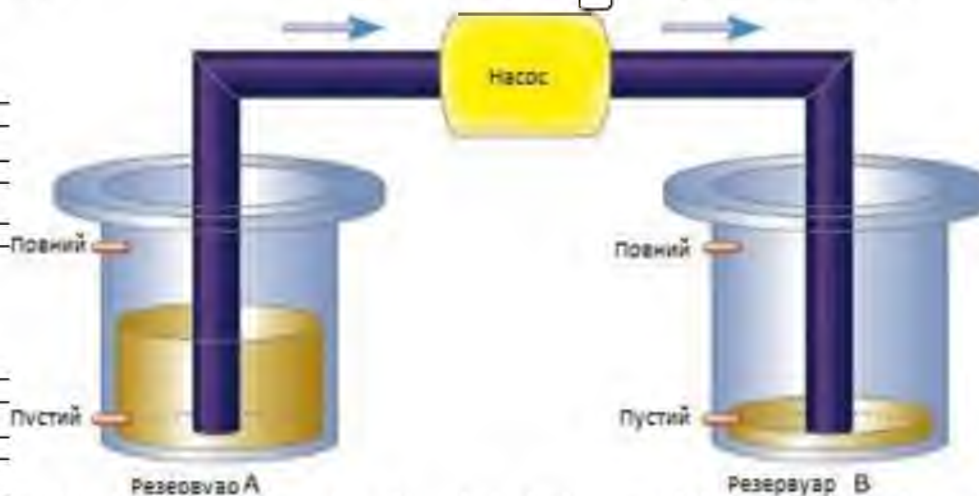

Рис.3.6 Насосна станція

Розглянемо просту систему управління перекачуванням, яка перекачує воду з вихідного резервуару A в інший зливний резервуар B за допомогою насоса P, як показано на малюнку 2. У кожному резервуарі є два вимірники рівня: один для визначення того, чи є його рівень порожнім, а інший - для виміру рівня. визначити, чи його рівень. Рівень у баку нормальний, якщо він не порожній та не повний; іншими словами, якщо він вище порожньої позначки, але нижче за повну позначку.

Спочатку обидва баки порожні. Насос повинен бути увімкнений, як тільки рівень води в баку A досягне нормального (з порожнього), за умови, що бак B не сповнений. Насос залишається увімкненим, поки бак A не порожній і поки бак B не сповнений. Насос необхідно вимкнути, коли резервуар A стане порожнім або резервуар B стане повним. Система не повинна намагатися вимкнути (ввімкнути) насос, якщо він уже вимкнений (ввімкнений). Хоча цей приклад може здатися тривіальним, він легко поширюється на контролер складної мережі насосів та трубопроводів для управління декількома резервуарами-джерелами та стоками, такими як резервуари на водоочисних спорудах або на підприємствах хімічного виробництва.

НУБіП України

Опис моделі SMV та перелік вимог наведено в Додатку А.

Модель цієї системи в SMV виглядає так і показана в лістингу . У першому розділі VAR декларується, що система має три змінні стани. Змінні level_a та level_b записують поточний рівень верхнього та нижнього резервуара відповідно; у кожен «момент» ці змінні набувають значення, яке може бути порожнім, нормально або повним. Змінний насос запис , чи є насос на або вимкнено .

Стан системи визначається набором значень кожної із цих трьох змінних. Наприклад, (level_a = empty, level_b = ok, pump = off) і l (evel_a = empty, level_b = full, pump = on) є можливими станами системи. INIT ділянка, ближче до кінця, визначає початкові значення для змінних (тут, спочатку передбачається, що насос вимкнений, але решта двох змінних може мати будь-яке значення).

Розділ ASSIGN визначає, як система переходить із одного стану в інший. Кожен наступний оператор визначає, як змінюється значення конкретної змінної. Передбачається, що всі ці оператори присвоєння працюють паралельно; наступний стан визначаються як чистий результат виконання всіх операторів надання в цьому розділі. Нижній бак може перейти з порожнього у порожній чи нормальний стан; від ОК до порожнього чи повного чи залишається гаразд, якщо насос включений; від нормального до нормального або повного, якщо вимкнений насос; з повного не може змінити стан, якщо вимкнений насос; від повного до нормального або повного, якщо насос увімкнено. Аналогічні зміни внесено до верхнього резервуару.

