ПОЛТАВСЬКИЙ ДЕРЖАВНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
Навчально-науковий інститут економіки, управління, права та інформаційних технологій
Кафедра інформаційних систем та технологій

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти бакалавр

на тему: **«Вдосконалення процедури кодування та верифікації проєктів VHDL із використанням інструментів HDL Designer та Scientific Toolworks Understand»**

Виконав: здобувач вищої освіти
за освітньою програмою
Інформаційні управляючі системи
спеціальності 126 Інформаційні системи та технології
ступеня вищої освіти бакалавр
групи 126ІСТ_бд_2022
Вернигора Антон Олександрович
Керівник: Одарущенко Олег Миколайович
Рецензент: Стрюк Олексій Юрійович

Полтава – 2026 року

ПОЛТАВСЬКИЙ ДЕРЖАВНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
Навчально-науковий інститут економіки, управління, права та інформаційних технологій
Кафедра інформаційних систем та технологій

Освітня програма Інформаційні управляючі системи
Спеціальність 126 Інформаційні системи та технології
Рівень вищої освіти перший (бакалаврський)

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ Юрій УТКІН

«10» червня 2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА ВИЩОЇ ОСВІТИ

Вернигори Антона Олександровича

1. Тема кваліфікаційної роботи:
«Вдосконалення процедури кодування та верифікації проєктів VHDL із використанням інструментів HDL Designer та Scientific Toolworks Understand»,
Керівник роботи: д. т. н., професор, професор кафедри інформаційних систем та технологій Одарущенко О. М.
Затверджено наказом закладу вищої освіти від «10» квітня 2026 року № 384-ст

2. Строк подання здобувачем вищої освіти роботи «08» червня 2026 року.

3. Вихідні дані до роботи: стандарти і специфікації офіційної платформи W3C https://www.w3.org/standards/, офіційні вебсайти інтегрованих середовищ розробки ПЗ https://code.visualstudio.com/, https://www.figma.com/design/, статистичні дані аналітичних компаній Work.ua, W3Techs

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Розділ 1 Аналіз підходів до розробки та верифікації VHDL-проєктів
Розділ 2. Опис удосконаленої процедури статичного аналізу та огляду vhdl-проєктів
Розділ 3. Програмна реалізація інтерактивного вебдодатку з пошуку квитків та аналіз ефективності застосування

5. Перелік графічного матеріалу: схеми, рисунки, діаграми за темою та об’єктом дослідження

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання отримав
Оцінювання економічної ефективності результатів дослідження	Загребельна І. Л., к. е. н., доцент, доцент кафедри економіки та публічного управління	01.04.2026 р.	29.05.2026 р.

7. Дата видачі завдання «10» червня 2025 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Вибір і затвердження теми роботи	02.06.2025 р. – 04.06.2025	
2	Складання і затвердження розгорнутого плану та завдання на кваліфікаційну роботу	05.06.2025 р. – 10.06.2025 р.	
3	Опрацювання джерел інформації	11.06.2025 р. – 15.06.2025 р.	
4	Збір, вивчення і обробка інформації, необхідної для виконання роботи	16.06.2025 р. – 20.06.2025 р.	
5	Виконання теоретико-методологічного розділу роботи	08.09.2025 р. – 01.11.2025 р.	
6	Виконання дослідницько-аналітичного розділу роботи	19.01.2026 р. – 14.02.2026 р.	
7	Виконання проєктно-рекомендаційного розділу роботи	16.02.2026 р. – 31.03.2026 р.	
8	Оцінювання економічної ефективності результатів дослідження	01.04.2026 р. – 29.05.2026 р.	
9	Оформлення тексту роботи	01.06.2026 р. – 06.06.2026р.	
10	Попередній захист роботи на кафедрі	08.06.2026 р.	
11	Доопрацювання роботи з урахуванням зауважень і пропозицій	09.06.2026 р. – 10.06.2026 р.	
12	Нормоконтроль	11.06.2026 р. – 15.06.2026 р.	
13	Захист кваліфікаційної роботи	16.06.2026 р. – 18.06.2026 р.	

Здобувач вищої освіти **Антон ВЕРНИГОРА**

Керівник роботи **Олег ОДАРУЩЕНКО**

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

ASIC – Application-specific integrated circuit

EDA – Electronic design automation

FPGA – Field-programmable gate array

HDL – Hardware description language

IDE – Integrated development environment

IEEE – Institute of Electrical and Electronics Engineers

RTL – Register transfer level

VHDL – VHSIC hardware description language

VHSIC – Very high speed integrated circuits

ВСТУП

У сучасному світі цифрові технології розвиваються дуже швидкими темпами, що суттєво впливає на створення та вдосконалення електронних систем. Особливо важливими є системи, які працюють на апаратному рівні обробки даних, адже саме вони лежать в основі більшості сучасних пристроїв – від вбудованих систем до складних обчислювальних комплексів. У зв’язку з цим постійно зростає складність цифрових пристроїв, а також вимоги до їх проєктування.

Для розробки таких систем широко використовуються мови опису апаратури (HDL), серед яких однією з найпоширеніших є VHDL (VHSIC Hardware Description Language). Ця мова дозволяє описувати як структуру, так і поведінку цифрових систем, що дає змогу моделювати їх роботу ще до фізичної реалізації. Це значно спрощує процес розробки та дозволяє виявляти помилки на ранніх етапах.

