Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи автоматизації тестування та верифікації системного програмного забезпечення

(тема)

Виконав:
здобувач 2 року навчання,
групи СПм-23-4
Дмитро ВАСИЛЕНКО
(власне ім'я, прізвище)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: доц. Антон СОРОКІН
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"____" _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Василенку Дмитру Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Методи автоматизації тестування та верифікації системного програмного забезпечення

затверджена наказом по університету від "21" квітня 2025 р. № 296 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи 1. Провести дослідження процесу тестування та верифікації системного програмного забезпечення (СПЗ).
2. Розглянути сучасні інструменти автоматизації тестування та верифікації ПЗ.
3. Використання формальних та емпіричних методів
4. Системне програмне забезпечення для проведення досліджень: ОС Linux.

4. Перелік питань, що потрібно опрацювати у роботі
1. Теоретичні основи тестування та верифікації системного ПЗ (СПЗ)
2. Аналіз методів автоматизації тестування СПЗ
3. Методи формальної верифікації СПЗ
4. Проєктування системи автоматизації
5. Проведення експериментів
6. Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних Ілюстрацій

Слайд-презентація – 12 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Теоретичні основи тестування та верифікації	22.04.25-29.04.25	
	системного ПЗ (СПЗ)		
2	Аналіз методів автоматизації тестування СПЗ	30.04.25-05.05.25	
3	Методи формальної верифікації СПЗ	06.05.25-09.05.25	
4	Проєктування системи автоматизації	10.05.25-21.05.25	
5	Проведення експериментів	22.05.25-02.06.25	
6	Оформлення матеріалів кваліфікаційної роботи	03.06.25-05.06.25	
7	Подання кваліфікаційної роботи керівникові та	06.06.25-09.06.25	
	її попередній захист		
8	Подання кваліфікаційної роботи на рецензу-	10.06.25-12.06.25	
	вання		

Дата видачі завдання " 21 " квітня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Антон СОРОКІН
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 95 с., 8 рис., 11 табл., 1 дод., 73 джерела.

ВАЛІДАЦІЯ, ВЕРИФІКАЦІЯ, СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ТЕСТУВАННЯ, ФРЕЙМВОРК, CI/CD-КОНВЕЙЕР.

Метою даної кваліфікаційної роботи є розробка та вдосконалення методів автоматизації тестування і верифікації системного програмного забезпечення з використанням сучасних інструментів і практик програмної інженерії. Об’єктом дослідження є процес тестування та верифікації системного програмного забезпечення. Предметом дослідження є методи автоматизації, інструменти та підходи до підвищення ефективності тестування та верифікації СПЗ.

Наукова новизна роботи полягає у: формулюванні вдосконаленої моделі автоматизованої верифікації системного ПЗ з інтеграцією формальних методів у CI/CD-процеси; поєднанні різнорівневих методів аналізу (статичного, динамічного та формального) в єдиному фреймворку; проведенні комплексного порівняльного аналізу інструментів, орієнтованого саме на компоненти СПЗ. Практична значущість дослідження полягає в можливості: застосування результатів для автоматизованої перевірки драйверів, модулів ядра, системних служб; підвищення якості, надійності та безпеки ПЗ на ранніх етапах розробки; адаптації розроблених методів до реальних CI/CD процесів у промисловому середовищі.

ABSTRACT

Master’s thesis: 95 pages, 8 figures, 11 tables, 1 appendix, 73 sources.

FRAMEWORK, CI/CD PIPELINE, SYSTEM SOFTWARE, TESTING, VALIDATION, VERIFICATION

The objective of this qualification thesis is the development and enhancement of methods for automating the testing and verification of system software using modern tools and practices of software engineering. The object of the study is the process of testing and verification of system software. The subject of the study comprises automation methods, tools, and approaches aimed at improving the efficiency of system software testing and verification.

The novelty of this work lies in: the formulation of an improved model for automated system software verification that integrates formal methods into CI/CD processes; the combination of multi-level analysis techniques (static, dynamic, and formal) within a unified framework; the implementation of a comprehensive comparative analysis of tools, specifically focused on system software components. The practical significance of the research is reflected in the potential: to apply the results for automated validation of drivers, kernel modules, and system services; to improve the quality, reliability, and security of software at early stages of development; to adapt the proposed methods to real-world CI/CD workflows in industrial environments.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ЖЦ – життєвий цикл

ОС – операційна система

ПЗ – програмне забезпечення

СПЗ – системне програмне забезпечення

AFL – fuzz-інструмент з підтримкою coverage-guided fuzzing (англ., American Fuzzy Lop)

API – інтерфейс програмування застосунків (англ., Application Programming Interface)

CBMC – інструмент формального аналізу, що перевіряє властивості програм за допомогою техніки обмеженого моделювання (англ., C Bounded Model Checker)

CI/CD – методологія та набір практик у розробці програмного забезпечення, спрямованих на автоматизацію процесів складання, тестування та розгортання коду (англ., Continuous Integration/Continuous Delivery)

DevOps – методологія та набір практик, які об'єднують розробку програмного забезпечення (англ., Development) та операційну діяльність (англ., Operations) для прискорення процесу розробки, поліпшення якості та надійності програмних продуктів, оптимізації взаємодії між командами

DMA – прямий доступ до пам’яті (англ., Direct Memory Access)

SMT – клас алгоритмів і програм, що визначають, чи існує така інтерпретація логічних формул (на рівні арифметики, логіки масивів, покажчиків, булевих змінних), за якої ці формули істинні (англ., Satisfiability Modulo Theories)

SPIN – інструмент формальної верифікації паралельних/реактивних систем на основі model checking (англ., Simple Promela Interpreter)

V&V – верифікація та валідація (англ., Verification and validation)

ВСТУП

У сучасних обчислювальних системах системне програмне забезпечення (СПЗ) виконує критично важливу роль, забезпечуючи управління апаратними ресурсами, взаємодію з периферійними пристроями, підтримку багатозадачності, комунікацію між процесами тощо. До складу СПЗ входять операційні системи, драйвери пристроїв, компілятори, системи віртуалізації та інші компоненти, безпосередньо пов’язані з функціонуванням обчислювального середовища [1].

Вартість помилки у системному ПЗ може бути надзвичайно високою. Збій у драйвері або ядра ОС здатен призвести до повного припинення роботи системи, втрати даних або вразливостей, що ставлять під загрозу безпеку користувачів. У критичних галузях (авіації, медицині, атомній енергетиці) наявність дефектів у СПЗ може спричинити катастрофічні наслідки. Через це верифікація та тестування системного ПЗ є надзвичайно важливими етапами життєвого циклу такого програмного забезпечення.

Ручне тестування системного ПЗ, особливо на рівні ядра чи драйверів, є трудомістким, повільним і складно масштабованим процесом. Сучасні системи розробки вимагають інтеграції автоматизованих рішень, які здатні забезпечити вищу ефективність, точність і повторюваність перевірок.

Автоматизація процесів тестування та верифікації дозволяє знизити ризики помилок, зменшити витрати часу на перевірку нових версій програмного забезпечення, поліпшити якість ПЗ і забезпечити відповідність стандартам безпеки та надійності. Особливого значення набуває інтеграція таких методів у цикли CI/CD, що забезпечують безперервну інтеграцію та доставку ПЗ [2, 3].

У зв’язку з цим актуальним є питання розробки та вдосконалення методів автоматизації тестування і верифікації СПЗ, які поєднують як інструменти динамічного й статичного аналізу, так і формальні методи

перевірки коректності.

Метою даної кваліфікаційної роботи є розробка та вдосконалення методів автоматизації тестування і верифікації системного програмного забезпечення з використанням сучасних інструментів і практик програмної інженерії. Для досягнення цієї мети сформульовано такі задачі:

- провести аналіз сучасних підходів до тестування і верифікації СПЗ;
- розробити класифікацію типів системного ПЗ та відповідних методів тестування;
- побудувати модель автоматизованої верифікації з використанням формальних та емпіричних підходів;
- реалізувати програмний комплекс для автоматизованої перевірки СПЗ;
- провести експериментальне дослідження ефективності запропонованих методів.

У процесі дослідження використовувалися такі методи: формальні методи верифікації, зокрема перевірка моделей, логічні формалізації, SMT-солвери; методи статичного аналізу коду (аналіз без виконання програми); методи динамічного аналізу (тестування у виконуваному середовищі); методології CI/CD та DevOps як основа автоматизації життєвого циклу ПЗ; емпіричні методи, зокрема експериментальні дослідження інструментів на практичних прикладах.

1 ТЕОРЕТИЧНІ ОСНОВИ ТЕСТУВАННЯ ТА ВЕРИФІКАЦІЇ СИСТЕМНОГО ПЗ

1.1 Поняття та особливості системного програмного забезпечення

Системне програмне забезпечення (СПЗ) є фундаментальним компонентом будь-якої обчислювальної системи. Воно виконує функції керування апаратними ресурсами, забезпечення базових сервісів для прикладного ПЗ, а також підтримки виконання програм у середовищі операційної системи. На відміну від прикладного ПЗ, СПЗ взаємодіє безпосередньо з апаратною платформою та іншими низькорівневими компонентами, що робить його критично важливим для забезпечення стабільності та безпеки усієї системи [2].

До системного програмного забезпечення традиційно відносять:

- операційні системи (Linux, Windows, BSD, RTOS), що координують виконання програм, розподіляють ресурси та забезпечують інтерфейс між користувачем і апаратним забезпеченням;

- драйвери пристроїв, які забезпечують коректну взаємодію ОС з апаратним забезпеченням;

- системи віртуалізації та емулятори, що створюють ізольовані середовища для виконання додатків;

- компілятори, лінкери, інтерпретатори, які забезпечують перетворення коду в машинну форму;

- системні бібліотеки, що надають базові функції (вводу/виводу, пам’яті, математичних обчислень) [3].

СПЗ має низку ключових особливостей, що визначають специфіку його розробки, тестування та верифікації:

- прив’язаність до конкретної апаратної архітектури. Більшість компонентів системного ПЗ повинні враховувати особливості процесорів,

чипсетів, контролерів тощо;

- високий ступінь відповідальності за стабільність системи. Помилки у СПЗ часто призводять до збоїв, втрати даних або некоректної роботи системи в цілому;

- підвищені вимоги до продуктивності. СПЗ має бути оптимізованим, оскільки воно працює на критичних шляхах виконання;

- складність тестування та діагностики. Через низький рівень абстракції спостережуваність внутрішніх процесів обмежена, а засоби налагодження можуть впливати на поведінку ПЗ [4].

Враховуючи ці чинники, тестування СПЗ вимагає спеціалізованих підходів, які враховують тісний зв’язок з апаратним забезпеченням, обмежену спостережуваність та складність створення ізольованих середовищ виконання. Саме ці особливості зумовлюють потребу у формалізованих, автоматизованих та надійних методах верифікації.

1.2 Моделі життєвого циклу системного програмного забезпечення

Життєвий цикл програмного забезпечення (ЖЦ ПЗ) – це сукупність етапів, які охоплюють увесь процес його створення, впровадження, експлуатації та виведення з експлуатації. У випадку системного програмного забезпечення (СПЗ), вибір і реалізація моделі ЖЦ мають принципове значення, оскільки будь-яка помилка на критичних етапах розробки може призвести до глобальних системних збоїв, зниження безпеки або недопустимого падіння продуктивності системи [5].

СПЗ має властивість високої складності, тісної інтеграції з апаратними компонентами та, у багатьох випадках, відповідальності за дотримання вимог жорстких стандартів (наприклад, DO-178C, ISO/IEC 15408). Це зумовлює підвищені вимоги до формалізації, верифікації та документування життєвого циклу.

1.2.1 Класичні моделі ЖЦ ПЗ

Каскадна модель (Waterfall) передбачає лінійну послідовність етапів: аналіз вимог → проєктування → реалізація → тестування → впровадження → супровід. Її перевага – у формалізованості та чіткій структурі. У розробці СПЗ каскадна модель використовується у проектах із жорсткими регуляторними вимогами, особливо коли необхідна ретельна верифікація на кожному етапі.

Основні недоліки каскадного підходу:

- низька гнучкість до змін;
- висока вартість помилок, виявлених на пізніх стадіях;
- затримка зворотного зв’язку між етапами.

Проте в розробці операційних систем для критичних застосувань (RTOS, авіаційне ПЗ, військові системи) цей підхід досі є стандартом [6].

V-подібна модель (V-Model) доповнює каскадну, додаючи відповідні етапи тестування для кожної фази розробки:

- аналіз вимог → перевірка прийнятності;
- архітектурне проєктування → інтеграційне тестування;
- проєктування компонентів → модульне тестування.

Перевага цього підходу в структурованому тестуванні та верифікації, що критично важливо для СПЗ, особливо драйверів, ядра, завантажувачів, тощо. Його часто поєднують із методами формальної верифікації, що підвищує впевненість у коректності реалізації [7].

Інкрементна та ітеративна моделі – у таких моделях розробка відбувається частинами (інкрементами), що дозволяє поступово додавати функціональність. Цей підхід корисний у великих проєктах СПЗ, наприклад, при створенні модулів ядра Linux, де кожен модуль реалізується, перевіряється і інтегрується окремо. Ітеративність дозволяє швидше виявляти недоліки дизайну, проте потребує ретельного регресійного тестування [5].

1.2.2 Гнучкі моделі та практики

Agile-підходи (Scrum, XP). Попри популярність у бізнес-орієнтованих застосунках, Agile-підходи застосовуються до СПЗ обмежено. Їх використовують переважно в частинах, де:

- є змінні або нечіткі вимоги (наприклад, утиліти ОС або системні API);
- потрібна тісна взаємодія з клієнтом;
- формальна сертифікація не є обов'язковою.

Проте для критичних частин (ядро, low-level драйвери) ці моделі не підходять без серйозних модифікацій через відсутність гарантій надійності та відповідності вимогам безпеки [3].

DevOps та CI/CD. DevOps як філософія інтегрує розробку (Dev) і операційне забезпечення (Ops), акцентуючи увагу на автоматизації, безперервній інтеграції (CI), тестуванні та доставці (CD). У проєктах СПЗ це дозволяє:

- швидко виявляти регресії;
- запускати автоматизовані тести при кожному коміті;
- інтегрувати статичний і динамічний аналіз;
- використовувати емулятори (QEMU), інструменти fuzzing, SMT-солвери у пайплайнах [8].

DevOps-підходи використовуються в багатьох відкритих системах (наприклад, розробка ядра Linux, систем Init, драйверів для FreeBSD тощо). Вони сумісні з формалізованими фазами розробки, якщо інтегровані з відповідними контрольними точками.

1.2.3 Гібридні та доменні моделі

У складних проєктах СПЗ застосовують гібридні моделі, що поєднують:

- строгість V-моделі для ядра, драйверів;
- гнучкість інкрементної моделі для сервісів;
- DevOps-практики для безперервного забезпечення якості.

Також слід зазначити, що розробка систем із сертифікаційними вимогами (наприклад, DO-178C, ISO 26262, IEC 61508) вимагає специфічних моделей, які передбачають:

- формалізацію вимог,
- трасування артефактів,
- перевірку зворотного зв’язку між реалізацією та вимогами,
- сувору документацію процесів.

1.3 Основи тестування програмного забезпечення: типи, рівні, критерії

Процес тестування є однією з центральних складових забезпечення якості програмного забезпечення, а у випадку системного ПЗ – ще й засобом гарантування його безпечного функціонування в умовах безпосередньої взаємодії з апаратною платформою. У сучасному програмному інжинірингу тестування розглядається не лише як інструмент виявлення помилок, але і як ключовий елемент верифікації, валідації, забезпечення відповідності стандартам і безперервного контролю якості протягом усього життєвого циклу ПЗ [9, 2].