Внутрішня більшість інструментів перевірки моделі згладжують (або розгортають) вхідну модель в систему переходів, звану структурою Крипке. Розгортання включає видалення ієрархій в EFSM, видалення паралельних композицій, а також видалення засобів захисту та дій на переходах. Кожен стан у структурі Крипке - це, власне, кортеж, що містить одне значення кожної змінної стану. Перехід у структурі Крипке означає зміну значення однієї чи кількох змінних стану. Ця властивість фактично перевіряється на відповідність структурі Крипке, отриманої шляхом розгортання даної моделі. Однак для розуміння того, що означає

затвердження властивості, структура Крипке розгортається в нескінченне дерево, де кожен шлях у дереві вказує на можливе виконання або поведінку системи.

Спочатку система може перебувати в будь-якому з дев'яти станів, де немає обмежень за рівнем води A або B, але передбачається, що насос вимкнений. Позначимо стан упорядкованим кортежем, де A і B позначають поточний рівень води в резервуарах A і B, а P позначає поточний стан насоса. Для ілюстрації припустимо, що початковий стан. Тоді, відповідно до моделі системи, наступним станом цього стану може бути будь-яке, . Від наступного стану може бути одним з , , , , або . Для кожного з цих станів ми могли визначити наступні можливі стани.

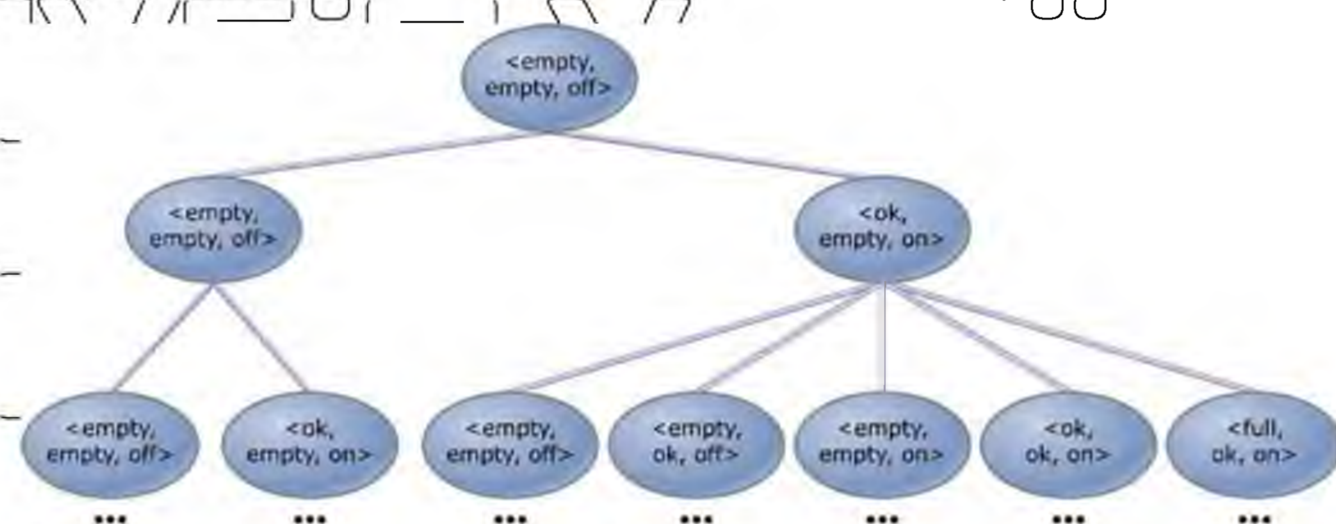

Рис.3.6 Початкова частина дерева виконання системи контролера насоса

Стан може бути організований як нескінченного дерева виконання (чи обчислень) , де корінь позначений нашим обраним початковим станом, а дочірні елементи будь-якого стану позначають такі можливі стану, як показано на Рис.3.7. Виконання системи - це шлях у цьому дереві виконання. Загалом таких шляхів виконання в системі нескінченно багато. Мета перевірки моделі - перевірити, чи дерево задовольняє виконання заданої користувачем специфікації властивості.