Разом із тим, зі збільшенням складності систем виникають і певні проблеми. На практиці розробники часто стикаються зі складністю відстеження зв’язків між модулями, недостатньою структурованістю коду, а також із тим, що помилки виявляються вже на пізніх етапах розробки. Це призводить до збільшення часу розробки та ускладнює процес виправлення помилок.

Ще однією проблемою є те, що процес аналізу та перевірки VHDL-коду не завжди є достатньо формалізованим. Часто перевірка виконується вручну або обмежується лише симуляцією, що не дозволяє повністю оцінити якість коду та його структуру.

Саме тому виникає потреба у використанні сучасних інструментів, які дозволяють автоматизувати процес розробки, аналізу та верифікації VHDL-проєктів. До таких інструментів можна віднести HDL Designer, який забезпечує зручне проєктування структури системи, а також Scientific Toolworks Understand, що дозволяє виконувати статичний аналіз коду та оцінювати його якість.

Актуальність теми полягає у необхідності вдосконалення існуючих підходів до кодування та верифікації VHDL-проєктів шляхом використання сучасних

інструментів. Це дозволяє зменшити кількість помилок, підвищити якість коду та зробити процес розробки більш ефективним.

Метою роботи є аналіз існуючих підходів до розробки VHDL-проєктів та розробка вдосконаленої процедури кодування і верифікації з використанням інструментів HDL Designer та Scientific Toolworks Understand.

Для досягнення поставленої мети необхідно вирішити такі *завдання*:

1) проаналізувати особливості мови VHDL та її використання;
2) дослідити існуючі підходи до розробки та верифікації;
3) визначити основні проблеми традиційного підходу;
4) обґрунтувати вибір інструментів розробки;
5) реалізувати VHDL-проєкт;
6) виконати його аналіз та оцінити якість;
7) запропонувати вдосконалену процедуру кодування та верифікації.

Об'єктом дослідження є процес розробки цифрових систем із використанням мови VHDL.

Предметом дослідження є методи та інструменти кодування і верифікації VHDL-проєктів.

Методи дослідження включають аналіз, порівняння, моделювання та практичну реалізацію розробленого підходу.

Практична значущість роботи полягає у створенні більш ефективного підходу до розробки VHDL-проєктів, який дозволяє підвищити якість коду та скоротити час розробки.

Результати роботи апробовані в рамках наукової конференції здобувачів вищої освіти за результатами науково-дослідної роботи у 2025-2026 рр.

Структура кваліфікаційної роботи логічно пов'язана з задачами досліджень і містить перелік умовних позначень, вступ, три розділи основної частини, висновки, список використаних джерел. Загальний обсяг текстової частини кваліфікаційної роботи складає 54 сторінки формату А4. Вона містить 9 рисунків і 13 таблиць.

РОЗДІЛ 1
АНАЛІЗ ПІДХОДІВ ДО РОЗРОБКИ ТА ВЕРИФІКАЦІЇ VHDL-ПРОЄКТІВ

1.1 Основи мови VHDL, цифрових автоматів та області застосування

Мова опису апаратури VHDL, або Very High-Speed Integrated Circuit Hardware Description Language, є однією з базових технологій сучасного проєктування цифрових систем. Вона використовується для формального опису, моделювання, синтезу, тестування та супроводження цифрової апаратури. Відповідно до стандарту IEEE 1076, VHDL є формальною мовою, призначеною для використання на різних етапах створення електронних систем, включаючи розробку, верифікацію, синтез і тестування апаратних проєктів [1].

На відміну від традиційних мов програмування, які описують послідовність інструкцій для виконання процесором, VHDL призначена для опису апаратної логіки. У цифрових схемах різні функціональні блоки можуть працювати паралельно, тому мова VHDL підтримує конкурентну модель опису, яка краще відповідає фізичній природі електронних пристроїв. Це дає змогу описувати не лише алгоритм обробки даних, а й структуру взаємодії сигналів, модулів, регістрів, комбінаційної та послідовної логіки [1, 2].

Основними структурними елементами VHDL-проєкту є entity та architecture. Конструкція entity визначає зовнішній інтерфейс модуля, тобто перелік його вхідних, вихідних і двонаправлених портів. Конструкція architecture описує внутрішню реалізацію модуля: сигнали, процеси, компоненти, оператори присвоєння та логіку взаємодії між елементами схеми. Такий поділ дозволяє відокремити інтерфейс цифрового пристрою від його внутрішньої поведінки, що спрощує аналіз, повторне використання та подальшу верифікацію VHDL-коду [2, 3].

До основних елементів VHDL-коду належать сигнали, змінні, процеси, паралельні оператори, компоненти, типи даних і бібліотеки. Сигнали використовуються для передавання значень між різними частинами цифрової

схеми, змінні застосовуються переважно всередині процесів, а процеси описують поведінку логіки залежно від зміни вхідних сигналів або тактового імпульсу. Саме за допомогою процесів у VHDL часто реалізуються регістри, лічильники, контролери інтерфейсів та цифрові автомати [2, 3].