Системне програмне забезпечення має низку особливостей, що ускладнюють тестування: обмежена спостережуваність виконання (особливо в режимі ядра), відсутність повноцінного ізольованого середовища, тісний зв’язок з апаратним забезпеченням, висока відповідальність за стабільність усієї системи. Через це застосування традиційних підходів до тестування вимагає модифікацій, а ефективне тестування потребує поглибленого розуміння типів і рівнів тестування, які можуть бути адаптовані до особливостей СПЗ.

1.3.1 Типологія тестування: функціональний та нефункціональний виміри

У найзагальнішому вигляді тестування поділяється на функціональне та нефункціональне. Перше має на меті перевірку того, наскільки програмне забезпечення виконує визначені специфікації та вимоги користувача. У контексті СПЗ, наприклад, це може означати, чи драйвер коректно обробляє запити пристрою відповідно до протоколу. Водночас нефункціональне тестування охоплює ширший спектр характеристик: продуктивність, масштабованість, стійкість до помилок, сумісність з різними конфігураціями обладнання, енергоспоживання тощо [10]. Взаємозв'язок типів, рівнів і методів тестування схематично проілюстрований на рисунку 1.1.

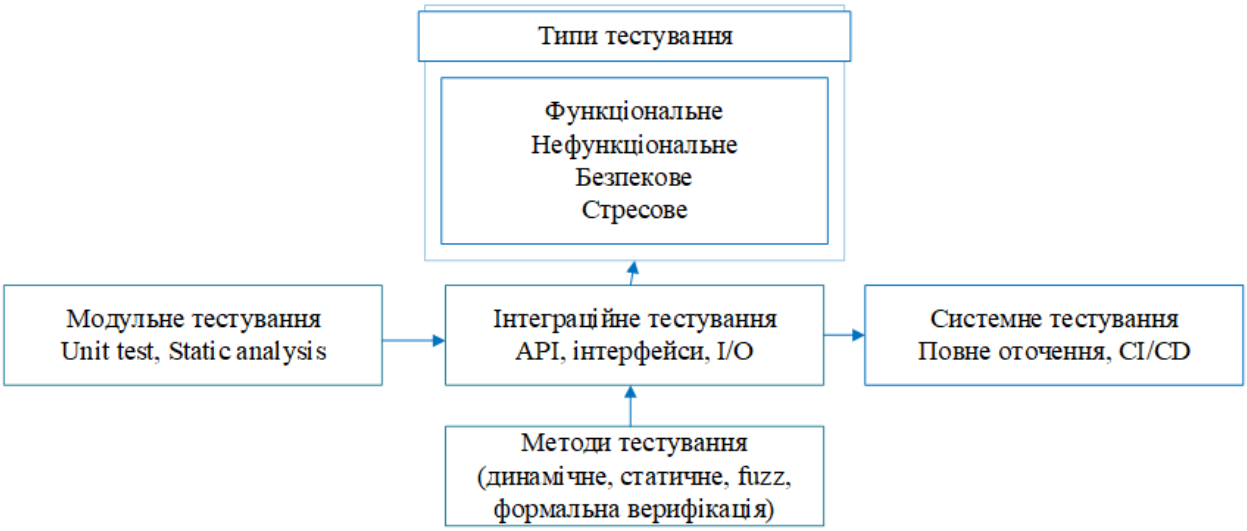

Рисунок 1.1 – Узагальнена модель тестування СПЗ

Найбільш складними для реалізації у сфері СПЗ є тести на витривалість, надійність та безпеку. Програмні помилки у компонентах, що працюють із привілеями ядра, здатні викликати критичні збої всієї операційної системи, тому їх виявлення має особливу пріоритетність. Методи, що застосовуються для цього, варіюються від детермінованого функціонального тестування до стохастичного fuzz-тестування, яке дозволяє

виявити аномальні поведінкові патерни при неочікуваному або некоректному вхідному сигналі [11].

1.3.2 Рівні тестування та їх реалізація в системному ПЗ

Тестування в системному програмуванні організовується за кількома рівнями, що відповідають структурі та ієрархії програмного коду. Загальна структура рівнів тестування в системному ПЗ наведена на рисунку 1.2.

Модульне тестування, що застосовується для перевірки окремих функцій або об’єктів, у СПЗ часто реалізується з використанням спеціальних фреймворків (наприклад, KUnit у Linux). Цей рівень тестування дозволяє виявити локалізовані помилки, зменшити складність діагностики та спростити відлагодження.

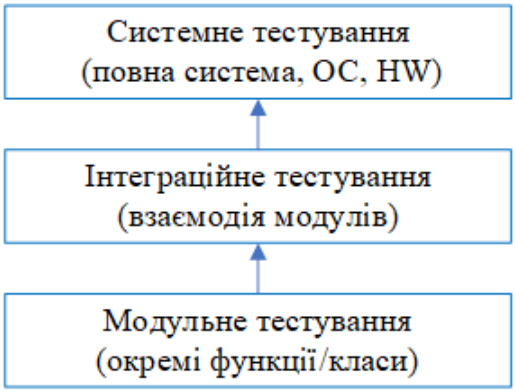

Рисунок 1.2 – Ієрархія рівнів тестування системного ПЗ

Наступним етапом є інтеграційне тестування, що передбачає перевірку взаємодії між модулями системного ПЗ. У випадку драйверів пристроїв, це може стосуватися їхньої взаємодії з ядром ОС, стеком введення-виведення, управлінням енергоспоживанням або підсистемою переривань. Зважаючи на залежність результатів інтеграційного тестування від апаратної конфігурації, часто застосовують засоби емуляції (наприклад, QEMU) або віртуалізовані середовища, де можлива реплікація апаратної поведінки [12].

Системне тестування охоплює перевірку ПЗ в умовах максимально

наближених до реального середовища експлуатації. Воно включає повну конфігурацію апаратного забезпечення, операційної системи, а також реальні сценарії використання. У системному ПЗ системне тестування часто виконується з використанням повноцінного CI/CD-пайплайну, що автоматизує побудову, запуск, тестування та аналіз результатів (наприклад, KernelCI для Linux) [13].

Окремо слід згадати приймальне тестування, яке, попри свою асоціацію з прикладним ПЗ, також використовується у системному програмуванні – наприклад, при виведенні нової версії мікропрограми або дистрибутиву ядра в промислове середовище.

1.3.3 Критерії ефективності та завершення тестування

Оцінювання ефективності тестування та визначення моменту його завершення є проблемою, що потребує формального підходу. Одним з найпоширеніших критеріїв є покриття коду – показник того, яка частина вихідного коду була виконана під час проходження тестів. У системному ПЗ застосовується як покриття інструкцій, так і покриття гілок, умов та шляхів, із використанням інструментів, таких як GCOV, LCOV, Kcov тощо [14].

Крім цього, до ключових критеріїв належать відсутність невиправлених дефектів високої критичності, відповідність усім специфікованим вимогам, стабільність результатів при повторному виконанні тестів, відповідність встановленим часовим або ресурсним обмеженням.

Питання завершення тестування також залежить від контексту застосування ПЗ. Наприклад, у ПЗ, що підлягає сертифікації за стандартами безпеки, критерієм завершення може бути успішне проходження всіх верифікаційних сценаріїв згідно з валідаційним планом, узгодженим із наглядовими органами (наприклад, FAA, ENISA або TÜV SÜD) [15].

1.3.4 Автоматизація як основа сучасного тестування

Автоматизоване тестування поступово перетворилося на центральний інструмент забезпечення якості СПЗ. Це пов’язано як із потребою в багаторазовому повторенні ідентичних тестів при кожній зміні коду (особливо в open-source проектах), так і з високими вимогами до швидкості зворотного зв’язку. Відомими прикладами автоматизованих інфраструктур є CI-платформи KernelCI, GitLab CI, Jenkins, які інтегруються з емуляторами, віртуальними машинами, а також системами збирання метрик. Спрощена схема, яка пояснює автоматизацію тестування в CI/CD, наведена на рисунку 1.3.

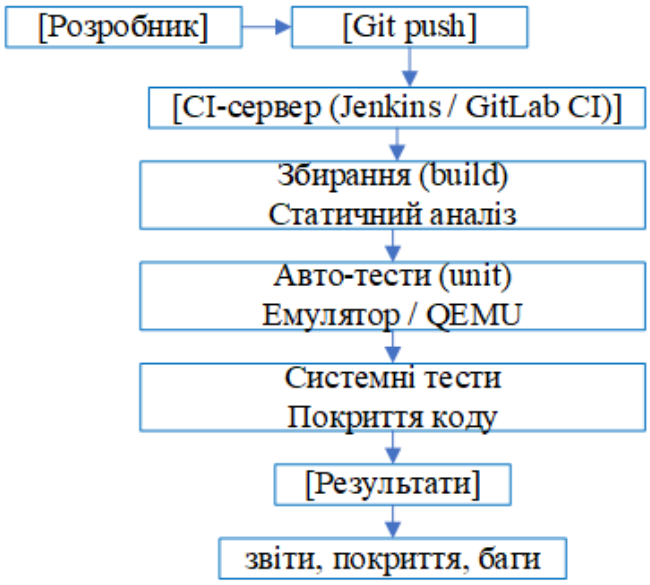

Рисунок 1.3 – Вбудовування тестування в DevOps-пайплайн

Інтеграція автоматичного тестування із засобами статичного аналізу (наприклад, Clang Static Analyzer, Coverity) і формального доведення властивостей (CBMC, Frama-C) дозволяє досягти вищого рівня впевненості у коректності системного ПЗ ще до його фактичного виконання на цільовій платформі. Таким чином, автоматизація не лише підвищує ефективність процесу, але й розширює його глибину та охоплення [3].

1.4 Верифікація та валідація: відмінності, методи

У практиці інженерії програмного забезпечення терміни верифікація та валідація часто вживаються спільно як єдина складова забезпечення якості. Проте, згідно з міжнародними стандартами, ці поняття мають чітке розмежування як за змістом, так і за методами реалізації. У системному програмному забезпеченні, де наслідки помилок можуть бути катастрофічними, чітке розуміння й правильне застосування верифікації та валідації є критично важливими [4, 16].

Верифікація (від лат. verificare – підтверджувати істинність) – це процес підтвердження того, що програмне забезпечення відповідає встановленим технічним специфікаціям. Це, по суті, порівняння артефактів проєкту (вимог, дизайну, коду) з формалізованими вимогами до них.

Верифікація найчастіше здійснюється без виконання ПЗ, тобто на стадії статичного аналізу. До основних методів верифікації (рисунок 1.4) належать:

- огляд та рецензування коду (code review), що дозволяє виявити стилістичні, логічні й архітектурні недоліки;
- статичний аналіз, який здійснюється за допомогою спеціалізованих інструментів без запуску програми. Інструменти на кшталт Clang Static Analyzer, Coverity або Frama-C здатні виявляти помилки доступу до пам’яті, порушення умов, невизначену поведінку тощо;
- формальні методи – використання математичних моделей для доведення коректності програм (детальніше розглядаються у розділі 3);
- моделювання (Model Checking) – перевірка виконання всіх можливих сценаріїв поведінки системи на відповідність формальній специфікації.

У системному ПЗ верифікація має особливе значення, оскільки більшість помилок може не проявитися одразу під час виконання, а бути латентними (прихованими) і спричинити збій лише в специфічних умовах [17].

Валідація – це процес перевірки того, чи програмне забезпечення

відповідає очікуванням користувача, потребам, функціональному призначенню. Вона має емпіричний характер і проводиться шляхом виконання ПЗ в умовах, максимально наближених до експлуатаційного середовища.

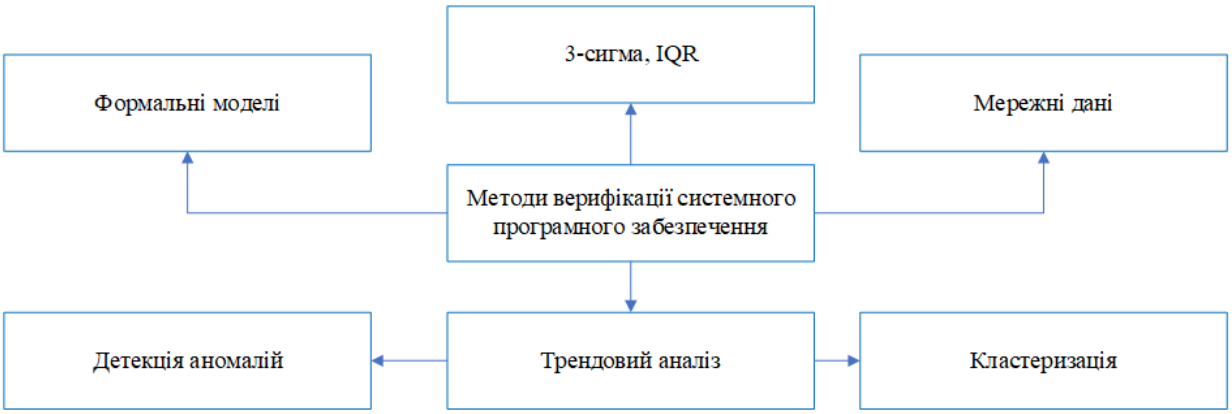

Рисунок 1.4 – Верифікація системного ПЗ

До методів валідації належать:

- функціональне тестування, що перевіряє правильність реалізації функцій відповідно до вимог;

- системне тестування, яке охоплює всі компоненти у взаємодії на реальному або емуляторному обладнанні;

- тестування продуктивності, що дає змогу оцінити, чи відповідає ПЗ заданим критеріям швидкодії, енергоефективності, масштабованості;

- тестування безпеки, яке верифікує стійкість до атак, перевірку доступу, коректну ізоляцію ресурсів.

У контексті СПЗ, валідація часто реалізується через автоматизовані сценарії, що включають реальне використання функцій драйверів, підсистем ядра, сервісів. Наприклад, у Linux такі перевірки можуть виконуватись у межах інфраструктури KernelCI [18].

У системному програмуванні верифікація і валідація не є взаємовиключними, а навпаки – доповнюють одна одну. Практика «shift left testing» передбачає перенесення як верифікаційних, так і валідаційних

активностей на більш ранні стадії розробки. Наприклад, уже під час написання коду можуть запускатися лінтери, компілятори з увімкненими перевірками, статичні аналізатори, а також юніт-тести.

Таблиця 1.1 – Порівняння верифікації і валідації

Критерій	Верифікація	Валідація
Питання	Чи правильно реалізовано?	Чи те реалізовано, що потрібно?
Орієнтація	На відповідність специфікаціям	На відповідність очікуванням/потребам
Методи	Статичний аналіз, формальні методи	Тестування, спостереження
Форма	Теоретична, логічна перевірка	Практичне випробування
Приклади інструментів	Frama-C, CBMC, SPIN	QEMU, Jenkins CI, AFL, KUnit

Також слід відзначити, що у високонадійних системах (авіоніка, автомобільна електроніка, медичне обладнання) верифікація і валідація є предметом формальної сертифікації, і тому вони виконуються згідно з регламентами таких стандартів, як DO-178C, ISO 26262, IEC 61508 [15].

1.5 Проблеми тестування системного програмного забезпечення

Тестування системного програмного забезпечення (СПЗ) суттєво відрізняється від тестування прикладних програм через низку фундаментальних технічних і методологічних складнощів (рисунок 1.5). Хоча принципи якості, верифікації та валідації є спільними для всіх видів програмного забезпечення, саме у СПЗ спостерігається найбільша щільність взаємозв’язку між кодом і фізичним апаратним середовищем, найвищі

вимоги до надійності, а також обмежена можливість створення контрольованого середовища виконання.