Логіка дерева обчислень (CTL) - технічно тимчасова логіка розгалуження - є простою і інтуїтивно зрозумілою системою позначень, придатною для цієї мети. CTL - це розширення звичайної логічної логіки висловлювань (яка включає логічні зв'язки, такі як, або, не має на увазі), де доступні додаткові тимчасові зв'язки.

Таблиця 3.2 Інтуїтивне значення деяких основних темпоральних зв'язок CTL

EF φ	істина в поточному стані, якщо існує деякий стан на певному шляху, що починається в поточному стані, який відповідає формулі φ
AF φ	істина в поточному стані, якщо існує деякий стан на кожному шляху, що починається в поточному стані, який відповідає формулі φ
EG φ	істина в поточному стані, якщо кожен стан на деякому шляху, що починається в поточному стані, відповідає формулі φ
AG φ	істина в поточному стані, якщо кожен стан на кожному шляху, що починається в поточному стані, відповідає формулі φ

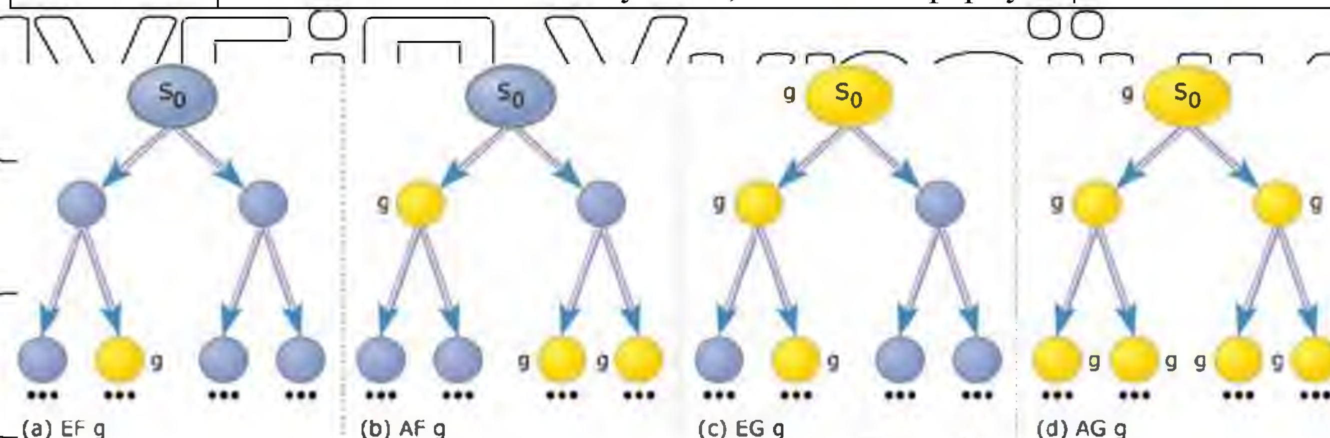

Рис.3.7 Формул CTL, які задовольняються в стані s 0

Таблиця 3.2 і Рис.3.7 ілюструють інтуїтивне значення деяких основних темпоральних зв'язок CTL. Насправді, E (для деякого шляху) і A (для всіх шляхів) є кванторами шляху для шляхів, що починаються зі стану. F (для деякого стану) та G (для всіх станів) є кванторами станів для станів у дорозі.

Враховуючи властивість та (можливо, нескінченне) дерево обчислень T, відповідне моделі системи, алгоритм перевірки моделі по суті досліджує T, щоб перевірити, чи T властивість задовольняє. Наприклад, розглянемо властивість AF g, де g - пропозиційна формула, що не містить жодних зв'язок CTL. На рис. g вірна у цьому стані. Якщо g істинно s 0, тоді ми закінчили, оскільки s 0 зустрічається на кожному шляху, що починається з s 0 . Але припустимо, що g невірно s 0 . Тоді, оскільки кожен шлях з s0 переходить або до лівого дочірнього елемента, або до

правого дочірнього елемента s0 властивість істинно s, якщо воно (рекурсивно перевірено) істинно для обох дочірніх елементів s0 T.