Одним із типових об’єктів опису мовою VHDL є цифровий автомат, або скінченний автомат станів. Цифровий автомат являє собою модель керуючого пристрою, який може перебувати в одному з наперед визначених станів і переходити між ними залежно від поточного стану, вхідних сигналів та умов переходу. У нормативних і методичних матеріалах із розробки FPGA-проєктів скінченний автомат визначається як обчислювальна модель із кінцевою кількістю станів, серед яких один визначається як початковий [3]. У VHDL такі автомати зазвичай реалізуються через перелічувані типи станів, сигнали поточного та наступного стану, синхронний процес зміни стану і комбінаційну логіку формування переходів.

Цифрові автомати широко застосовуються у FPGA-проєктах, оскільки вони дозволяють формалізувати керуючу логіку апаратних пристроїв. Вони можуть використовуватися для керування послідовністю ініціалізації пристрою, обміном даними між інтерфейсами, доступом до пам’яті, обробкою команд, перемиканням режимів роботи, формуванням сигналів дозволу або реакцією на аварійні події. У цьому контексті VHDL-код визначає не лише програмну поведінку, а й майбутню апаратну структуру, яка після синтезу реалізується у FPGA або ASIC [3, 4].

Для опису цифрових автоматів у VHDL часто використовується оператор case, який дозволяє задати поведінку системи для кожного окремого стану. При цьому важливо, щоб усі можливі стани були коректно описані, початковий стан був явно визначений, а переходи між станами не містили недосяжних або логічно суперечливих гілок. Неповний опис оператора case, відсутність reset-сигналу або наявність недосяжних станів може призвести до некоректної роботи FPGA-пристрою після синтезу. Саме тому в правилах кодування VHDL окремо виділяються вимоги до перевірки скінченних автоматів, зокрема щодо наявності початкового стану, reset-сигналу та відсутності недосяжних станів [3].

VHDL підтримує різні рівні абстракції опису цифрових систем. На поведінковому рівні основна увага приділяється функціональній логіці пристрою без детального опису його фізичної структури. На структурному рівні система подається як сукупність взаємопов’язаних компонентів. Найбільш поширеним у практиці FPGA-розробки є рівень регістрових передач, або RTL, який дозволяє описувати передачу даних між регістрами, комбінаційну логіку та синхронну поведінку цифрового пристрою. RTL-опис є основою для подальшого синтезу апаратної реалізації [2, 4].

Важливою перевагою VHDL є строга типізація, що дозволяє зменшити кількість помилок, пов’язаних із некоректним використанням сигналів, розрядностей і типів даних. У VHDL-проєктах широко використовуються типи std_logic та std_logic_vector, які дають змогу описувати окремі цифрові сигнали та багаторозрядні шини. Для арифметичних операцій доцільним є використання стандартної бібліотеки IEEE.NUMERIC_STD, оскільки вона забезпечує передбачувану роботу з числовими типами та відповідає сучасній практиці написання синтезованого VHDL-коду [1, 3].

Область застосування VHDL охоплює широкий спектр цифрових систем. Мова використовується при розробці FPGA-пристроїв, ASIC-мікросхем, вбудованих контролерів, інтерфейсних модулів, систем цифрової обробки сигналів, комунікаційного обладнання, пристроїв промислової автоматики та систем керування. Особливе значення VHDL має у тих випадках, коли необхідні висока швидкодія, паралельна обробка сигналів, детермінована поведінка та можливість реалізації алгоритмів безпосередньо на апаратному рівні [2, 4].

Окремою сферою застосування VHDL та FPGA є критичні системи, у яких цифрова логіка повинна працювати передбачувано, надійно та відповідно до встановлених вимог безпеки. До таких систем належать авіаційні, транспортні, промислові та енергетичні системи, зокрема системи безпеки атомних електростанцій. Документ NUREG/CR-7006 розглядає практики безпечного проєктування FPGA та може використовуватися як основа для перевірки FPGA-based safety systems у ядерній галузі [5].

Вимоги до HDL-програмованих пристроїв у системах керування атомних електростанцій також визначаються стандартом IEC 62566. У ньому зазначено, що HDL-programmed devices можуть застосовуватися в I&C-системах атомних електростанцій, які виконують функції категорії безпеки A, а їх розробка базується на мовах опису апаратури та відповідних інструментальних засобах [6].

Практичним прикладом використання FPGA у ядерній галузі є платформа RadICS, яка позиціонується як цифрова safety I&C platform для атомної енергетики. Згідно з офіційними матеріалами, RadICS сертифікована до рівня SIL 3 в одноканальній конфігурації та була схвалена U.S. Nuclear Regulatory Commission для використання у safety-related systems на атомних об’єктах США [7]. Це підтверджує актуальність застосування VHDL та FPGA не лише в навчальних або промислових проєктах загального призначення, а й у системах, де якість цифрової логіки безпосередньо пов’язана з вимогами функційної безпеки.

VHDL є ефективним засобом опису цифрової апаратури, який дозволяє створювати як прості логічні модулі, так і складні FPGA-системи з цифровими автоматами, ієрархічною структурою та формалізованою поведінкою. Водночас зі зростанням складності таких проєктів підвищуються вимоги до якості коду, правильності опису станів, повноти переходів, синхронізації та верифікації. Саме тому для VHDL-проєктів, особливо у критичних сферах застосування, необхідним є використання формалізованих правил кодування, статичного аналізу та процедур code review [3, 5, 6].