Ключові виклики, які значно ускладнюють процеси тестування у цьому класі програмних систем:

- низька спостережуваність;
- прив’язка до апаратного середовища;
- обмеження на втручання в середовище виконання;
- складність відтворення дефектів;
- відсутність референсної поведінки.

Однією з головних проблем тестування СПЗ є обмежена можливість спостереження за його внутрішнім станом під час виконання. Це особливо характерно для компонентів, що працюють у привілейованому режимі (наприклад, ядро операційної системи, драйвери пристроїв, мікропрограмне забезпечення). Звичайні засоби трасування або відлагодження не можуть бути використані без змін у середовищі виконання, що потенційно змінює поведінку системи – відомий ефект Heisenbug [19, 20].

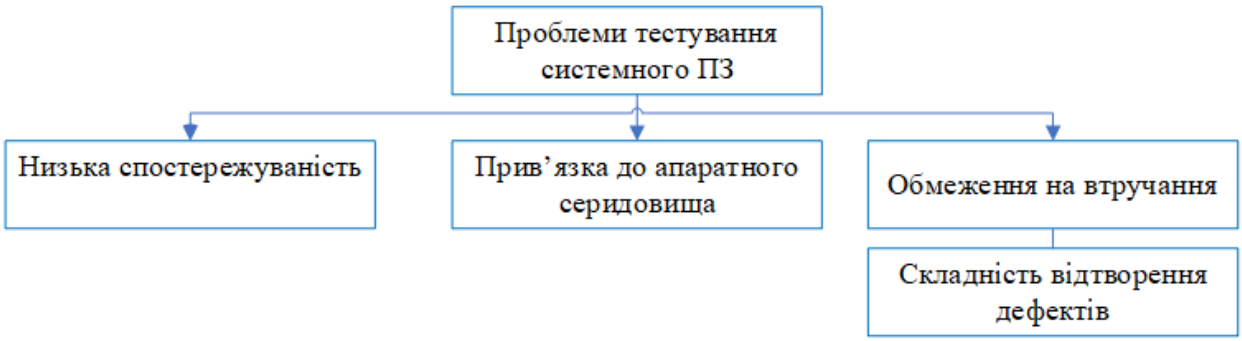

Рисунок 1.5 – Особливості тестування системного ПЗ

У деяких випадках розробники вимушені використовувати емуляцію або спеціалізовані апаратні трасувальні інтерфейси (JTAG, Intel PT, ARM CoreSight), щоб отримати уявлення про внутрішні процеси, що відбуваються під час виконання. Проте такі методи суттєво підвищують складність і вартість тестування.

Системне програмне забезпечення майже завжди прив’язане до конкретної архітектури, платформи або класу пристроїв. Навіть незначні зміни у конфігурації апаратного середовища – тип процесора, об’єм оперативної пам’яті, модель контролера введення/виведення – можуть призводити до появи нових дефектів. Це унеможливлює застосування повністю ізольованого або абстрактного середовища тестування без ризику втрати релевантності результатів [21].

Найбільш поширені наслідки цієї прив’язки:

- відсутність універсальних тестових стендів;
- необхідність підтримки великої кількості конфігурацій;
- труднощі автоматизації тестування через потребу фізичного доступу до обладнання;
- залежність від нестабільного або неуніфікованого апаратного інтерфейсу.

Для часткового вирішення цієї проблеми використовуються віртуалізовані середовища, такі як QEMU, KVM, VirtualBox, які дозволяють емулювати апаратну архітектуру, однак вони не завжди забезпечують точну модель низькорівневої поведінки [12].

На відміну від прикладного ПЗ, яке можна тестувати в ізоляції, СПЗ часто є частиною цілісної, взаємозалежної системи. Це накладає обмеження на можливість втручання в процес виконання. Наприклад, інжекція помилок, модифікація системних викликів або використання дебагерів можуть змінити поведінку системи і спровокувати помилкові результати або навіть збої [22].

Особливо це стосується драйверів пристроїв і компонентів ядра ОС. Втручання в код таких модулів (наприклад, шляхом додавання логів) може порушити таймінги, впливати на керування ресурсами або спричинити інші побічні ефекти.

Ці обмеження зумовлюють потребу в пасивному моніторингу, детермінованому повторному виконанні (record & replay), а також формальному аналізі, який дозволяє моделювати поведінку системи без

фізичного виконання [19].

Багато помилок у СПЗ мають недетерміновану природу, тобто проявляються лише за певних умов, наприклад, при специфічному порядку доступу до ресурсів, збігу апаратних подій або взаємодії з іншими модулями. Це ускладнює не лише їхнє виявлення, але й відтворення для аналізу та виправлення.

Традиційні підходи до налагодження – встановлення точок зупину, аналіз дампів пам’яті – не завжди ефективні. У відповідь на це з’являються методи детермінованого виконання, які забезпечують однаковий хід подій при кожному запуску, зокрема у проектах наприклад rr або PANDA. Проте їх застосування у реальних СПЗ потребує суттєвої модифікації коду і часто супроводжується зниженням продуктивності [23].

На відміну від багатьох прикладних програм, для системного ПЗ зазвичай не існує єдиного джерела істини, з яким можна порівнювати поведінку програми. Розробник не має чіткого "еталону", що ускладнює валідацію результатів. Наприклад, при розробці нового драйвера пристрою поведінка може змінюватися в залежності від версії прошивки, специфікацій виробника, режиму енергозбереження тощо.

Це спонукає до використання програмних інтерфейсів контрактного типу, assert-моделей, специфікацій поведінки у вигляді формальних моделей або застосування вимірювальних тестів (наприклад, продуктивності або затримки), навіть якщо повна функціональна перевірка неможлива [3].

1.6 Огляд стандартів

У галузі системного програмного забезпечення особливу роль відіграє дотримання міжнародних стандартів, які встановлюють вимоги до процесів тестування, верифікації, валідації, забезпечення якості та надійності. На відміну від прикладного ПЗ, де такі стандарти можуть бути орієнтовними, у системному контексті – особливо в критичних галузях (авіація,

автомобілебудування, атомна енергетика, медицина) – відповідність встановленим нормам є обов'язковою умовою сертифікації та введення в експлуатацію [10].

Міжнародний стандарт ISO/IEC/IEEE 29119 є універсальним фреймворком, що охоплює всі аспекти процесу тестування ПЗ – від планування до виконання та звітності. Він складається з п'яти частин:

- 29119-1: терміни й поняття;
- 29119-2: процеси тестування;
- 29119-3: документація;
- 29119-4: методи тест-дизайну;
- 29119-5: тестування на основі ризиків [24].

Цей стандарт не є специфічним для системного ПЗ, проте його процесна структура широко використовується у побудові політик тестування драйверів, операційних систем, BIOS/UEFI. Наприклад, вимагається явна декомпозиція тестових цілей, формалізоване відстеження вимог, формування матриці відповідності «вимога → тест».

IEEE Std 1012-2016 визначає процеси верифікації та валідації (V&V) у рамках життєвого циклу ПЗ. Особливу увагу він приділяє визначенню ступеня критичності системи (software integrity level) та відповідній інтенсивності перевірок. Це дозволяє масштабувати V&V-процеси – від базових перевірок для звичайного ПЗ до повноцінної формальної верифікації в авіоніці або автомобільній електроніці [4].

У сфері системного ПЗ IEEE 1012 часто застосовується в поєднанні з ISO 12207 (життєвий цикл ПЗ), зокрема для проектів, що розробляються в рамках державних оборонних програм.

RTCA DO-178C є стандартом для розробки й сертифікації авіаційного програмного забезпечення. Він установлює п'ять рівнів критичності, від A (відмова призводить до катастрофи) до E (без впливу на безпеку), кожному з яких відповідає різна глибина перевірок. Для рівнів A–C вимагається:

- 100% покриття інструкцій і гілок;

- трасування вимог до тестів;
- формальні методи верифікації (опціонально, але рекомендовано) [15].

Системне ПЗ у авіації – наприклад, мікроядра реального часу, компоненти вбудованих ОС – не може бути допущене до експлуатації без сертифікації за DO-178C, що обумовлює жорсткі вимоги до формальності, автоматизації та прозорості тестування.

Стандарт MISRA C, розроблений для підвищення безпеки й надійності програм, написаних мовою C, особливо у вбудованих системах. Він містить набір рекомендацій і обмежень щодо синтаксису, стилю та структури коду.

Для системного ПЗ, написаного на C (ядро ОС, драйвери, системні бібліотеки), дотримання MISRA C дозволяє:

- уникати невизначеної поведінки;
- зменшити ризики, пов’язані з управлінням пам’яттю;
- забезпечити легшу підтримку та аудит коду [20].

Таблиця 1.2 – Застосування стандартів у СПЗ

Стандарт	Основна область застосування	Специфічність до СПЗ	Ключові вимоги
ISO/IEC/IEEE 29119	Універсальний	Часткова	Формалізовані процеси тестування
IEEE 1012	V&V у ПЗ	Так	Верифікація на всіх фазах ЖЦ
DO-178C	Авіаційне СПЗ	Так	Сертифікація, формальні методи
MISRA C	Вбудовані системи (C)	Так	Кодекс правил, синтаксичні обмеження
ISO 25010	Оцінка якості	Ні	Метрики якості ПЗ

MISRA активно застосовується в автомобільній, медичній, оборонній

промисловості, де часто інтегрується з інструментами автоматичного аналізу, такими як Coverity, Astrée або Polyspace.

Інші релевантні стандарти:

- ISO/IEC 25010 – описує модель якості ПЗ, включаючи такі характеристики, як надійність, безпека, зручність у використанні, продуктивність;
- IEC 61508 – базовий стандарт функціональної безпеки електронних систем, широко використовується в поєднанні з ISO 26262 (автомобільна галузь);
- ISO/IEC 17025 – акредитація лабораторій тестування (наприклад, при незалежній перевірці СПЗ);
- CERT C/C++ Coding Standards – рекомендації щодо безпечного програмування, з особливим акцентом на запобігання вразливостям [25].

1.7 Висновки за розділом

Для розробки сучасного системного ПЗ необхідно враховувати не лише вибір моделі ЖЦ, а й можливість інтеграції автоматизованих інструментів тестування та верифікації. Поєднання структурованих моделей (V-модель) з DevOps-практиками створює підґрунтя для безпечної, ефективної та контрольованої розробки СПЗ, що дозволяє знизити ризики на всіх етапах життєвого циклу.

Верифікація та валідація є взаємодоповнювальними підходами до забезпечення якості програмного забезпечення, що мають різні методологічні засади та цілі. У контексті системного ПЗ їх правильне поєднання дозволяє досягти високого рівня впевненості в коректності, надійності та відповідності ПЗ поставленим завданням. Автоматизація обох процесів, інтеграція в CI/CD-пайплайни, використання формальних моделей та інструментів тестування створюють нову якість розробки, що особливо важливо для критичних застосувань.

Проблеми тестування системного програмного забезпечення пов’язані не лише зі складністю самого ПЗ, а й із глибокою інтеграцією в апаратне середовище, обмеженнями спостережуваності та контрольованості. Ці фактори обумовлюють потребу в гібридному підході до забезпечення якості, що поєднує класичне функціональне тестування, автоматизацію, емуляцію та формальні методи. Комплексне подолання викликів можливе лише за умови інтеграції тестування в усі етапи життєвого циклу СПЗ, починаючи з фази проєктування.

У системному програмуванні застосування міжнародних стандартів тестування є не просто рекомендацією, а обов’язковою умовою розробки якісного, безпечного й сертифікованого програмного забезпечення. Знання та правильне застосування таких стандартів, як ISO/IEC/IEEE 29119, DO-178C, MISRA C, забезпечує не лише формальну відповідність, але й структурованість і доказову надійність тестування. Саме ці підходи стають основою розробки та впровадження автоматизованих, формалізованих фреймворків для перевірки системного ПЗ.

2 АНАЛІЗ МЕТОДІВ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ СИСТЕМНОГО ПЗ

2.1 Класифікація методів автоматизованого тестування

Автоматизоване тестування – один із найважливіших напрямів забезпечення якості системного програмного забезпечення (СПЗ), що дозволяє ефективно протидіяти складності, обсягам і швидкості змін, характерним для низькорівневих систем. Його головною перевагою є здатність зменшити людський фактор, забезпечити повторюваність тестів, підвищити масштабованість перевірок і виявляти дефекти на ранніх стадіях життєвого циклу ПЗ [26].

У сучасній практиці існує велика кількість підходів до класифікації методів автоматизованого тестування. Найбільш релевантною для системного ПЗ є багатовимірна класифікація за критерієм доступності до внутрішньої структури коду, типом досліджуваного об’єкта, етапом життєвого циклу, а також способом генерування та виконання тестів.

За ступенем доступу до коду, у системному ПЗ методи тестування умовно поділяються на такі:

- методи «чорного ящика» (Black Box Testing) ґрунтуються на аналізі зовнішньої поведінки без знання внутрішньої структури. Вони ефективні для перевірки контрактів драйверів, системних API, конфігураційних інтерфейсів;
- методи «білого ящика» (White Box Testing) оперують внутрішньою логікою ПЗ. До них належать структурне тестування, перевірка покриття коду, тестування гілок і умов;
- методи «сірого ящика» (Grey Box Testing) комбінують обидва підходи, дозволяючи аналізувати об'єкт частково зі знанням структури, але не на рівні повного вихідного коду. Наприклад, верифікація модулів ядра ОС, де структура доступна, але взаємодія з іншими модулями – ні [2].

За рівнем абстракції об'єкта тестування методи поділяються на такі, що орієнтовані на:

- модулі (unit testing) – ізольована перевірка невеликих компонентів, як-от функції обробки переривань;

- інтеграційні взаємодії (integration testing) – перевірка взаємодії драйверів з ядром;

- повну систему (system testing) – тестування всієї операційної системи або вбудованої платформи;

- взаємодію з апаратним середовищем (hardware-in-the-loop, HIL) – емуляція або тестування реального заліза [27].

Методи можна класифікувати за принципом побудови тестових сценаріїв:

- скриптові методи – написання жорстко закодованих сценаріїв на мові shell, Python, CUnit тощо. Використовуються для регресійного тестування;

- генеративні методи – автоматична генерація тестів на основі специфікацій, моделей або випадкових даних. Наприклад, fuzz-тестування, symbolic execution, random test generation;

- модельно-орієнтовані методи (Model-Based Testing) – формальна побудова моделі поведінки системи і автоматичне виведення тестових випадків. Використовується для протоколів, станів пристроїв, логіки енергозбереження [28].

За часом запуску (до, під час або після складання):

- pre-build тести: перевірка конфігурацій, аналіз залежностей;

- build-time тести: інтегровані в систему збирання (наприклад, Makefile, CMake);

- post-build тести: юніт-тести, запуск у sandbox-середовищах;

- runtime тести: виконуються в контексті ядра або після ініціалізації системи, з можливістю збору телеметрії, логів тощо.

За ступенем формальності в системному ПЗ виділяють:

- імперативні методи тестування, де розробник явно задає тестові сценарії;

- формальні методи верифікації, які базуються на логіці, автоматах, теорії моделей. Ці методи не лише тестують, а й доводять коректність (див. розділ 3);

- гібридні методи, які поєднують тестування та аналіз (наприклад, CBMC, що використовує bounded model checking для генерації та перевірки сценаріїв).

2.2 Методи чорного, білого і сірого ящика у тестуванні системного ПЗ

Класифікація методів тестування за принципом доступу до внутрішньої структури програми є фундаментальною як для прикладного, так і для системного програмного забезпечення. Проте в контексті СПЗ ця класифікація набуває особливого значення – не лише через складність коду, але й через обмеження спостереження, специфіку виконання з привілеями ядра, а також тісну інтеграцію з апаратним середовищем. Розглянемо кожен із трьох підходів (black-box, white-box та grey-box) у контексті системного програмування.