На Рис.3.7 показано, що g істинно корені лівого піддерева (позначено зафарбованим кружком). Отже, всі шляхи від s0 до лівого дочірнього елемента і далі вниз у лівому піддереві задовольняють цю властивість. Тепер припустимо, що g невірно в правому нащадку s 0; отже, властивість рекурсивно перевіряється всім його дочірніх елементів. На малюнку 4b показано, що g істинно для всіх дочірніх елементів правого дочірнього елемента s 0 (позначено зафарбованими кружками), і, отже, властивість для правого піддерева s 0 . Таким чином, властивість правильна для всіх піддерев'я s 0 і, отже, також правильна для s 0 .

Рисунок 3.7 підсумовує аналогічні міркування, що використовуються для перевірки властивостей, зазначених в інших формах, таких як EG g та AG g. Звичайно, на практиці алгоритми перевірки моделі набагато складніші за це; вони використовують витончені хитрощі, щоб скоротити простір станів, щоб уникнути перевірки тих частин, у яких властивість гарантовано істинно. 1, 2 Деякі хороші засоби перевірки моделей використовувалися для перевірки властивостей просторів станів розміром до 1040 станів.

В SMV властивість, що перевіряється, задається користувачем у розділі SPEC. Логічні зв'язки не, або, і має на увазі (якщо-то), представлені символом! , | , & та -> відповідно. Скроневі сполучні елементи CTL - це AF, AG, EF, EG тощо. буд. Властивість AF (pump = on) показує, що кожного шляху, що починається у початковому стані, є стан цьому шляху, у якому насос включений. Ця властивість явно помилкова у вихідному стані, оскільки існує шлях від вихідного стану, в якому насос завжди залишається вимкненим (наприклад, якщо резервуар A назавжди залишається порожнім). Якщо ця властивість зазначена у SPEC У розділі SMV створюється наступний контрприклад для властивості. Цикл вказує нескінченну послідовність станів (тобто шлях), починаючи з початкового стану, так що резервуар B заповнений у кожному стані шляху і, отже, насос вимкнений.

НУБіП України

- специфікація AF pump = on є хибною
- як демонструє наступна послідовність виконання Цикл починається тут, стан 1.1:

```
level_a = full
level_b = full
pump = off
```

Подвійна властивість AF (pump = off) вказує, що для кожного шляху, що починається в початковому стані, є стан цього шляху, при якому насос вимкнений. Ця властивість тривіально істинна у початковому стані, тому що в самому початковому стані (яке включено у всі шляхи) pump = off істинно.

Ви можете вказати цікаві та складні властивості, комбінуючи тимчасові та логічні зв'язки. Властивість AG ((pump = off) -> AF (pump = on)) стверджує, що завжди буває так, що якщо насос вимкнений, він зрештою стає включеним. Ця властивість явно неправильна у вихідному стані. Властивість AG AF (pump = off > (level_a = empty | level_b = full)) вказує, що насос завжди вимкнений, якщо нижній резервуар порожній або верхній резервуар сповнений. Властивість AG (EF (level_b = ok | level_b = full)) вказує на те, що завжди можливо досягти стану, коли верхній резервуар в порядку або повний.

ВИСНОВКИ

Комп'ютерним системам дедалі частіше призначаються завдання, що мають важливе значення для життя; їхня внутрішня складність неухильно зростає; водночас гарантувати їхню безпеку стає все важче, у той час як вони піддаються все більшій кількості загроз безпеці. Формальні методи є ключовою технологією для створення безпечних і захищених комп’ютерних систем.

Методи формальної верифікації програмного забезпечення комп'ютерних систем управління дозволяють гарантувати перевірку виконання моделлю системи властивостей, що верифікуються.

Нині ці методи активно розвиваються у напрямі зниження загальної вартості формальної перевірки, підтримки сучасних концепцій програмування та мінімізації «ручної» праці під час переходу від моделі системи до її реалізації.

Результати, отримані в магістерській роботі, є дослідження автоматизації комп'ютерних систем за допомогою формальних методів. Отримано такі теоретичні та практичні результати:

1. Проведено системний аналіз предметної області , опис предметної області, походження формальних методів, визначення поняття формальні методи . ;

2. Виявлено різницю між неформальними та формальними методами, продемонстровано взаємодію осіб формальної специфікації

3. Реалізовано перевірку моделі інструментом перевірки моделей символічної моделі (SMV) . та зроблений опис моделі SMV

Таким чином, всі поставлені в роботі задачі виконані й мета досягнута.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. G.J. Holzmann. An Analysis of Bitstate Hashing. Formal Methods in System Design, 13(3), 1998, P. 287-307. DOI: 10.1023/A:1008696026254.