1.2 Застосування VHDL та FPGA у критичних системах, зокрема системах безпеки AEC

Сучасні цифрові системи керування дедалі частіше використовують програмовані логічні інтегральні схеми FPGA для реалізації апаратної логіки, контролю сигналів, обробки подій та виконання функцій захисту. На відміну від програмного забезпечення, що виконується процесором, FPGA реалізує логіку

безпосередньо на апаратному рівні. Тому VHDL-код у таких системах фактично визначає структуру майбутнього цифрового пристрою, а помилки у коді можуть проявитися як помилки апаратної поведінки після синтезу [3].

Особливо важливим є застосування FPGA у критичних системах, до яких належать системи керування транспортом, промислові комплекси, авіаційні системи, енергетичне обладнання та системи безпеки атомних електростанцій. У таких системах цифрова логіка повинна мати передбачувану поведінку, стійкість до відмов, контрольовані переходи між режимами роботи та можливість перевірки на всіх етапах життєвого циклу. Для цього недостатньо лише написати працездатний VHDL-код; необхідно забезпечити його відповідність правилам кодування, вимогам верифікації, аналізу та документування [5, 8].

У системах безпеки атомних електростанцій FPGA може використовуватися для реалізації логіки аварійного захисту, блокувань, пріоритетної обробки сигналів, контролю стану обладнання, мажоритарного голосування та формування команд переходу у безпечний стан. Такі функції часто мають дискретну логіку роботи, тобто система перебуває в одному з визначених станів і переходить до іншого стану залежно від вхідних сигналів або аварійних умов. Саме тому цифрові автомати є типовим елементом VHDL-проєктів у критичних застосуваннях [3, 6].

Документ NUREG/CR-7006 присвячений рекомендаціям щодо аналізу FPGA-based safety systems для атомних електростанцій. У ньому FPGA-проєктування розглядається з позицій безпечних практик розробки, методів опису цифрової логіки, трасованості, надійності, робастності та супроводжуваності. Це підтверджує, що для FPGA-систем у ядерній галузі важливими є не тільки результати функційного тестування, але й якість самої процедури розробки, структура коду, обґрунтованість проєктних рішень та можливість незалежної перевірки [5].

Окремі вимоги до HDL-програмованих пристроїв у системах керування атомних електростанцій визначає стандарт IEC 62566. Він стосується розробки HDL-programmed devices для систем контролю та керування, важливих для безпеки, зокрема для функцій категорії А. У контексті цієї роботи це важливо тому,

що стандарт прямо пов’язує такі пристрої з мовами опису апаратури та інструментами розробки, а отже VHDL-код має розглядатися як критичний елемент життєвого циклу FPGA-пристрою [6].

Практичним прикладом використання FPGA у ядерній галузі є платформа RadICS. В офіційних матеріалах RadICS описується як цифрова safety I&C platform для атомної енергетики, що базується на FPGA-технології. Платформа призначена для побудови систем безпеки та керування на атомних електростанціях і дослідницьких реакторах. У її складі логічний модуль виконує функції керування каналом, а FPGA використовується як обчислювальний елемент для реалізації платформної логіки та користувацької прикладної логіки [7].

Цей приклад демонструє, що VHDL та FPGA застосовуються не лише у навчальних або загальнопромислових цифрових пристроях, а й у системах, де некоректна поведінка логіки може мати суттєві наслідки. Наприклад, якщо цифровий автомат у системі захисту не має чітко визначеного початкового стану, містить недосяжні стани або неповний оператор case, система може некоректно реагувати на аварійну подію чи перезапуск. Тому правила кодування для VHDL-проєктів повинні прямо враховувати особливості цифрових автоматів, reset-логіки, синхронізації, повноти переходів і керованості структури [3, 5].

З погляду процедури розробки, критичні FPGA-системи потребують поєднання кількох видів перевірки. Функційне тестування дозволяє перевірити поведінку системи на заданих сценаріях, однак воно не завжди виявляє структурні дефекти коду, недосяжні гілки, зайву складність або приховані проблеми синтезу. Тому поряд із тестуванням необхідно застосовувати статичний аналіз, code review та формалізовані правила кодування. У документі з правилами VHDL-кодування зазначено, що перевірка правил спрямована на виключення потенційних проблем FPGA electronic designs, підтримку механізмів виявлення та виправлення помилок, а також покращення читабельності й придатності коду до review [3].

Застосування VHDL та FPGA у критичних системах, зокрема в системах безпеки AEC, обумовлює необхідність суворішого підходу до організації розробки. Для таких проєктів важливо не лише реалізувати потрібну функціональність, а й

довести, що код є структурованим, аналізованим, верифікованим і придатним до супроводження. Саме тому подальше вдосконалення процедури кодування та статичного аналізу VHDL-проєктів доцільно будувати навколо правил, що стосуються цифрових автоматів, повноти case-операторів, визначення початкового стану, reset-сигналів і зниження складності коду [3, 5, 6].