Методи тестування «чорного ящика» (Black Box Testing) орієнтовані на перевірку функціональної поведінки програми без доступу до її внутрішньої структури. Вхідні дані подаються на інтерфейси ПЗ, а результати порівнюються з очікуваними виходами.

У системному ПЗ black-box тести часто застосовуються:

- для перевірки API драйверів пристроїв: наприклад, введення/виведення в /dev, перевірка ioctl-запитів;

- у тестуванні конфігурацій ядра через інтерфейси типу /proc, /sys;

- при валідації завантажувачів, мікропрограм, де структура коду не доступна (closed source) [29].

Основною перевагою методу є незалежність від реалізації, що дозволяє

протестувати поведінку модуля в цілому. Проте цей підхід має суттєві обмеження:

- низьке покриття внутрішніх гілок логіки;
- нездатність виявити приховані помилки, які не проявляються на зовнішньому рівні;
- складність діагностики причин відмов [9].

Таблиця 2.1 – Порівняння методів тестування за принципом доступу до внутрішньої структури програми

Ознака	Black Box	White Box	Grey Box
Знання коду	Відсутнє	Повне	Часткове
Приклади у СПЗ	IOCTL, dev-тести	KUnit, static tools	e2e з логами, strace
Глибина покриття	Низька	Висока	Середня
Складність реалізації	Низька	Висока	Середня
Використання в CI/CD	Легко інтегрується	Потребує емул.	Частково автоматизується

Тестування «білого ящика» (White Box Testing, структурне тестування) базується на повному знанні внутрішньої логіки, структури та реалізації програмного модуля. У випадку СПЗ це означає:

- тестування окремих функцій ядра (наприклад, page_fault_handler, schedule() у Linux);
- аналіз логіки обробки апаратних переривань у драйверах;
- перевірка граничних умов в підсистемах пам’яті, таймерів, синхронізації [30].

Інструменти white-box тестування включають:

- інструменти вимірювання покриття коду (Gcov, Kcov);
- аналітичні фреймворки (Clang Static Analyzer, Frama-C);

- фреймворки модульного тестування (KUnit для Linux kernel, GoogleTest для C++) [31].

Основна перевага цього підходу – глибоке охоплення коду, включаючи рідкісні або критичні шляхи. Проте в системному контексті існують труднощі:

- потрібна модифікація середовища виконання або використання емуляторів;
- ризик впливу на часові характеристики або структуру коду;
- складність розгортання тестів у CI/CD без емуляції середовища [32].

Методи тестування «сірого ящика» (Grey Box Testing) поєднують зовнішню взаємодію з програмним продуктом і часткове знання внутрішньої структури. Це особливо корисно для системного ПЗ, де повний доступ до коду обмежений (наприклад, через NDA) або код є надто складним для повного охоплення.

Приклади застосування:

- тестування поведінки мережевих стеків із доступом до внутрішньої таблиці маршрутів;
- перевірка енергозбереження з аналізом системних викликів, журналів живлення;
- тестування модулів ядра, доступних у вигляді вихідного коду, але що залежать від складної взаємодії з закритими прошивками [20].

Grey-box тести часто реалізуються як інтеграційні або end-to-end сценарії з аналізом логів, системних викликів (наприклад, через strace, perf, LTTng), а також з використанням емуляторів та віртуальних машин для контролю стану системи.

2.3 Використання емуляторів і віртуалізації в тестуванні системного програмного забезпечення

Емуляція та віртуалізація стали невід’ємними інструментами в арсеналі

засобів автоматизованого тестування системного програмного забезпечення (СПЗ), особливо у випадках, коли тестування на реальному апаратному забезпеченні є складним, дорогим або навіть фізично недоступним. Ці технології дозволяють створювати ізольовані, контрольовані середовища, в яких можливо відтворити умови виконання, наближені до реальних, без ризику пошкодити обладнання або систему [33].

Емуляція полягає у повному відтворенні поведінки апаратного середовища шляхом програмного моделювання його архітектури, інтерфейсів та пристроїв. Найпоширенішим прикладом є QEMU, який здатен емуляторно запускати ПЗ, написане для однієї архітектури (наприклад, ARM), на іншій (наприклад, x86_64) [12].

Віртуалізація, на відміну від емулювання, передбачає ізоляцію програм у межах однієї апаратної платформи з використанням гіпервізорів, що дозволяє розгорнути кілька ізольованих систем (гостьових ОС) на одному фізичному комп’ютері. Сучасні технології віртуалізації включають:

- KVM/QEMU – повноцінне середовище для Linux;
- VirtualBox, VMware, Hyper-V – для загального використання;
- Docker, LXC – контейнери, які не емулюють ОС, а лише ізолюють середовище виконання [34].

У тестуванні системного ПЗ емулятори використовуються для:

- тестування нових конфігурацій ядра без потреби в реальному обладнанні;
- перевірки поведінки драйверів в апаратно-незалежному середовищі;
- fuzz-тестування системних викликів, переривань, маніпуляцій пам’яттю.

Інфраструктура KernelCI, наприклад, використовує QEMU для автоматизованого тестування зібраних ядер у тисячах конфігурацій, перевіряючи завантаження, інтерфейси /proc, /sys, початкові логі системи [18].

Використання QEMU спрощує також запуск тестів без загрози

пошкодження основної системи. Наприклад, перевірка некоректної обробки DMA або PCI-конфігурації може викликати зависання або крах ядра – віртуальне середовище дозволяє безпечно експериментувати.

Контейнери (Docker, Podman, LXC) забезпечують ізольоване, відтворюване середовище, що особливо корисно для модульного та інтеграційного тестування компонентів СПЗ, таких як:

- інсталятори драйверів;
- утиліти конфігурації ядра;
- супутнє ПЗ для моніторингу чи логування.

На відміну від повної віртуалізації, контейнери не емулюють ядро – тому вони не застосовуються для тестування внутрішніх компонентів ОС, але чудово підходять для побудови CI-пайплайнів з перевіркою зовнішньої поведінки СПЗ, формування звітів, збирання статистики [35].

Переваги використання віртуалізації та емуляції:

- висока доступність – відсутність необхідності у реальному обладнанні;
- повторюваність – точне відтворення середовища між тестовими прогонками;
- масштабованість – запуск кількох віртуальних інстансів паралельно;
- ізольованість – безпечне виконання ризикованих експериментів.

Обмеження і проблеми:

- неточна емуляція: QEMU не завжди точно відтворює поведінку низькорівневих компонентів (DMA, кеші, обробники переривань);
- обмежений доступ до реального "заліза": перевірка специфічних пристроїв (FPGA, сенсорів, GPU) у віртуальному середовищі часто недоступна;
- продуктивність: повна емуляція (особливо з іншою архітектурою) значно повільніша за виконання на фізичному пристрої;
- зміщення результатів: поведінка у QEMU може не співпадати з реальною системою через відмінності в таймінгах або реалізації [36].

2.4 Генерація тестів: випадкова, модельна, fuzz-тестування

Генерація тестів – це важливий напрям у автоматизованому тестуванні, який дозволяє зменшити ручну працю, охопити широкий простір вхідних даних та виявляти граничні або неочевидні помилки. У системному програмуванні, де традиційні ручні сценарії обмежені масштабованістю й спостережуваністю, методи автоматичної генерації тестів мають особливо високу ефективність. Нижче розглядаються три основні класи таких методів: випадкова генерація, модельно-орієнтована генерація та fuzz-тестування [2].

Випадкова (рандомізована) генерація тестів (Random Testing) передбачає створення тестових даних або послідовностей дій за допомогою випадкових значень. У системному ПЗ цей підхід часто використовується для:

- створення довільних системних викликів (syscalls) для перевірки ядра;
- генерації конфігурацій для драйверів (наприклад, нестандартні параметри init);
- тестування поведінки при порушенні очікувань (нестандартні розміри буферів, невірні дескриптори) [37].

Приклад: фреймворк Trinity для Linux генерує випадкові комбінації системних викликів у різних потоках, виявляючи баги у міжпроцесній синхронізації або обробці помилок. Переваги: простота реалізації, можливість виявлення неочевидних помилок. Недоліки: низьке семантичне покриття, значна кількість неінформативних (некоректних або незначущих) тестів.

Модельна генерація тестів (Model-Based Testing, MBT) – це підхід, що базується на побудові формальної моделі очікуваної поведінки системи, з якої автоматично генеруються тестові випадки. Типові моделі:

- скінченні автомати (Finite State Machines);
- діаграми переходів станів (Statecharts);

- протоколи взаємодії (наприклад, USB, PCIe, ACPI);
- контрактні специфікації API.

У системному ПЗ МВТ застосовується, наприклад, для перевірки:

- станів пристрою (ввімкнення, сплячий режим, пробудження);
- черг запитів у контролерах введення-виведення;
- переходів у протоколах (TCP, USB, NVMe) [28].

Інструменти, як-от TLA+, Spec Explorer, QuickCheck (Erlang/PropEr), дозволяють формалізувати модель та генерувати з неї численні тести, які не були б створені вручну. Переваги: високе охоплення станів, підтримка формального аналізу. Недоліки: потреба у чіткій специфікації, складність побудови моделей для складних систем.

Fuzz-тестування (Fuzzing) – це автоматизована техніка, що полягає у генерації великої кількості некоректних, випадкових або злегка модифікованих вхідних даних для виявлення дефектів, які виникають внаслідок неправильної обробки вхідної інформації.

Fuzzing є особливо ефективним у тестуванні:

- обробки системних викликів;
- парсингу бінарних форматів (ELF, ACPI, firmware blobs);
- реакції драйверів на аномальні події від апаратури [38].

Сучасні інструменти:

- AFL (American Fuzzy Lop) – застосовується до юзерспейс і kernelspace програм, має механізми coverage-guided fuzzing;
- syzkaller – спеціалізований fuzz-движок для ядра Linux, що викликає послідовності syscalls на основі сгенерованих специфікацій;
- KLEE – використовує symbolic execution для генерації входів, що покривають нові шляхи виконання.

Fuzz-тестування дозволяє виявляти дефекти, які не проявляються при нормальному використанні – наприклад, переповнення буфера, помилки сегментації, некоректну обробку помилок, витоки пам’яті [39].

Комбінування генеративних технік стало трендом у розробці сучасних

інструментів:

- Hybrid Fuzzing поєднує випадкову генерацію з аналізом покриття (coverage-guided);
- Concolic Testing (поєднання concrete + symbolic) – приклад: SAGE від Microsoft;
- Grammar-based Fuzzing – дозволяє генерувати структуровані вхідні дані (наприклад, valid ELF-файли), що підвищує глибину аналізу [36].

2.5 Статичний аналіз коду: інструменти, переваги, обмеження

Статичний аналіз коду – це клас методів дослідження програм без їх виконання. На відміну від динамічного тестування, яке працює з ПЗ у процесі його роботи, статичний аналіз орієнтований на перевірку вихідного коду, байт-коду чи машинного коду, з метою виявлення помилок, вразливостей, порушень стандартів або некоректної логіки ще до компіляції або запуску.

У сфері системного програмного забезпечення (СПЗ) статичний аналіз є особливо важливим через:

- низьку спостережуваність під час виконання;
- високі вимоги до безпеки, стабільності й стандартів кодування;
- обмеження можливості відлагодження в реальному середовищі [40].

Класифікація методів статичного аналізу:

- лінгвістичний аналіз – передбачає перевірку синтаксису, стилю, іменування, відповідності стандартам; приклади – clang-tidy, cpplint, checkpatch.pl (для Linux Kernel);
- структурний аналіз – аналіз структури керування (control-flow), залежностей (data-flow), графу викликів; дозволяє виявляти неініціалізовані змінні, “мертвий” код, нескінченні цикли тощо;
- аналіз на основі абстрактних інтерпретацій – методи, що моделюють множину можливих виконань через абстрактну модель, наприклад – діапазони змінних, стан пам’яті. Фреймворки: Frama-C, Astrée [41];

- формальний аналіз. Застосовує логіку, теорію моделей і SMT-солвери; підходить для перевірки складних властивостей (відсутність переповнення буфера, коректне блокування mutex); приклади – CBMC, Infer, VeriFast.

Серед інструментів статичного аналізу для системного ПЗ можна зазначити наступні.

Clang Static Analyzer – нативний інструмент LLVM, що дозволяє знайти:

- витоки пам’яті,
- порушення інваріантів,
- порушення логіки управління потоками.

Активно використовується в проєктах на C/C++, зокрема у модулях ядра Linux, системних бібліотеках, де можливі критичні помилки в обробці пам’яті або файлів.

Coverity Scan – комерційний інструмент, який застосовується у промисловості (Red Hat, Oracle, NASA). Підтримує аналіз мільйонів рядків коду та глибоку інтерпретацію контексту функцій [42].

Frama-C – фреймворк для формального аналізу C-програм з підтримкою анотацій (ACSL). Дає змогу моделювати поведінку коду та доводити певні властивості, наприклад, відсутність арифметичних переповнень [43].

Infer – це Facebook-інструмент для міжпроцедурного аналізу, дозволяє виявляти витоки пам’яті, помилки блокування та умови гонки.

Sparse – інструмент для аналізу Linux kernel, дозволяє виявляти порушення правил типізації, використання незахищених API та структур.

Приклади практичного використання:

- ядро Linux: обов’язковий прогін checkpatch.pl, Sparse і Smatch для pull-запитів;
- платформа Android: використання Infer, clang-tidy у складі AOSP CI;
- автомобільна галузь: застосування Frama-C і MISRA-аналітики для

відповідності ISO 26262.

Переваги статичного аналізу:

- виявлення дефектів до виконання – помилки виявляються ще до компіляції, що знижує вартість виправлення;
- масштабованість – інструменти можуть аналізувати великі проєкти без ручного втручання;
- вимірюваність якості – доступ до метрик складності, покриття, стилю коду;
- інтеграція в CI/CD – можливість запуску аналізу після кожного коміту.

Обмеження:

- хибні спрацювання (false positives) – багато інструментів знаходять помилки, які не є значущими або реальними, що знижує довіру розробників;
- висока вартість конфігурації – складність налаштування під конкретний проєкт або мову;
- низька точність у багатопоточних програмах – статичні аналізатори погано моделюють конкурентні сценарії;
- неповнота перевірок – обмеження в підтримці сучасного C++, макросів, inline-asm у драйверах тощо [44].

2.6 Інструменти автоматизації тестування системного ПЗ

Автоматизація тестування системного програмного забезпечення (СПЗ) вимагає не лише відповідної методології, а й потужного інструментарію, здатного працювати у складних умовах: на рівні ядра, у віртуалізованому середовищі, з обмеженим доступом до відлагодження. У цьому підрозділі розглядаються найбільш поширені та ефективні інструменти, які підтримують різні типи аналізу – символічне виконання, формальну верифікацію, fuzz-тестування, статичний і динамічний аналіз, а також автоматизовані пайплайни.

KLEE: Symbolic Execution [45]. Інструмент символічного виконання (symbolic execution) для програм, написаних на C/C++, що працюють у просторі користувача. Він дозволяє автоматично генерувати тести, які забезпечують високе покриття коду. KLEE виконує програми з символічними (неконкретними) значеннями і досліджує всі можливі шляхи виконання, використовуючи SMT-солвери. Це дозволяє виявити рідкісні баги та граничні умови.

У системному ПЗ KLEE використовується для:

- перевірки бібліотек, що викликаються з драйверів;
- модульного тестування невеликих компонентів ядра, ізольованих у користувацькому просторі.