2. G.J. Holzmann. The SPIN Model Checker. Primer and Reference Manual. Addison-Wesley, 2003. — 608 p.

3. R.W. Floyd. Assigning Meaning to Programs. Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics, 19, 1967. P. 19-32. DOI: 10.1090/psapm/019/0235771.

4. P. Cousot, N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. Conference Record of the 5th Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages (POPL), 1978. P. 84-97. DOI: 10.1145/512760.512770.

5. I. Burdonov, A. Kossatchev, A. Petrenko, D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. Proceedings of Formal Methods: World Congress on Formal Methods in the Development of Computing Systems (FM), LNCS 1708, Volume 1, 1999. P. 608-621. DOI: 10.1007/3-540-48119-2 34.

6. E.W. Dijkstra. Guarded Commands, Non-determinacy and Formal Derivation of Programs. Communications of ACM, 18(8), 1975. P. 453–457. DOI: 10.1145/360933.360975.

7. P. Wolper. Constructing Automata from Temporal Logic Formulas: A Tutorial. Lectures on Formal Methods and Performance Analysis, 2002. P. 261-277.

8. C.H. Papadimitriou, K. Steiglitz. Combinatorial Optimization: Algorithms and Complexity. Courier Corporation, 1982. — 496 p. (Русский перевод: Х. Пападимитриу, К. Стайглиц. Комбинаторная оптимизация. Алгоритмы и сложность. — М.: Мир, 1985. — 512 с.)

9. Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal Methods: Practice and Experience. ACM Computing Surveys, 41(4), 2009.

10. Jeannette M. Wing. A Symbiotic Relationship Between Formal Methods and Security. In Proceedings of the ONR/SNF Workshop on Computer Security, Dependability, and Assurance: From Needs to Solution, Washington DC, USA, pages 26–38, 1998. Also available as Carnegie Mellon University report CMU-CS-98-188, December 1998.

11. Ing Widya and Gert-Jan van der Heijden. Towards an Implementationoriented Specification of TP Protocol in LOTOS. In Jim Woodcock and Peter Gorm Larsen, editors, Proceedings of the 1st International Symposium of Formal Methods
Europe on Industrial-Strength Formal Methods (FME'93), Odense, Denmark, volume 670 of Lecture Notes in Computer Science, pages 93–109. Springer, 1993

НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України

ДОДАТОК А

ОПИС МОДЕЛІ SMV

НУБіП України

Сторінка - 1

```
МОДУЛЬ main VAR
   level_a: {empty, ok, full}; - нижній рівень бака
level_b: {порожній, нормально, повний}; -    насос
верхнього бака: {вкл, вимк}; ASSIGN
   наступний (level_a): = випадок       level_a =
порожньо: {порожні, ки};          level_a = ok & pump
= off: {ok, full};      level_a = ok & pump = on:
{ok, empty, full};        level_a = full & pump =
off: full;        level_a = full & pump = on: {ok,
full};      1: {нормально, порожньо, повно};
esac;
   наступний (level_b): = case       level_b =
empty & pump = off: empty;        level_b = empty &
pump = on: {empty, ok};        level_b = ok & pump
= off: {ok, empty};       level_b = ok & pump =
on: {ok, empty, full};       level_b = full & pump
= off: {ok, full};      level_b = full & pump =
on: {ok, full};      1: {нормально, порожньо,
повно};    esac;
   наступний (насос): = case
       pump = off & (level_a = ok | level_a = full) &
(level_b = empty | level_b = ok): on;      насос =
включений & (level_a = empty | level_b = full): off;
       1: насос; - зберегти статус помпи як
esac; INIT
   (pump = off)
SPEC
```


Сторінка - 2

```
 -     насос, якщо завжди вимкнений, якщо резервуар на землі
порожній або резервуар вгору повний
 -     AG AF (pump = off -> (level_a = empty | level_b =
full))
```

- завжди можна досягти стану коли верхній бак у порядку або повний

```
AG (EF (level_b = ok | level_b = full))
```

НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України