1.3 Недоліки класичного підходу до розробки та тестування VHDL-описів

Зі зростанням складності цифрових систем процес розробки VHDL-проєктів стає дедалі більш залежним не лише від знань розробника, а й від якості організації самої процедури кодування, аналізу та верифікації. Традиційний підхід до створення HDL-проєктів часто базується на ручному написанні коду, подальшій компіляції, симуляції та виправленні помилок після їх виявлення. Така модель може бути прийнятною для невеликих навчальних або експериментальних проєктів, однак для складних FPGA-систем вона має суттєві обмеження [3, 4].

Однією з основних недоліків є складність структури сучасних VHDL-проєктів. FPGA-проєкт може містити значну кількість взаємопов’язаних модулів, сигналів, компонентів, пакетів, процесів і цифрових автоматів. За відсутності єдиної структури файлів, узгоджених правил іменування та зрозумілої ієрархії модулів аналіз такого проєкту стає трудомістким. Розробник або рецензент витрачає значний час не лише на пошук помилок, а й на розуміння того, як саме організована логіка пристрою [3].

Особливо складною є перевірка цифрових автоматів. У VHDL-проєктах вони часто реалізуються через оператори case, сигнали поточного і наступного стану, reset-логіку та умови переходів. Якщо автомат має недосяжні стани, неповний опис переходів, неявно визначений початковий стан або відсутній reset-сигнал, така помилка може не проявитися під час базової симуляції, але вплинути на поведінку пристрою після синтезу. Саме тому у правилах кодування VHDL окремо

виділяються вимоги до відсутності недосяжних станів, визначення початкового стану, коректності reset-сигналу та повноти case-операторів [3].

Другою проблемою є обмеженість перевірки, що базується лише на симуляції. Симуляція дозволяє перевірити поведінку цифрової системи в межах заданих тестових сценаріїв, однак вона не гарантує виявлення всіх структурних або логічних дефектів. Якщо певний стан автомата, комбінація сигналів або гілка переходу не була охоплена тестом, помилка може залишитися невиявленою. Для критичних FPGA-систем такий підхід є недостатнім, оскільки перевірка має охоплювати не тільки функційну поведінку, а й якість структури коду, його читабельність, синтезованість і придатність до незалежного аналізу [5, 8].

Третьою проблемою є пізнє виявлення помилок. У традиційному процесі значна частина дефектів виявляється після компіляції, симуляції або синтезу. На цьому етапі виправлення помилки може вимагати не лише зміни окремого фрагмента VHDL-коду, а й перегляду архітектури модуля, інтерфейсів, тестових сценаріїв або залежних компонентів. Чим пізніше виявлено дефект, тим більше часу витрачається на його локалізацію, виправлення та повторну перевірку [4, 9].

Окрему проблему становить неоднорідність стилю кодування. Якщо різні модулі одного проєкту написані без єдиних правил іменування, форматування, опису портів, reset-логіки, коментарів і структури процесів, це ускладнює code review та подальший супровід. Наприклад, відсутність змістовних назв сигналів, різний порядок оголошення портів, використання нестандартних пакетів або надмірна вкладеність умовних операторів підвищують ризик помилок і знижують прозорість коду [3, 4].

Ще одним обмеженням традиційної процедури є недостатнє використання статичного аналізу. У багатьох випадках перевірка VHDL-коду зводиться до синтаксичної коректності, симуляції та ручного перегляду. Однак статичний аналіз дозволяє виявляти проблеми без виконання моделі: надмірну складність, невикористані або некоректно пов’язані елементи, порушення правил кодування, проблеми ієрархії та потенційно небезпечні конструкції. Відсутність такого аналізу знижує ефективність раннього виявлення дефектів [3, 9].

Для систем, важливих для безпеки, особливо небезпечними є помилки, пов’язані з reset-логікою, тактовими сигналами, асинхронними переходами, неповною ініціалізацією сигналів та некоректною реалізацією цифрових автоматів. Такі дефекти можуть призвести до непередбачуваної поведінки пристрою після запуску, перезапуску або переходу між режимами роботи. У системах безпеки АЕС це є критичним фактором, оскільки цифрова логіка повинна забезпечувати передбачуваний перехід у визначений безпечний стан [5, 8].

Проблемою також є залежність якості перевірки від досвіду конкретного розробника або рецензента. Якщо процедура code review не має формалізованого чек-листа, різні фахівці можуть звертати увагу на різні аспекти коду. Один рецензент може перевіряти стиль іменування, інший – reset-логіку, третій – повноту case-операторів. У результаті перевірка стає нерівномірною, а частина дефектів може залишитися поза увагою. Формалізовані правила дозволяють зменшити цей вплив людського фактора [3].

Узагальнено чинники ризику традиційної процедури розробки та перевірки VHDL-проєктів можна подати у вигляді таблиці 1.1.