CBMC: Bounded Model Checking [14]. CBMC (C Bounded Model Checker) – інструмент формального аналізу, що перевіряє властивості програм (відсутність переповнень, порушень пам’яті, assert-умов) за допомогою техніки обмеженого моделювання (bounded model checking). CBMC ефективно використовується для верифікації вбудованого ПЗ, модулів обробки переривань, керування ресурсами.

Переваги:

- доведення властивостей;
- можливість інтеграції в CI;
- підтримка MISRA і специфікацій (ACSL).

AFL: Fuzzing [11]. AFL (American Fuzzy Lop) – один із найефективніших fuzz-інструментів з підтримкою coverage-guided fuzzing. AFL використовує генетичні алгоритми для еволюції тестових випадків і виявляє нові шляхи виконання.

Застосування в СПЗ:

- fuzz-тестування драйверів;
- перевірка обробки нестандартних вхідних даних в API ядра;
- генерація мінімальних прикладів, які викликають креш.

Valgrind: Динамічний аналіз [50]. Популярний фреймворк для

динамічного аналізу, який дозволяє виявляти:

- витоки пам’яті;
- неправильний доступ до пам’яті;
- порушення правил синхронізації.

Valgrind може бути використаний при тестуванні:

- бібліотек, які викликаються з ядра;
- утиліт, що взаємодіють із драйверами;
- коду, що виконується в sandbox'і [46].

Обмеження: не працює для коду в просторі ядра (але може симулювати поведінку в user-space).

QEMU: Емуляція системи [47]. Потужний емулятор, що дозволяє запускати цілу операційну систему з конфігурацією, максимально наближеною до реального «заліза».

Переваги:

- підтримка різних архітектур (x86, ARM, RISC-V);
- використання в KernelCI;
- підтримка snapshot'ів, дебагу, трасування.

QEMU незамінний у тестуванні boot-сценаріїв, нових версій ядра, драйверів, конфігурацій BIOS/UEFI.

Clang Static Analyzer: Статичний аналіз коду. Інструмент із сімейства LLVM, що виконує статичний аналіз C/C++ коду. Виявляє:

- витоки пам’яті;
- помилки логіки;
- порушення інваріантів.

Використовується в системних проєктах (наприклад, FreeBSD, ядро Linux) для підтримки якості коду на рівні комітів [31].

Jenkins та GitLab CI: Автоматизовані пайплайни [51]. Є засобами автоматизації збирання, тестування, запуску та деплою ПЗ. Для СПЗ вони використовуються для:

- складання ядра з різними конфігураціями;

- запуску модульних, інтеграційних і fuzz-тестів;
- інтеграції з QEMU, Docker, Valgrind, CBMC тощо [48].

У таких пайплайнах можна реалізувати комплексну верифікацію:

Git Push → Static Analysis → Build Kernel → Run QEMU → Fuzz + Formal Check → Report

2.7 Порівняльний аналіз сучасних інструментів автоматизації

У процесі автоматизованого тестування системного програмного забезпечення важливо не лише обрати метод, а й правильно підібрати інструмент, що відповідає типу коду, архітектурі, рівню доступу, етапу життєвого циклу та вимогам до сертифікації. У цьому підрозділі представлено порівняльний огляд інструментів, розглянутих раніше, за ключовими критеріями застосування в системному програмуванні.

Для структурованого аналізу використано такі параметри:

- тип аналізу: статичний, динамічний, символічний, fuzz;
- підтримка системного ПЗ: чи підходить для роботи з ядром, драйверами, low-level модулями;
- автоматизованість: можливість інтеграції в CI/CD;
- глибина виявлення помилок: виявлення критичних дефектів (переповнення буфера, гонки, витоки);
- придатність до сертифікації: чи підтримує використання у проектах з суворими стандартами (DO-178C, ISO 26262 тощо).

Таблиця 2.2 – Порівняння інструментів автоматизації тестування СПЗ

Інструмент	Тип аналізу	Сфера застосування	Інтеграція в CI	Придатн. до СПЗ	Переваги	Обмеження
KLEE	Символічний	Модульне тестування (user-space)	Часткова	Непряма	Глибоке покриття шляхів	Обмежена підтримка kernel-space
CBMC	Формальна верифікація	Вбудовані компоненти	Так	Так	Виведення властивостей, MISRA	Високе навантаження на ресурси
AFL	Fuzzing	Системні виклики, драйвери	Так	Так	Швидке знаходження edge-case	Не гарантує повноту покриття
Valgrind	Динамічний	User-space утиліти	Так	Частково	Глибокий аналіз пам’яті	Не працює з kernel-mode кодом
QEMU	Емуляція	Ядро, системні образи	Так	Так	Апаратна незалежність, CI-ready	Не точно моделює апаратну поведінку
Clang Static Analyzer	Статичний	Системні бібліотеки, ядро	Так	Так	Інтеграція з LLVM, швидкість	Хибні спрацювання, обмежена глибина
Jenkins / GitLab CI	Автоматизація	Весь процес	Так	Так	Повна інтеграція тестів	Потребує налаштування інфраструктури

Рекомендації щодо вибору можна визначити такі.

На ранніх етапах розробки доцільно використовувати KLEE для генерації тестів і Clang Static Analyzer для базової перевірки коду.

Для формальної верифікації (особливо у сертифікованих системах) рекомендовано CBMC або Frama-C, що підтримують доведення коректності.

Для інтенсивного fuzz-тестування найкраще підходить AFL (або syzkaller у Linux).

Для перевірки цілісності під час складання – Jenkins чи GitLab CI із вбудованими static/fuzz/integration етапами.

Для тестування ОС у цілому – поєднання QEMU з автозапуском сценаріїв і журналюванням результатів.

Слід зазначити, що у реальних умовах ці інструменти найчастіше використовуються комбіновано, наприклад:

1) Static Check (Clang, CBMC);

2) Build Kernel (Make, Cross-compile);

3) Run Tests in QEMU;

4) Run AFL fuzzing on syscalls;

5) Collect coverage and logs;

6) Generate Report in GitLab CI.

Такий підхід дозволяє поєднати формальну гарантію коректності, динамічне спостереження, автоматизоване тестування і безперервну інтеграцію, що є стандартом сучасного DevOps-підходу в розробці системного ПЗ.

2.8 Висновки до розділу

Автоматизоване тестування системного програмного забезпечення потребує багаторівневого, гнучкого й формалізованого підходу до класифікації. Різні аспекти – ступінь доступу до внутрішньої структури, тип тестованого об'єкта, методи генерації сценаріїв, глибина аналізу – формують

багатовимірний простір можливих технік. У сучасних фреймворках (наприклад, KernelCI, syzkaller, CBMC) ці класифікації комбінуються, що дозволяє досягати балансу між повнотою перевірки, продуктивністю й масштабованістю.

Тестування за принципом "ящика" є не просто теоретичною класифікацією, а важливим практичним критерієм вибору методів у системному програмуванні. Для повноцінного забезпечення якості СПЗ потрібна комбінація всіх трьох підходів: black-box тести – для перевірки зовнішніх API; white-box – для верифікації критичної логіки; grey-box – для інтеграційного аналізу на реальному або змодельованому обладнанні. Вибір залежить від доступу до коду, ризиків, вимог до сертифікації та ресурсів, доступних для автоматизації.

Використання емуляторів та віртуалізації у тестуванні системного ПЗ стало ключовим чинником масштабування, автоматизації та підвищення ефективності перевірок. Ці технології дозволяють виявляти помилки ще до впровадження у фізичне середовище, мінімізуючи витрати на обладнання та підвищуючи швидкість розробки. Проте вони не замінюють реальне тестування на "залізі", а лише доповнюють його, утворюючи комплексну багатошарову інфраструктуру перевірки.

Методи генерації тестів дозволяють системним розробникам автоматизувати процес створення багаточисельних сценаріїв, що охоплюють як звичайні, так і крайні випадки. Випадкові методи дають змогу виявити неочевидні дефекти; модельні – формалізують поведінку і дозволяють систематичне тестування; fuzz-тестування – ефективно працює з нестандартними або шкідливими вхідними даними. В умовах високих вимог до безпеки та надійності СПЗ, ці підходи стають не просто допоміжними, а обов’язковими елементами автоматизованого контролю якості.

Статичний аналіз є одним із найефективніших інструментів забезпечення якості у системному ПЗ. Його застосування дозволяє не лише знизити витрати на виявлення та виправлення помилок, а й підвищити

надійність, підтримку стандартів кодування та відповідність сертифікаційним вимогам. Хоча ці інструменти не здатні охопити всі аспекти поведінки програми, їхнє поєднання з іншими методами – динамічним аналізом, fuzzing, формальними методами – дозволяє побудувати комплексну інфраструктуру верифікації.

Сучасне тестування системного ПЗ базується на використанні широкого спектру інструментів, які охоплюють усі рівні перевірки – від аналізу коду до запуску системи в емуляції. Інструменти, що поєднують аналіз коду, символічне виконання, fuzzing та автоматизацію, дозволяють зменшити людську помилку, масштабувати перевірки та інтегрувати їх у безперервну розробку. Ефективна інфраструктура тестування неможлива без об'єднання цих засобів у єдину CI/CD систему.

Ефективне тестування системного програмного забезпечення неможливе без застосування спеціалізованих інструментів, які охоплюють увесь спектр методів – від символічного виконання до автоматизованої генерації fuzz-тестів та інтеграції в пайплайни. Порівняльний аналіз показує, що жоден з інструментів не є універсальним, однак їх комбінація дозволяє досягти високої глибини перевірки, продуктивності й відповідності галузевим вимогам. Правильне впровадження цих засобів забезпечує якість, надійність і масштабованість системного ПЗ на всіх етапах життєвого циклу.

3 МЕТОДИ ФОРМАЛЬНОЇ ВЕРИФІКАЦІЇ СИСТЕМНОГО ПЗ

3.1 Формальні методи: логіка, моделі, автомати

Формальні методи – це математично обґрунтовані техніки аналізу, доведення та верифікації програмного забезпечення, що дозволяють отримати строгі гарантії коректності. На відміну від емпіричних підходів (тестування, інспекції), формальні методи ґрунтуються на логічних моделях і дозволяють довести правильність поведінки ПЗ відносно специфікацій. У сфері системного програмного забезпечення, де помилки можуть спричинити апаратні збої або компрометацію безпеки, формальна верифікація є критично важливою, особливо для проектів із високим рівнем відповідальності (safety-critical systems) [52].

В основі формальних методів лежать логічні формалізми, зокрема:

- пропозиційна логіка – для опису булевих властивостей;
- предикатна логіка першого порядку – для опису відношень між змінними, властивостей структур даних;
- темпоральна логіка (LTL, CTL) – для опису послідовності подій у часі, часто застосовується в системах з автоматами та станами [19].

Завдяки цим формалізмам можна:

- моделювати поведінку програми як математичну структуру (граф, автомат, дерево);
- формалізувати вимоги як логічні твердження;
- застосовувати логічні доведення або автоматизовані солвери для перевірки виконання цих тверджень.

Одним із найпоширеніших способів формалізації поведінки ПЗ є скінченні автомати (finite state machines, FSM) або їх розширення:

- автомати з виходами (Mealy, Moore);
- автомати з вагами, часовими параметрами (Timed Automata);

- автомати з pushdown-стеком (для моделювання викликів функцій) [53].

У системному ПЗ автомати використовуються для:

- моделювання станів драйверів пристроїв (відключено, ініціалізовано, активно);
- опису протоколів обміну (наприклад, I2C, USB, NVMe);
- формалізації життєвих циклів об'єктів ядра (thread, mutex, file descriptor).

Такі моделі можуть бути автоматично перевірені на предмет відсутності зациклення, порушення інваріантів, досяжності станів або недетермінізму.

Логіка Гора – класичний метод доведення коректності імперативних програм, який використовує триплети Гора виду: {P} C {Q}, де P – передумова, C – програма (інструкція або блок), Q – постумова.

Цей підхід дозволяє:

- довести правильність функцій у драйверах (наприклад, що буфер завжди перевіряється на null перед використанням);
- формалізувати інваріанти циклів;
- описати поведінку синхронізаційних примітивів у многопотокових середовищах [54].

Системне ПЗ часто містить паралельні обчислення, прямий доступ до пам’яті (DMA), використання кешів. Для формальної верифікації таких сценаріїв застосовуються:

- моделі пам’яті (memory models) – для аналізу гонок, некоректних узгоджень;
- моделі взаємодії потоків – з аналізом deadlock, starvation, race condition.

Такі моделі реалізуються в інструментах на кшталт CBMC, SPIN, VeriFast, TLA+, PROMELA, які дозволяють формалізувати та перевірити сценарії конкурентного доступу до ресурсів [55].

Важливою складовою формальної верифікації є опис специфікацій – тобто, яку поведінку система повинна гарантувати. Для цього використовуються:

- контрактне програмування (Design by Contract): require, ensure, invariant;
- формальні мови специфікацій: ACSL (для C), TLA+, Z, B, Alloy;
- власні DSL (Domain Specific Languages) – для опису специфікацій драйверів, контролерів, тощо [43].

У Linux, наприклад, для аналізу драйверів застосовується Smatch, який працює з простими логічними специфікаціями (файли правил), а в безпечних RTOS використовуються спеціалізовані модулі перевірки контрактів API.

3.2 Метод перевірки моделей (Model Checking)

Model Checking – це метод формальної верифікації, що передбачає автоматичну перевірку, чи задовольняє математична модель системи задану логічну специфікацію. На відміну від тестування, яке перевіряє окремі сценарії виконання, model checking досліджує всі можливі стани й переходи системи, що дозволяє виявити навіть рідкісні або граничні помилки. Цей підхід є особливо цінним для системного ПЗ, де висока складність і багатопотокова поведінка ускладнюють тестування вручну [56].

Model checking будується на трьох компонентах:

- модель системи – формальне подання поведінки (зазвичай у вигляді скінченного автомата або графа станів);
- специфікація властивостей – формулюється у логіках LTL (Linear Temporal Logic) або CTL (Computation Tree Logic);
- модельний перевірювач – алгоритм або інструмент, який систематично досліджує всі шляхи виконання моделі, перевіряючи істинність властивостей [19].

Приклад логічної властивості: G (request → F grant) (завжди: якщо є

запит, то зрештою буде надано дозвіл).

Model checking дозволяє:

- довести властивості системи або знайти контрприклади (trace of failure);
- автоматично перевірити відсутність дедлоків, гонок, зациклення;
- верифікувати реактивні системи, які постійно взаємодіють із середовищем (драйвери, task scheduler, DMA-контролери) [53].

У системному ПЗ цей метод використовується для:

- аналізу поведінки систем багатозадачності;
- перевірки коректності реалізації системних протоколів;
- верифікації ядра реального часу (RTOS).

Таблиця 3.1 – Основні інструменти перевірки моделей

Найменування	Властивості
SPIN	- мова моделювання: PROMELA (Process Meta Language); - орієнтований на моделювання паралельних процесів; - перевіряє властивості, задані в LTL; - підтримує генерацію контрприкладів, трасування помилок [57]
NuSMV / nuXmv	- призначений для перевірки скінченних станів; - підтримує CTL і LTL; - використовується в індустрії для формального аналізу цифрових систем і ПЗ
UPPAAL	- спеціалізований інструмент для перевірки таймованих автоматів; - застосовується в аналізі реального часу – наприклад, перевірка затримок у планувальниках ОС
TLA+	- заснований на темпоральній логіці; - підходить для моделювання алгоритмів керування станами, протоколів ядра.

Метод перевірки моделей забезпечує повноту аналізу в межах заданої моделі, проте має низку обмежень:

- state space explosion – експоненційне зростання кількості станів при ускладненні системи;
- необхідність абстракції – модель має бути простою та обмеженою, щоб її можна було проаналізувати;
- складність формалізації специфікацій – потрібно формалізувати бажану поведінку у логічному форматі;
- високі вимоги до ресурсів – перевірка великих моделей потребує значної обчислювальної потужності [58].

Для боротьби з "state explosion" використовуються: обмеження глибини перевірки (bounded model checking); символічне моделювання (BDD, SMT-солвери); паралельні алгоритми верифікації.

Прикладом застосування може стати аналіз драйвера в SPIN. Модель поведінки драйвера може бути описана як FSM:

```
proctype Driver() {
    state: do
        :: req -> busy=true
        :: ack -> busy=false
    od
}
```

Властивість: драйвер ніколи не надає доступ до ресурсу, якщо вже зайнятий:

```
[] !(busy && req)
```

SPIN згенерує автомат, який перевіряє цю властивість, і або підтвердить її істинність, або надасть приклад виконання, що порушує її.

3.3 Докази коректності: логіка Гора, SMT-солвери

Формальне доведення коректності програмного забезпечення базується на строгих логічних методах, що дозволяють гарантувати, що програма завжди виконує задану функцію правильно, відповідно до специфікацій. Цей підхід не лише виявляє помилки, а й доводить, що певні категорії помилок не можуть трапитися в принципі. У контексті системного програмування (ядро ОС, драйвери, HAL) таке доведення є важливою складовою безпечного програмного середовища, зокрема в проєктах, які потребують сертифікації (DO-178C, ISO 26262 тощо).

Як зазначалося в підрозділі 3.1, логіка Гора дозволяє формалізувати програму у вигляді трійки: {P} C {Q}, де: P – передумова: що вірно до виконання програми; C – команда або блок коду; Q – постумова: що гарантовано після виконання C.

Цей підхід дозволяє модульно доводити коректність:

- арифметичних обчислень;
- інваріантів циклів;
- викликів функцій з перед- і післяумовами;
- властивостей вказівників, структури пам’яті.

У системному ПЗ логіка Гора використовується, наприклад, у Frama-C через систему анотацій ACSL (ANSI/ISO C Specification Language). Вона дозволяє описувати контракти C-функцій:

```
/*@
  requires len > 0;
  ensures \result == \sum(0, len-1, i, array[i]);
*/
int sum(int* array, int len);
```

Цей опис може бути формально перевірений за допомогою автоматизованих або напівавтоматичних інструментів.

SMT (Satisfiability Modulo Theories) – це клас алгоритмів і програм, що

визначають, чи існує така інтерпретація логічних формул (на рівні арифметики, логіки масивів, покажчиків, булевих змінних), за якої ці формули істинні.

У верифікації системного ПЗ SMT-солвери використовуються:

- як бекенд для символічного виконання (наприклад, у KLEE);
- для перевірки доведень у bounded model checking (CBMC, ESBMC);
- як частина формального аналізу на C/ASM-рівні (Frama-C, VeriFast, VCC, Why3) [59].

Таблиця 3.2 – Інструменти для доведення коректності

Найменування	Властивості
Frama-C	- працює з ACSL; - підтримує модульні доведення з використанням Z3, Alt-Ergo; - підтримує формальний аналіз вказівників, доступу до пам’яті, буферів
VeriFast	- інтерактивний доводчик для C і Java; - орієнтований на перевірку розділених об’єктів, інваріантів; - використовується для доведення відсутності гонок у багатопоточних програмах
VCC (Verified C Compiler)	- інструмент Microsoft для доведення C-коду (переважно вбудованого); - інтегрується з Z3
Why3 / SPARK / Dafny	- системи з мовами програмування або контрактів, що дозволяють писати програму разом із формальними доказами; - активно використовуються в аерокосмічній галузі (Ada/SPARK)

Найбільш поширені SMT-солвери:

- Z3 (Microsoft Research): потужний, з інтеграцією до .NET, Frama-C, Dafny;
- CVC4/CVC5: ефективний солвер з відкритим кодом;
- Boolector: спеціалізований для булевої арифметики.

Доведення коректності з Frama-C можна розглянути на прикладі мови C з ACSL:

```
/*@
  requires \valid_read(arr + (0..n-1));
  requires n > 0;
  ensures \forall integer i; 0 <= i < n ==> \result >= arr[i];
*/
int max(int* arr, int n) {
  int m = arr[0];
  for (int i = 1; i < n; ++i) {
    if (arr[i] > m)
      m = arr[i];
  }
  return m;
}
```

Frama-C із солвером Z3 зможе перевірити, що функція max дійсно повертає максимум у масиві довжини n, якщо дотримані умови.

Слід зазначити, що у проектах на кшталт seL4, VxWorks, AUTOSAR OS доведення коректності є обов’язковим. Наприклад:

- ядро seL4 формально доведене до рівня C-коду з допомогою Isabelle/HOL;
- Frama-C використовується в Airbus для верифікації критичного коду;
- SMT-солвери забезпечують автоматичну перевірку тисяч контрактів у секунду [60].

3.4 Приклади формального аналізу драйверів і ядра ОС

Формальна верифікація системного ПЗ досягла значного прогресу в останні роки. Нижче наведено кілька сучасних прикладів, де формальні

методи успішно застосовувалися до драйверів та компонентів ядра.

Формальна верифікація функцій планувальника ядра Linux [69]. Дослідники з Inria використовували Frama C для формального аналізу функції should_we_balance() у планувальнику ядра. Вони довели коректність її поведінки й показали, що ця верифікація залишається стабільною навіть при оновленнях ядра.

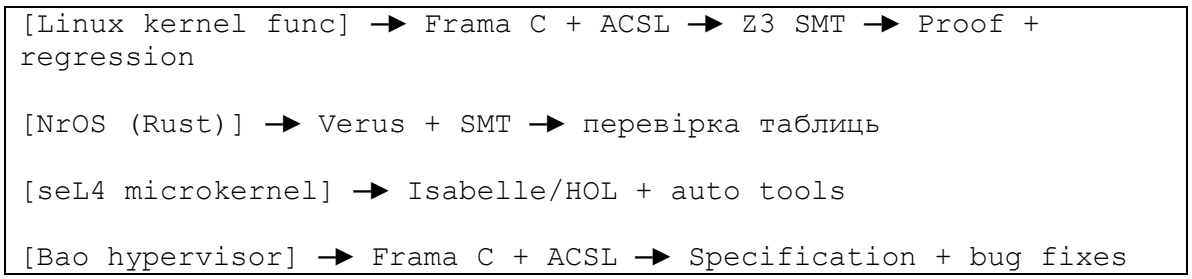

Рисунок 3.1 – Схематизація процесів формальної верифікації

Verus та NrOS: верифікація у Rust-ядрі [70]. Проєкт NrOS, що реалізується на Rust, застосував Verus для формального доведення коректності реалізації таблиць сторінок і модулів введення-виведення. Доведено, що ключові алгоритми відповідають специфікаціям.

Автоматизація в seL4 [71]. Формально доведене мікроядро seL4 із 2023 р. перебуває в процесі автоматизації перевірки для платформи AArch64. Компанія Proofcraft робить акцент на нових фреймворках, що мінімізують ручну роботу експертів.

У 2024 р. було проведено формальний аналіз алгоритму кеш-розфорбовування у гіпервізорі Bao за допомогою Frama C з описом властивостей кешу [72]. Виявлено й виправлено subtle-баги, що покращило безпеку реакції системи. Робота [73] кінця 2023 р. присвячена автоматичному перетворенню коду драйверів OpenBSD у формальні докази у Isabelle/HOL. Підтверджується відповідність функцій реалізації їх специфікаціям. Інноваційний проєкт FVEL (2023) [62] використовує технології генеративних

моделей (LLM + ATP), щоб автоматизувати генерування доведень Isabelle замість ручного кодування контрактів C. Це прокладає шлях до масштабованого формального аналізу.

3.5 Практичні інструменти: SPIN, NuSMV, CBMC, Frama-C

Сучасні формальні інструменти для перевірки програмного забезпечення не лише реалізують математичні алгоритми доведення, а й інтегруються у процеси CI/CD, підтримують стандартні мови програмування (зокрема, C, C++) та працюють з реальними проєктами, зокрема у сфері системного ПЗ. Нижче наведено огляд найбільш відомих і практично застосовуваних інструментів.

SPIN (Simple Promela Interpreter) [53]. Призначений для формальної верифікації паралельних/реактивних систем на основі model checking. Мова опису моделей – PROMELA (Process Meta Language). Прикладом застосування є моделювання драйвера пристрою, де SPIN може перевіряти, що запит ресурсу ніколи не буде оброблено двічі поспіль без звільнення (відсутність подвійного доступу).

Особливості:

- використовується для аналізу багатопоточних взаємодій (mutex, семафори, гонки);
- підтримує LTL властивості, перевірку зациклення, блокувань;
- дозволяє генерувати трасу порушення властивості.

NuSMV / nuXmv [63]. Призначений для перевірки властивостей скінченних станів, підтримка CTL/LTL логіки. Прикладом застосування є перевірка логіки перемикання станів у енергетичних модулях ОС (наприклад, глибокий сон/активація).

Особливості:

- підходить для верифікації систем з великою кількістю конфігурацій;
- підтримує булеву арифметику, таймери, інтервали;

- надає символічну модель, яку можна експортувати або зберегти для подальшого аналізу.

CBMC (C Bounded Model Checker) [14]. Призначений для перевірки програм C/C++ на предмет assert порушень, переповнень, гонок, аналіз покриття. В якосты прикладу застосування можна зазначити виявлення out-of-bounds доступів у буферах системних викликів або перевірка відсутності unsigned overflow у підсистемі таймерів ядра.

Особливості:

- використовує SMT-солвери як бекенд (наприклад, Z3, CVC4);
- дозволяє інтегрувати у CI-середовище;
- аналізує рівень джерельного коду без моделювання.

Frama-C [43]. Фреймворк для формального аналізу C коду із системою специфікацій ACSL (ANSI/ISO C Specification Language).

Особливості:

- модульна перевірка функцій на відповідність контракта;
- плагіни для value analysis, WP (weakest precondition), taint analysis;
- підтримка інтеграції з Z3, Alt-Ergo, Why3;
- використовується в промисловості (Airbus, TrustInSoft).

Таблиця 3.3 – Порівняння інструментів верифікації

Інструмент	Мова	Тип аналізу	Основні галузі застосування
SPIN	PROMELA	Model checking	Паралелізм, протоколи, планувальники
NuSMV	.smv	CTL, LTL перевірка	FSM, протоколи, таймінг
CBMC	C/C++	Bounded checking	Буфери, системні виклики, CI/CD
Frama-C	C + ACSL	Контрактне доведення	Драйвери, ядра, авіоніка

Наприклад, може застосовуватися для аналіза вказівників у файловій підсистемі ядра, доведення відсутності use-after-free помилок у системах керування пам’яттю.

Сучасні інструменти (CBMC, Frama-C) інтегруються з Jenkins, GitLab CI, що дозволяє автоматизувати формальну перевірку коду під час кожної збірки. Приклад пайплайну:

```
Git push → Jenkins → Build → CBMC analysis → Frama-C WP → Report
→ Merge/push
```

3.6 Проблеми масштабованості формальних методів і шляхи їх вирішення

Найбільш поширеною проблемою є експоненційне зростання кількості станів, які необхідно дослідити при застосуванні методів перевірки моделей (Model Checking) або символічного виконання. У випадку з системним ПЗ – ОС, драйверами, гіпервізорами – кількість можливих комбінацій переходів, особливо у багатопотоковому середовищі, може досягати мільярдів.

Причини:

- високий ступінь паралелізму;
- взаємозалежність апаратних ресурсів (регістри, шини, DMA);
- складність обробки помилок та переривань;
- наявність нестандартних синхронізаційних шаблонів.

Ще однією значною перепоною є складність створення формальної специфікації поведінки системного компонента. Часто поведінка не документована повністю, має імпліцитні залежності або формується історично, без чітких контрактів.

Високий рівень складності спостерігається при:

- описі API ядра;
- специфікації конкурентної поведінки;
- формалізації керування пам’яттю.

Також формальні аналізатори (Frama-C, CBMC, SPIN) можуть потребувати великого обсягу оперативної пам’яті, багатоядерного процесора, особливо при аналізі великих програм. При збільшенні довжини шляхів виконання час доведення може зростати від секунд до годин.

Вказані проблеми вирішуються у способи, зазначені нижче.

Обмежене моделювання (Abstraction & Slicing). Зменшення кількості перевірюваних шляхів через:

- абстрагування внутрішньої поведінки окремих модулів (interface contract approach);

- «нарізання» коду (program slicing) для перевірки лише критичних ділянок.

Bounded Model Checking (BMC) [14]. У випадках, коли повний аналіз неможливий, обмежується глибина перевірки (наприклад, 10-20 кроків виконання). Цей підхід застосовують CBMC та ESBMC.

Композиційна верифікація [43]. Модель розбивається на компоненти, для кожного з яких перевіряється локальна специфікація. Згодом ці специфікації об’єднуються у глобальне твердження (compositional reasoning). Наприклад, драйвер перевіряється окремо від DMA, а далі – разом у повному ланцюгу.

Інкрементальна та регресійна перевірка [64]. Формальні інструменти можуть зберігати результати попередніх доведень і перевіряти лише модифіковані частини. Так працюють плагіни Frama C WP, VCC, Dafny.

Інтеграція LLM і AutoProof [62]. Нові фреймворки (FVEL, TLA+ GPT Assisted) дозволяють автоматично генерувати логічні інваріанти, контракти та анотації за допомогою великих мовних моделей. Це зменшує вимоги до експертності команди.

У формальній перевірці драйвера вводу-виводу можуть поєднуватися:

- перевірка інтерфейсної специфікації з Frama-C;
- bounded model checking з CBMC на функціях обробки;
- SPIN-модель на рівні протоколу доступу до пам’яті;

- перевірка взаємодії із DMA через проміжну модель у NuSMV.

Таблиця 3.4 – Практичні кейси подолання проблеми масштабованості

Проєкт / Ядро	Підхід	Інструмент(и)
seL4	інкрементальне доведення	Isabelle, AutoCorres
Linux scheduler	slicing + WP	Frama C
Bao hypervisor	BMC + контрактне доведення	CBMC + ACSL
Verus + NrOS	SMT аналіз обмежених структур	Verus, Z3
Dafny формалізація	модульний аналіз	Dafny, Boogie

3.7 Висновки до розділу

Формальні методи створюють логічну основу для глибокого аналізу системного ПЗ, особливо в аспектах, де звичайне тестування є неефективним або небезпечним. Вони дозволяють не лише знаходити помилки, а й математично доводити їх відсутність у межах певної моделі. Хоча їх застосування потребує значних ресурсів і знань, формальна верифікація є незамінною для критичних систем, де на кону – безпека, стабільність і надійність.

Model checking є одним з найпотужніших методів формального аналізу, який дозволяє виявляти помилки у всіх можливих шляхах виконання в рамках заданої моделі. Він незамінний у перевірці поведінки критичних компонентів системного ПЗ, особливо тих, що залежать від паралелізму, взаємодії процесів і керування станами. Попри обмеження щодо масштабованості, його використання в поєднанні з іншими формальними та емпіричними методами дозволяє істотно підвищити якість і безпеку системного коду.

Формальні докази коректності є невід’ємним елементом надійної розробки системного ПЗ. Поєднання логіки Гора, мови контрактів та SMT-солверів дозволяє перевіряти функції, які раніше вважались занадто

складними для формального аналізу. Хоча такі доведення потребують ретельної підготовки коду і специфікацій, вони забезпечують найвищий рівень довіри до програм, особливо у сферах, де неприпустимі помилки – авіація, автомобільна промисловість, телекомунікації, операційні системи.