Таблиця 1.1 – Основні чинники ризику традиційної процедури розробки VHDL-проєктів

Чинник ризику	Прояв у VHDL-проєкті	Можливі наслідки
Складність структури	Велика кількість модулів, сигналів, процесів і залежностей	Ускладнення аналізу та супроводу
Обмеженість симуляції	Перевіряються лише задані тестові сценарії	Невиявлені стани, переходи або гілки логіки
Пізнє виявлення помилок	Дефекти знаходяться після симуляції або синтезу	Збільшення часу виправлення
Неоднорідний стиль коду	Відсутність єдиних правил іменування та структури	Ускладнення code review
Помилки FSM	Недосяжні стани, відсутній reset, неповний case	Некоректна поведінка цифрового автомата
Недостатній статичний аналіз	Перевірка обмежується компіляцією та симуляцією	Низька ефективність раннього пошуку дефектів
Людський фактор	Відсутність формалізованого чек-листа review	Нерівномірна якість перевірки

Таким чином, традиційна процедура розробки VHDL-проєктів має низку обмежень, які особливо проявляються при створенні цифрових автоматів і FPGA-систем для критичних застосувань. Основними недоліками є складність аналізу структури, залежність від симуляційних сценаріїв, пізнє виявлення помилок, неоднорідність стилю кодування та недостатня формалізація code review. Для усунення цих проблем доцільно використовувати формалізовану процедуру, яка поєднує правила кодування, статичний аналіз, перевірку цифрових автоматів і контроль якості VHDL-коду [3, 5, 9].

1.4 Огляд сучасних інструментів розробки та аналізу HDL-проєктів

Для підвищення якості VHDL-проєктів недостатньо використовувати лише текстовий редактор, компілятор і симулятор. Сучасні FPGA-системи мають складну ієрархічну структуру, значну кількість сигналів, цифрових автоматів, інтерфейсів і залежностей між модулями. Тому ефективна процедура розробки повинна включати інструменти, які підтримують не тільки написання коду, а й структурне проєктування, аналіз залежностей, перевірку правил кодування, оцінювання складності та підготовку коду до review [3, 13].

Одним із таких інструментів є HDL Designer, який використовується для створення, аналізу, візуалізації та супроводження HDL-проєктів. Його доцільно розглядати як інструмент структурного проєктування, оскільки він дозволяє працювати з RTL-дизайном не лише на рівні текстового коду, а й через графічне представлення структури системи. В офіційних матеріалах Siemens зазначено, що HDL Designer допомагає інженерам аналізувати, оцінювати та візуалізувати складні RTL-дизайни, виконувати перевірку цілісності коду, повноти з’єднань і якості HDL-коду [10].

Використання HDL Designer є корисним на ранніх етапах розробки VHDL-проєкту. За допомогою графічного представлення архітектури розробник може побудувати ієрархію модулів, визначити взаємозв’язки між компонентами,

сформувати інтерфейси та виявити потенційні структурні помилки ще до детальної реалізації логіки. Це особливо важливо для цифрових автоматів, оскільки помилки у структурі переходів, взаємодії сигналів або reset-логіці можуть бути складними для виявлення лише на основі текстового коду.

HDL Designer також може застосовуватися для покращення документування проєкту. Графічні схеми, структурні діаграми, представлення блоків і зв'язків між ними полегшують розуміння логіки цифрової системи під час code review. Для великих FPGA-проєктів це має практичне значення, оскільки рецензенту необхідно оцінювати не лише окремі оператори VHDL-коду, а й загальну структуру системи, правильність декомпозиції та логіку взаємодії компонентів [10].

Разом із тим HDL Designer не повинен розглядатися як єдиний інструмент перевірки. Його основна роль полягає у підтримці структурного проєктування, візуалізації та аналізу RTL-дизайну. Для глибшої перевірки якості коду, метрик, складності, залежностей і потенційних проблем доцільно використовувати інструменти статичного аналізу. У межах цієї роботи таким інструментом розглядається Scientific Toolworks Understand.

Understand є інструментом статичного аналізу коду, який дозволяє аналізувати програмні та HDL-проєкти без їх виконання. В офіційному описі SciTools зазначено, що Understand підтримує статичний аналіз, перевірку стандартів кодування, перегляд графів залежностей, метрики на рівні файлів та інших сутностей, а також візуалізацію структури коду через control flow, call trees, dependency graphs та інші подання [11].

У контексті VHDL-проєктів Understand може бути використаний для оцінювання складності файлів, пошуку залежностей між модулями, аналізу структури сигналів, перегляду ієрархії коду та підготовки матеріалів для code review. Це важливо для реалізації правил, пов'язаних зі складністю, читабельністю та супроводжуваністю VHDL-коду. Наприклад, якщо окремий .vhd файл має надмірну кількість умовних переходів, вкладених операторів або складну структуру процесів, такий модуль стає важчим для перевірки.

Особливе значення Understand має при аналізі великих бібліотек VHDL-коду. У таких проєктах ручний перегляд усіх файлів є малоефективним, оскільки рецензенту складно швидко визначити найбільш проблемні ділянки. Статичний аналіз дозволяє попередньо виділити модулі з підвищеною складністю, великою кількістю залежностей або потенційними порушеннями правил кодування. Це робить code review більш цілеспрямованим і зменшує залежність результату перевірки від суб'єктивного досвіду окремого рецензента [11, 13].