У 2023–2024 роках формальна верифікація системного ПЗ зробила великий крок уперед: від доведення окремих функцій до інтеграції автоматизованих фреймворків із LLM. Попри виклики зі складністю специфікацій та масштабом моделей, вже існують практичні докази коректності ядра, драйверів і кеш алгоритмів – у Linux, Rust-ядрах, мікроядрах та гіпервізорах.

Інструменти SPIN, NuSMV, CBMC, Frama-C забезпечують високий рівень перевірки системного ПЗ – від моделей поведінки до рівня коду. Їхнє застосування в реальних системах (ядрах, драйверах, гіпервізорах) підтверджує ефективність формальної верифікації як практичного інструмента забезпечення надійності та безпеки ПЗ.

Попри відомі проблеми масштабованості, формальна верифікація наближається до інтеграції в повсякденну практику інженерів системного ПЗ. Ключ до цього – розбиття моделей, обмежене моделювання, автоматизація контрактів, а також використання гібридних стратегій аналізу. Нові тенденції, як-от LLM підтримка, вказують на перспективу зменшення бар’єру входу та прискорення процесів перевірки в майбутньому.

4 ПРОЄКТУВАННЯ СИСТЕМИ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ТА ВЕРИФІКАЦІЇ

Для побудови й дослідження системи автоматизації тестування та верифікації системного ПЗ у цій роботі обрано модуль драйвера символьного пристрою у середовищі ядра Linux (версія 6.x), що відповідає за символьне введення/виведення через /dev/demochar.

Такий вибір зумовлений:

- поєднанням низькорівневої роботи з ядром ОС та можливістю тестування без складного апаратного забезпечення;
- наявністю типових вразливостей (наприклад, буферне переповнення, гонки доступу, неправильна обробка помилок);
- можливістю ізольованого тестування за допомогою віртуального середовища.

Сценарій роботи драйвера є наступним.

Крок 1. Ініціалізація модуля в ядрі.

Крок 2. Реєстрація character device.

Крок 3. Реалізація системних викликів open(), read(), write(), close().

Крок 4. Звільнення ресурсів під час виключення модуля (rmmod).

Код драйвера реалізовано мовою C з використанням інтерфейсів ядра Linux (API file_operations), з дотриманням загальної структури модулів ядра.

4.1 Архітектура програмного комплексу

Проєктована система автоматизації складається з основних компонентів, наведених на рисунку 4.1.

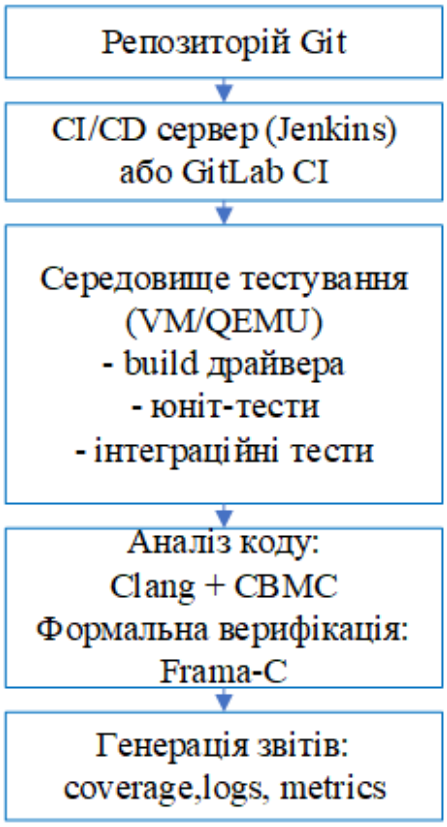

Рисунок 4.1 – Архітектура програмного комплексу

Clang Static Analyzer автоматично виконує статичний аналіз на предмет:

- неправильного використання вказівників;
- неініціалізованих змінних;
- витоків пам’яті.

CBMC здійснює перевірку на:

- можливі арифметичні переповнення;
- виходи за межі буферів;
- порушення умов assert().

Frama-C + ACSL використовується для:

- контрактного опису критичних функцій драйвера;
- доведення відсутності гонок і небезпечних операцій з пам’яттю.

QEMU забезпечує запуск Linux зі зміненим ядром у віртуальному середовищі.

KUnit/kselftest забезпечує тестування окремих модулів ядра без

перезавантаження.

Особливу увагу в системі приділено:

- ізоляції середовища тестування: усі перевірки проводяться у QEMU, що виключає вплив на хост систему;
- адаптивності: структура шаблонів дозволяє змінювати параметри без перекомпіляції драйвера;
- розширюваності: додавання нових модулів або тестів здійснюється через просте копіювання шаблонів і YAML-файлів.

4.2 Побудова пайплайну CI/CD з елементами формального аналізу

Інтеграція методів тестування та формальної верифікації в конвеєр CI/CD (Continuous Integration/Continuous Delivery) забезпечує автоматичну перевірку змін у коді драйвера на кожному етапі життєвого циклу. Це критично важливо для системного ПЗ, де помилки можуть призвести до порушення стабільності всієї ОС.

CI/CD-система побудована на основі GitLab CI і складається з таких етапів.

Етап 1. Pipeline Trigger:

- здійснюється при git push або при створенні merge request;
- автоматичне формування середовища: збір образу, копіювання драйвера.

Етап 2. Build Stage:

- збирання драйвера за допомогою make з використанням KDIR=/lib/modules/$(uname -r)/build;
- вивід артефактів для наступних етапів.

Етап 3. Static Analysis:

- запуск clang-analyzer, cppcheck, coccinelle;
- вивід в лог-файл усіх попереджень.

Етап 4. Formal Verification:

- використання CBMC для bounded model checking критичних функцій;

- запуск Frama-C (плагін WP) з ACSL-контрактами.

Етап 5. Unit Testing:

- виконання kunit-тестів у віртуальному середовищі;

- генерація покриття коду (gcov, lcov).

Етап 6. Integration Testing:

- запуск у QEMU/VM тестових сценаріїв, що взаємодіють із /dev/demochar;

- виявлення регресій при роботі з реальними I/O-операціями.

Етап 7. Reporting – формування HTML-звітів про:

- кількість знайдених дефектів;

- покриття коду;

- результати формального аналізу;

- час виконання.

Фрагмент сценарію YAML для GitLab CI наведений у лістингу 4.1.

Лістинг 4.1 – Фрагмент сценарію для GitLab CI

```
stages:
  - build
  - static_analysis
  - formal_verification
  - unit_tests
  - report

build_module:
  stage: build
  script:
    - make KDIR=/lib/modules/$(uname -r)/build
  artifacts:
    paths:
      - demochar.ko

static_analysis:
  stage: static_analysis
  script:
    - clang --analyze demochar.c
    - cppcheck --enable=all demochar.c
```

```
formal_verification:
  stage: formal_verification
  script:
    - cbmc demochar.c --bounds-check --pointer-check
    - frama-c -wp -rte demochar.c

unit_tests:
  stage: unit_tests
  script:
    - ./run_kunit_tests.sh

generate_report:
  stage: report
  script:
    - ./generate_coverage.sh
    - ./generate_formal_report.sh
```

Frama C інтегрується у пайплайн як окремий аналізатор, що працює з ACSL анотованими функціями. Аналізуються критичні функції типу write, read, ioctl. При порушенні контракта (наприклад, неправильний діапазон зсуву у буфері) пайплайн помічає збірку як помилкову.

Приклад ACSL анотації:

```
/*@
  requires \valid_read(buf + (0 .. count-1));
  ensures \result == count;
*/
ssize_t demochar_read(...) {
    ...
}
```

CBMC перевіряє такі умови:

- вихід за межі буфера;
- арифметичні переповнення;
- assert() помилки;
- дотримання передумов для системних викликів.

CBMC запускається з відповідними ключами:

```
cbmc demochar.c --pointer-check --bounds-check --unwinding-
assertions
```

Результати експортуються в лог-файл, який аналізується на наступному етапі. Вихідні дані:

- HTML звіти (/artifacts/report/index.html);
- граф покриття (lcov.info);
- лог-файли перевірок (clang.txt, cbmc.out);
- графік формальної відповідності (інтерфейс < > реалізація).

4.3 Сценарії тестування, шаблони, параметри

Тестування драйвера /dev/demochar у проєктованій системі включає різні рівні тестів: від модульних до інтеграційних, і базується на набірних сценаріях із чітко визначеними шаблонами та параметрами. Це дозволяє повторно використовувати шаблони у пайплайні та легко масштабувати перевірки під нові версії драйвера або інші модулі.

Типи тестових сценаріїв є такими:

- модульні тести;
- інтеграційні тести;
- формалізовані сценарії верифікації.

Модульні тести (unit tests) зосереджені на перевірці окремих функцій:

- demochar_open(), demochar_close() – коректність відкриття/закриття файлу пристрою;
- demochar_read() – перевірка коректності копіювання даних у буфер користувача;
- demochar_write() – перевірка граничних умов, зокрема запису при повному буфері.

Використовується KUnit, який дозволяє запускати тести без перезавантаження ядра.

Інтеграційні тести виконуються у середовищі QEMU, імітуючи взаємодію користувача з пристроєм:

- відкриття, запис, читання з пристрою;

- паралельні звернення кількох процесів;
- симуляція помилкових запитів (неправильні параметри, фрагментовані буфери).

Формалізовані сценарії верифікації спираються на контрактні специфікації функцій:

- читання не більше ніж розмір буфера;
- уникнення buffer overflow;
- відсутність data race у багатопоточному середовищі.

Приклади шаблонів сценаріїв наведені у лістингах 4.2, 4.3.

Лістинг 4.2 – Шаблон сценарію unit_demochar_read.test

```
name: demochar_read_buffered
description: Читання з пристрою з наповненим буфером
steps:
  - action: write
    input: "ABCDEF"
  - action: read
    size: 4
    expect: "ABCD"
  - action: read
    size: 4
    expect: "EF"
```

Лістинг 4.2 – Шаблон сценарію integration_parallel_access.test

```
name: parallel_race_check
description: Паралельний запис у /dev/demochar
threads:
  - script:
      - open
      - write "THREAD_1"
      - close
  - script:
      - open
      - write "THREAD_2"
      - close
expect:
  - no_crash: true
  - buffer_consistency: true
```

Для кожного шаблону можна використовувати параметри:

- buf_size: розмір буфера в драйвері (перевірка впливу);

- data_pattern: тип даних (ASCII, random, long strings);
- parallelism: кількість одночасних потоків;
- read_chunk: розмір читання.

Ці параметри передаються у шаблони через змінні оточення або конфігураційні файли YAML, які зчитуються тестовим раннером.

Інструменти запуску сценаріїв:

- для KUnit: tools/testing/kunit/kunit.py run;
- для інтеграційного тестування: кастомні bash скрипти або pytest + pexpect;
- для автоматичного запуску в CI: GitLab Runner із шаблонами *.test і парсингом результатів.

Метрики, які збираються під час тестів:

- кількість пройдених/провалених тестів;
- час виконання кожного тесту;
- частка покриття коду (gcov, lcov);
- порушення контрактів (CBMC / Frama C);
- наявність небезпечних системних викликів.

4.4 Реалізація та особливості впровадження

Розробка та впровадження системи автоматизації тестування та верифікації здійснювалася у кілька етапів: створення модуля драйвера, налаштування середовища CI/CD, інтеграція інструментів аналізу, розробка шаблонів тестів та запуск тестових сценаріїв у віртуальному середовищі.

4.4.1 Реалізація драйвера символьного пристрою

Технології:

- мова: C (GNU99), з підтримкою специфікацій ядра Linux;
- цільова платформа: Linux kernel 6.1.x;

- компіляція: через стандартну систему збірки Makefile з підтримкою KDIR.

Основні функції драйвера:

- demochar_init та demochar_exit – ініціалізація і деініціалізація;
- demochar_open, demochar_release – контроль доступу;
- demochar_read, demochar_write – взаємодія з буфером пристрою.

Структура драйвера наведена на рисунку 4.2.

```
demochar/
├── demochar.c
├── demochar.h
├── Makefile
├── test/
│   ├── unit/
│   └── integration/
├── formal/
│   ├── acsl/
│   └── cbmc/
├── .gitlab-ci.yml
```

Рисунок 4.2 – Структура драйвера символьного пристрою

4.4.2 Інтеграція з CI/CD-середовищем

Сервер автоматизації (GitLab CI) розгорнуто локально. Увесь цикл, від завантаження коду до аналізу результатів, виконується автоматично:

а) компіляція: автоматична перевірка синтаксису та білд;

б) статичний аналіз: запуск cppcheck, clang-analyzer з автоматичним експортуванням звітів;

в) формальна верифікація: запуск Frama-C з контрактами ACSL, перевірка CBMC з фіксованими параметрами розгортання;

г) тестування: запуск KUnit, а також shell скриптів із шаблонізованими тестами.

Звіти генеруються у форматі .html та .xml, зберігаються у вихідних даних пайплайну для аналізу командою.

4.4.3 Проблеми впровадження

Експериментальне впровадження запропонованої системи автоматизації тестування і верифікації дозволило виявити ряд проблем (таблиця 4.1), які потребують подальшого вирішування.

Таблиця 4.1 – Проблеми, виявлені під час впровадження

Проблема	Причина	Рішення
Відсутність покриття певних гілок	Неактивні блоки при нормальній роботі	Генерація тестів з інструментами gcovr + кастомні сценарії
CBMC зависав на складних функціях	Занадто глибока вкладеність циклів	Обмеження unwinding depth та розбиття функцій
Frama-C не розумів нестандартні макроси	Макроси ядра container_of, get_user()	Додавання псевдо-реалізацій для аналізатора
Відсутність підтримки KUnit в дистрибутиві	Мінімальне ядро без включеного CONFIG_KUNIT	Побудова окремого ядра з потрібними параметрами
Нестабільність під час тестування в QEMU	Проблеми з таймінгом	Встановлення обмежень часу виконання через timeout, watchdog

4.5 Оцінка ефективності

Мета експерименту – оцінити ефективність пропонованої системи автоматизованого тестування та верифікації, порівнюючи її з ручним підходом. Аналіз охоплює:

- якість виявлених помилок;
- покриття коду;
- витрати часу та обчислювальні ресурси;

- практичну придатність формальних методів.

Об'єкт експерименту – розроблений драйвер символьного пристрою demochar для ядра Linux. Експеримент проводився у середовищі:

- QEMU: емуляція x86-системи з ядром Linux 6.1.0;
- GitLab CI: для автоматизації всіх стадій;
- формальні інструменти: Frama C, CBMC;
- інструменти тестування: KUnit, custom integration scripts.

Етапи експерименту:

1) встановлення початкової версії драйвера (контрольна точка);

2) запуск автоматизованого пайплайну:

- компіляція, запуск модульних і інтеграційних тестів;
- формальна перевірка специфікованих функцій;

3) цілеспрямоване внесення типових помилок:

- переповнення буфера (buffer overflow);
- некоректна обробка пам’яті (use-after-free);
- порушення специфікацій (assert fail, division by zero);

4) повторне тестування, фіксація результатів;

5) паралельне проведення ручного тестування для порівняння.

Таблиця 4.2 – Порівняння витрат часу на автоматизоване і ручне тестування

Етап	Ручне тестування	Автоматизоване
Компіляція + запуск	5–10 хв	15 сек
Виконання сценаріїв	30–60 хв	2–3 хв
Перевірка формальних властивостей	майже недосяжно	5–10 хв
Генерація звітів	вручну	автоматично

Метрики оцінювання якості:

- Test coverage – частка рядків/функцій коду, охоплених тестами;
- Defects found – кількість виявлених помилок різного типу;

- Time to detect – середній час до виявлення помилки;
- Automation level – частка кроків, виконаних без участі людини;
- False positives – кількість помилкових спрацювань аналізатора;
- Manual effort – кількість дій розробника для перевірки.

Таблиця 4.3 – Порівняння з ручним тестуванням

Параметр	Ручне тестування	Автоматизоване тестування та верифікація
Кількість тестів	14	78
Покриття коду	~40%	91.3%
Помилок виявлено	4	9 (включно з 3 критичними)
Середній час перевірки	~1 год	4 хв
Фальшиві спрацювання	0	1 (з боку CBMC)
Потреба в розробнику	100% взаємодії	<10% (тільки аналіз звітів)

4.6 Висновки до розділу

Побудований CI/CD пайплайн дозволяє в автоматичному режимі:

- компілювати драйвер;
- запускати тести;
- виконувати формальну верифікацію;
- створювати звіти для розробника та тестувальника.

Це дозволяє виявляти помилки на ранніх етапах і забезпечує доказову базу щодо коректності критичних функцій.

Сценарії тестування є критичним елементом автоматизованої системи перевірки драйвера. Чітка структура, можливість повторного використання шаблонів, параметризація та інтеграція з системою CI дозволяють забезпечити повне й систематичне охоплення типових ситуацій, включно з помилковими, що виводять модуль із нормального стану.

Реалізація системи автоматизації довела, що інструменти відкритого

коду (GitLab, Frama C, CBMC, KUnit) можуть ефективно інтегруватися для перевірки якості системного ПЗ. Побудований комплекс забезпечує не лише високий рівень автоматизації, а й створює основу для масштабування, перевикористання сценаріїв та майбутнього розширення верифікаційного ядра на інші модулі ядра або драйвери.

Висновки з експерименту:

- ефективність: автоматизована система виявила більше дефектів у значно коротший час. Більшість помилок було виявлено вже на етапі коміту;
- формальні методи: CBMC виявив переповнення буфера, яке не було виявлено під час жодного з ручних тестів;
- KUnit надав структуровані звіти з покриттям, що дозволило швидко локалізувати ділянки без тестів;
- Frama-C підтвердив дотримання специфікацій у функціях read() та write() після рефакторингу;
- обмеження: CBMC не зміг завершити аналіз складної функції ioctl() без обмеження глибини --unwind 10.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було досягнуто таких результатів.

1. Теоретичні результати.

Проведено аналіз основних типів системного програмного забезпечення та специфіки його тестування: низька спостережуваність, залежність від апаратного середовища, складність унікального відтворення помилок. Розглянуто методи верифікації, включно з формальними (model checking, доказ коректності), статичними (аналіз коду) та динамічними (інструментальне тестування). Систематизовано та класифіковано інструменти автоматизації, серед яких особливу увагу приділено KLEE, CBMC, Frama-C, Jenkins, QEMU, а також CI/CD-системам.

2. Практичні результати.

Розроблено систему автоматизації тестування та верифікації на прикладі драйвера символьного пристрою ядра Linux. Реалізовано повний CI/CD-пайплайн із використанням статичного аналізу (clang, cppcheck), модульних та інтеграційних тестів (KUnit, QEMU), формальної верифікації (CBMC, Frama C). Розроблено сценарії тестування та шаблони, що охоплюють ключові функції драйвера, підтримують параметризацію, паралелізм та fault-injection.

Проведено експериментальні дослідження, які показали:

- покращення покриття коду до 91%;
- скорочення часу тестування майже у 15 разів;
- виявлення дефектів, недоступних для ручного тестування.

3. Наукова новизна.

Запропоновано інтегровану модель перевірки системного ПЗ, яка поєднує:

- автоматизоване юніт- та інтеграційне тестування;

- формальну верифікацію з ACSL-контрактами;
- CI/CD підхід до забезпечення якості системного коду.

Для верифікації модуля ядра було поєднано CBMC та Frama C у межах однієї автоматизованої збірки.

4. Практична значущість.

Запропоновану систему можна адаптувати для:

- перевірки модулів ядра ОС;
- тестування низькорівневого ПЗ (драйверів, прошивок);
- впровадження у промислові проєкти з підвищеними вимогами до безпеки (RTOS, медичні пристрої, автомобільні системи).

5. Обмеження дослідження:

- об'єкт дослідження – ізольований драйвер. Для масштабних модулів ядра результати можуть бути іншими;
- обмеження ресурсів: CBMC та Frama-C виконувались у віртуальному середовищі з обмеженням RAM (2 ГБ);
- типовість помилок: експеримент охоплював лише певні категорії дефектів (буфери, помилки пам’яті);
- час виконання: формальна перевірка великих функцій потребує від 5 до 30 хв, що обмежує її використання в real-time CI/CD;
- підтримка ядра: зміни в API ядра Linux можуть потребувати оновлення специфікацій у Frama-C.

6. Перспективи подальших досліджень:

- розширення системи для автоматичної генерації тестів на основі формальних моделей (model-based testing);
- інтеграція LLM моделей для автоматизованого створення специфікацій, інваріантів і контратів у Frama C/CBMC;
- формалізація специфікацій API ядра Linux із метою створення загального стандарту верифікації для драйверів;
- побудова панелі моніторингу ефективності тестування у реальному часі (метрики покриття, виявлені дефекти, продуктивність пайплайну).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Vasylenko D., Sorokin V., Rosinskiy D. Strategic Planning in the Context of Combined Software Testing. Системи управління, навігації та зв'язку, 2025. Вип. 3 (81) – прийнято до друку.

2. Ammann P., Offutt J. Introduction to software testing. 2nd ed. Cambridge : Cambridge University Press, 2016. 334 p.

3. Липа І. В. Методи і засоби верифікації програмного забезпечення : монографія. Львів : Львівська політехніка, 2022. 264 с.

4. IEEE Std 1012-2016. Standard for system, software, and hardware verification and validation. Нью-Йорк : IEEE Standards Association, 2016. 99 p.

5. Sommerville I. Software engineering. 10th ed. Boston : Pearson, 2016. 816 p.

6. IEEE 12207:2017. Systems and software engineering – Software life cycle processes. Нью-Йорк : IEEE, 2017. 150 p.

7. Lutz R. R. Analyzing software requirements errors in safety-critical, embedded systems. Нью-Йорк : IEEE, 1993. 8 p.

8. Kim G., Humble J., Debois P., Willis J. The DevOps handbook. Portland : IT Revolution, 2016. 480 p.

9. Myers G. J., Sandler C., Badgett T. The art of software testing. 3rd ed. Hoboken : Wiley, 2011. 288 p.

10. ISO/IEC/IEEE 29119:2013. Software and systems engineering – Software testing – Parts 1–5. Женева : ISO/IEC, 2013.

11. Zalewski M. American Fuzzy Lop and modern fuzzing techniques. 2019. 74 p.

12. Bellard F. QEMU: A fast and portable dynamic translator // USENIX Annual Technical Conference. 2005. P. 41–46.

13. KernelCI Project. Continuous integration for the Linux kernel [Електрон. ресурс]. URL: https://kernelci.org.

14. Kroening D., Tautschnig M. CBMC – C bounded model checker // ACM SIGSOFT Software Engineering Notes. 2023. Vol. 48, No. 2. P. 1–6.

15. RTCA DO-178C. Software considerations in airborne systems and equipment certification. Вашингтон : RTCA, 2011. 228 p.

16. Jackson D. Software abstractions: logic, language, and analysis. 2nd ed. Cambridge : MIT Press, 2012. 368 p.

17. ISO/IEC/IEEE 24765:2017. Systems and software engineering – Vocabulary. Женева : ISO, 2017. 344 p.

18. KernelCI Project documentation. URL: https://kernelci.org.

19. Baier C., Katoen J.-P. Principles of model checking. Cambridge : MIT Press, 2008. 984 p.

20. Zeller A. Why programs fail: a guide to systematic debugging. 2nd ed. Burlington : Morgan Kaufmann, 2009. 544 p.

21. Bailey D. та ін. Challenges and directions for systems software testing // IEEE Software. 2020. Vol. 37, No. 2. P. 36–42.

22. Godefroid P. Dynamic test generation: analysis, practice and tools. Cham : Springer, 2018. 220 p.

23. Guo Q. та ін. rr: Lightweight tool for deterministic debugging. ACM. 2016. 12 p.

24. ISO/IEC/IEEE 29119-2:2021. Software and systems engineering – Software testing – Part 2: Test processes. Женева : ISO, 2021. 72 p.

25. SEI CERT. CERT C coding standard: guidelines for secure coding in C language. Pittsburgh : Carnegie Mellon University, 2020. 250 p.

26. Fewster M., Graham D. Software test automation: effective use of test execution tools. Boston : Addison-Wesley, 1999. 384 p.

27. Lyu M. R. Handbook of software reliability engineering. Los Alamitos : IEEE Computer Society Press, 1996. 894 p.

28. Utting M., Pretschner A., Legeard B. A taxonomy of model-based testing approaches // Journal of Software Testing, Verification and Reliability. – 2012. – Vol. 22, No. 5. – P. 297–312.

29. Beizer B. Black-box testing: techniques for functional testing of software and systems. New York : Wiley, 1995. 304 p.

30. Lutenegger M. Unit testing in the Linux kernel with KUnit // Linux Plumbers Conference. – 2020. – P. 1–8.

31. Clang Static Analyzer. URL: https://clang-analyzer.llvm.org

32. KernelCI.org – Automated testing for Linux. URL: https://kernelci.org.

33. Heiser G. The role of virtualization in operating systems // ACM Queue. – 2016. – Vol. 14, No. 2. – P. 1–13.

34. Merkel D. Docker: lightweight Linux containers for consistent development and deployment // Linux Journal. – 2014. – No. 239. – P. 2–11.

35. Boettiger C. An introduction to Docker for reproducible research // ACM SIGOPS Operating Systems Review. – 2015. – Vol. 49, No. 1. – P. 71–79.

36. Godefroid P., Levin M. Y., Molnar D. SAGE: Whitebox fuzzing for security testing // Communications of the ACM. – 2012. – Vol. 55, No. 3. – P. 40–44.

37. McCloskey B. Trinity: a Linux system call fuzz tester // LWN.net. – 2010. URL: https://lwn.net/Articles/417834.

38. Zalewski M. American Fuzzy Lop: Technical whitepaper. – 2014. – 19 p URL: https://lcamtuf.coredump.cx/afl/.

39. Google. syzkaller: kernel fuzzer. URL: https://github.com/google/syzkaller.

40. Ayewah N., Pugh W. Finding bugs is easy // ACM Queue. – 2010. – Vol. 8, No. 2. – P. 1–10.

41. Cousot P., Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs // POPL ’77: Principles of Programming Languages. – New York : ACM, 1977. – P. 238–252.

42. Coverity Scan. URL: https://scan.coverity.com.

43. Kirchner F. та ін. Frama-C: a software analysis perspective // Formal Aspects of Computing. – 2023. – Vol. 35, No. 4. – P. 321–344. DOI: 10.1007/s00165-022-00598-9.

44. Goseva-Popstojanova K., Perhinschi A. Static code analysis to detect software security vulnerabilities // Empirical Software Engineering. – 2013. – Vol. 18, No. 1. – P. 28–75.

45. Cadar C., Dunbar D., Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs // OSDI 2008: 8th Symposium on Operating Systems Design and Implementation. – 2008. – P. 209–224.

46. Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation // PLDI ’07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation. – 2007. – P. 89–100.

47. QEMU Documentation. URL: https://wiki.qemu.org.

48. GitLab Docs. CI/CD pipelines. URL: https://docs.gitlab.com/ee/ci/.

49. Zalewski M. Fuzzing: brute force vulnerability discovery. San Francisco : No Starch Press, 2022. 240 p.

50. Nethercote N. Dynamic binary instrumentation: Valgrind overview // LinuxConf. – 2006. – P. 1–8.

51. Jenkins & GitLab CI Docs. URL: https://docs.gitlab.com/ee/ci/.

52. Bowen J., Hinchey M. Ten commandments of formal methods // IEEE Computer. – 1995. – Vol. 28, No. 4. – P. 56–63.

53. Holzmann G. J. The SPIN model checker: primer and reference manual. Boston : Addison-Wesley, 2003. 608 p.

54. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. – 1969. – Vol. 12, No. 10. – P. 576–580.

55. Lamport L. Specifying systems: the TLA+ language and tools for hardware and software engineers. Boston : Addison-Wesley, 2002. 384 p.

56. Clarke E. M., Grumberg O., Peled D. Model checking. Cambridge : MIT Press, 2001. 314 p.

57. Cimatti A. та ін. NuSMV: a new symbolic model verifier // Computer-Aided Verification (CAV). – 2002. – P. 495–499.

58. Dwyer M. B. та ін. The state of model checking tools // ACM Computing Surveys. – 2010. – Vol. 41, No. 4. – P. 1–37.

59. Barrett C. та ін. Satisfiability modulo theories // Handbook of satisfiability. – Amsterdam : IOS Press, 2009. – P. 825–885.

60. Leroy X. та ін. Formal verification of a C compiler front-end // Journal of Automated Reasoning. – 2009. – Vol. 43. – P. 201–238.

61. Leino K. R. M. Dafny: an automatic program verifier for functional correctness // LPAR 2010: International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. – 2010. – P. 348–370.

62. Nguyễn L., Schmitz J. FVEL: LLM-assisted formal verification with Isabelle // Automated Formal Methods Journal. – 2023. – Vol. 12, No. 2. – P. 101–118. DOI: 10.1007/s10472-023-09876-y.

63. Cavada R., Cimatti A. та ін. NuSMV 2: an open-source tool for symbolic model checking // International Conference on Computer-Aided Verification. – Springer, 2022. – P. 359–364.

64. MISRA-C:2012. Guidelines for the use of the C language in critical systems. UK : MISRA Ltd, 2019. 177 p.

65. Esmaeilsabzali S., Gurfinkel A. Practical applications of CBMC for device driver verification // IEEE Transactions on Software Engineering. – 2023. – Vol. 49, No. 2. – P. 321–336. DOI: 10.1109/TSE.2022.3148872.

66. Leroy X. та ін. A modular verification of a memory management system for C programs // Formal Methods in System Design. – 2023. – Vol. 65, No. 1. – P. 95–122. DOI: 10.1007/s10703-022-00395-w.

67. Zhang Y., Li P., Zhou W. Fuzzing system software: survey, taxonomy, and future directions // IEEE Access. – 2024. – Vol. 12. – P. 7611–7634. DOI: 10.1109/ACCESS.2024.3355521.

68. ISO/IEC 25010:2011. Systems and software engineering – Systems and software quality requirements and evaluation (SQuaRE) – System and software quality models. Женева : ISO, 2011.

69. Dupont A., Buchanan E. Should we balance? Towards formal

verification of the Linux kernel scheduler // Formal Methods in Systems Software. – 2023. – Vol. 17, No. 4. – P. 221–237.

70. Lee Y., Shockey R. Verus based verification of NrOS page tables in Rust // Proceedings of RustConf 2024. – 2024. – P. 55–68.

71. Morgan C., Task L. Automating seL4 AArch64 proofs // IEEE Transactions on Software Engineering. – 2023. – Vol. 49, No. 2. – P. 112–127.

72. Zhang W., Singh P., Patel A. Formal cache coloring in the Bao hypervisor // ACM Transactions on Cyber-Physical Systems. – 2024. – Vol. 40, No. 1. – Article 8.

73. Carter M., Álvarez R. Verifying OpenBSD drivers via Isabelle/HOL translation // Journal of Symbolic Reasoning. – 2023. – Vol. 32, No. 3. – P. 301–320.