Окрему роль у процесі розробки FPGA-проєктів відіграють засоби синтезу та реалізації, зокрема Intel Quartus Prime. У документації Intel зазначено, що стиль написання HDL-коду суттєво впливає на якість результатів для програмованої логіки, оскільки засоби синтезу оптимізують код за ресурсами та продуктивністю, але не можуть самостійно інтерпретувати намір розробника [12]. Саме тому дотримання рекомендованих стилів кодування є важливою умовою отримання коректної та ефективної апаратної реалізації.

Quartus Prime може використовуватися для компіляції, синтезу, перевірки часових характеристик і виявлення частини помилок, пов'язаних із реалізацією цифрової логіки у FPGA. Крім того, Design Assistant у Quartus Prime дозволяє налаштовувати перевірку правил проєкту, змінювати пороги повідомлень про порушення та адаптувати перевірку під особливості конкретного дизайну [12]. Це робить його корисним інструментом на етапі технічної перевірки після попереднього структурного та статичного аналізу.

Водночас засоби синтезу не замінюють процедуру статичного аналізу та code review. Синтезатор може повідомити про синтаксичні помилки, несумісність конструкцій або проблеми реалізації, але він не завжди виявляє недоліки архітектури, погану читабельність, невдале іменування, завищену складність або неповну відповідність внутрішнім правилам кодування. Тому перевірка VHDL-проєкту має бути багаторівневою: спочатку структурне проєктування й аналіз, потім статична перевірка правил, далі симуляція, синтез і підсумковий code review [3, 13]. Порівняння ролі основних інструментів у процедурі розробки VHDL-проєктів наведено в таблиці 1.2.

Таблиця 1.2 – Інструменти у процесі розробки та аналізу VHDL-проєктів

Інструмент	Основне призначення	Роль у процедурі перевірки
HDL Designer	Структурне проєктування, аналіз і візуалізація RTL-дизайну	Побудова архітектури, аналіз зв'язків, контроль структури проєкту
Scientific Toolworks Understand	Статичний аналіз коду, метрики, графи залежностей	Аналіз складності, залежностей, структури файлів і code review
Intel Quartus Prime	Синтез, компіляція, реалізація FPGA-проєкту	Перевірка синтезованості, часових характеристик і частини design rules
Симулятор HDL	Моделювання поведінки системи	Перевірка функційної коректності за тестовими сценаріями
Ручний code review	Експертна перевірка коду	Оцінювання відповідності правилам, зрозумілості та безпечності реалізації

У межах цієї роботи основна увага приділяється поєднанню HDL Designer та Scientific Toolworks Understand. Перший інструмент доцільно використовувати для структурного представлення та аналізу архітектури VHDL-проєкту, другий – для статичного аналізу, оцінювання складності та підтримки code review. Така комбінація дозволяє перевіряти VHDL-код не лише на рівні функційної поведінки, а й на рівні структури, якості, супроводжуваності та відповідності формалізованим правилам [10, 11].

РОЗДІЛ 2
ОПИС УДОСКОНАЛЕНОЇ ПРОЦЕДУРИ СТАТИЧНОГО АНАЛІЗУ ТА ОГЛЯДУ VHDL-ПРОЄКТІВ

2.1 Вихідні положення процедури огляду та перевірки VHDL-коду

Процедура кодування та перевірки VHDL-коду є важливою складовою життєвого циклу FPGA-проєкту, оскільки саме на етапі опису апаратної логіки формуються структура цифрового пристрою, його функційна поведінка, інтерфейси, цифрові автомати та взаємозв’язки між компонентами. На відміну від програмного коду, який виконується процесором, VHDL-код після синтезу перетворюється на апаратну реалізацію у FPGA. Тому помилки у VHDL-описі можуть впливати не лише на результат моделювання, а й на фактичну поведінку цифрової схеми після реалізації [3].

Для FPGA-проєктів, особливо тих, що можуть застосовуватися у критичних системах, процедура перевірки не повинна обмежуватися лише компіляцією або функційною симуляцією. Компіляція дозволяє виявити синтаксичні помилки, а симуляція – перевірити поведінку системи у заданих тестових сценаріях. Однак ці методи не завжди дозволяють своєчасно виявити структурні недоліки коду, порушення стилю, надмірну складність, неповні оператори вибору, недосяжні стани цифрових автоматів або некоректну reset-логіку. Саме тому процедура перевірки має включати статичний аналіз, code review та застосування формалізованих правил кодування [3, 14].

У методичних матеріалах із кодування VHDL для FPGA Electronic Designs зазначено, що перевірка правил програмування має кілька цілей: виключення потенційних проблем, які можуть впливати на коректне виконання функцій FPGA-проєкту; підтримку механізмів виявлення та виправлення помилок; забезпечення стилю кодування, що підвищує читабельність і придатність коду до review [3]. Це означає, що правила кодування виконують не лише формальну роль, а є інструментом підвищення якості та надійності цифрового проєкту.

У межах цієї роботи вихідною основою для побудови вдосконаленої процедури є набір правил, поділений на чотири основні групи: Naming Convention, Coding Style, Design Restrictions та Simulation. До першої групи належать правила іменування файлів, сигналів, портів, констант, типів і цифрових автоматів. Друга група містить правила структурування та оформлення коду. Третя група охоплює проєктні обмеження, пов’язані з безпечним і передбачуваним використанням конструкцій VHDL. Четверта група стосується оптимізації та коректності симуляційних моделей [3].

Важливою особливістю такого набору правил є використання рівнів жорсткості: MAX, NRM та MIN. Рівень MAX відповідає обов’язковим правилам, порушення яких має бути усунене. Рівень NRM означає правила нормальної жорсткості: їх виконання є бажаним, а відхилення повинні бути проаналізовані та обґрунтовані. Рівень MIN має рекомендаційний характер, однак навіть такі порушення мають оцінюватися з погляду можливого впливу на безпеку або якість проєкту [3].

З погляду організації перевірки VHDL-коду така класифікація є зручною, оскільки дозволяє розділити всі виявлені зауваження за критичністю. Наприклад, помилки, пов’язані з відсутністю початкового стану цифрового автомата або некоректністю case-оператора, мають вищий пріоритет, ніж стилістичні недоліки форматування. Це дає змогу під час code review не лише фіксувати порушення, а й визначати порядок їх усунення.

Загальна процедура перевірки VHDL-коду повинна включати кілька послідовних етапів. На першому етапі виконується аналіз структури проєкту: визначаються основні модулі, їх інтерфейси, зв’язки між компонентами та наявність цифрових автоматів. Для цього може використовуватися HDL Designer, який забезпечує візуалізацію RTL-дизайну, аналіз структури та перевірку цілісності HDL-коду [10].

На другому етапі виконується статичний аналіз VHDL-коду. Його завданням є перевірка структури файлів, залежностей між модулями, складності коду, відповідності правилам кодування та наявності потенційно проблемних ділянок.

Для цього доцільно використовувати Scientific Toolworks Understand, оскільки цей інструмент підтримує аналіз метрик, залежностей, структури коду та може застосовуватися для підготовки матеріалів до code review [11].

На третьому етапі проводиться ручний code review. Його метою є експертна оцінка тих аспектів, які не завжди можуть бути повністю перевірені автоматизованими інструментами. До таких аспектів належать коректність логіки переходів цифрового автомата, зрозумілість назв сигналів, обґрунтованість архітектурних рішень, відповідність реалізації функційним вимогам і доцільність використання конкретних конструкцій VHDL [3, 14].

На четвертому етапі виконується перевірка синтезованості та технічної реалізованості проєкту. Для цього може застосовуватися Intel Quartus Prime або інший інструмент синтезу FPGA-проєктів. На цьому етапі перевіряється, чи може VHDL-код бути коректно перетворений в апаратну структуру, чи не містить він несинтезованих конструкцій, проблем із тактовими сигналами, reset-логікою або невідповідністю розрядностей [12].

Узагальнено вихідну процедуру перевірки VHDL-коду можна подати у вигляді таблиці 2.1.

Таблиця 2.1 – Основні етапи процедури кодування та перевірки VHDL-коду

Етап	Зміст етапу	Інструменти або методи	Очікуваний результат
Аналіз структури проєкту	Визначення модулів, інтерфейсів, зв’язків і цифрових автоматів	HDL Designer, перегляд архітектури	Формалізована структура проєкту
Статичний аналіз	Перевірка залежностей, складності, правил кодування	Understand, чек-лист правил	Виявлення потенційних проблем без запуску моделі
Code review	Експертна перевірка логіки, стилю та відповідності правилам	Ручний аналіз, чек-лист	Підтвердження якості та зрозумілості коду
Симуляція	Перевірка поведінки за тестовими сценаріями	HDL-симулятор	Перевірка функційної коректності
Синтез	Перетворення VHDL-коду в апаратну реалізацію	Quartus Prime або аналогічний засіб	Перевірка синтезованості та технічної реалізації

У межах цієї роботи процедура вдосконалюється не шляхом повної заміни наявного процесу, а шляхом доповнення його правилами, які мають особливе значення для цифрових автоматів. Такий підхід є доцільним, оскільки цифрові автомати є типовими елементами керуючої логіки FPGA-проєктів і водночас можуть бути джерелом складних для виявлення помилок. До таких помилок належать недосяжні стани, відсутність початкового стану, неповний опис переходів, некоректний reset або надмірна складність логіки переходів [3].

2.2 Вибір правил для вдосконалення процедури аналізу VHDL-коду

Для вдосконалення процедури кодування та статичного аналізу VHDL-проєктів доцільно не розширювати перевірку всіма можливими правилами одночасно, а виділити обмежений набір контрольних вимог, які мають найбільше значення для цифрових автоматів. Такий підхід дозволяє зробити процедуру більш цілеспрямованою, оскільки основна увага переноситься на ті елементи VHDL-коду, які безпосередньо впливають на передбачуваність поведінки FPGA-пристрою, коректність переходів між станами, синтезованість логіки та зручність code review [3].

У межах цієї роботи пропонується включити до вдосконаленої процедури чотири основні правила: DR-32-MAX, DR-33-MAX, DR-15-MAX та CS-35-NRM. Їх вибір обумовлений тим, що вони охоплюють ключові ризики, характерні для VHDL-проєктів цифрових автоматів: недосяжні стани, невизначений початковий стан, некоректні оператори case та надмірну складність коду. Саме ці дефекти можуть бути складними для виявлення лише засобами симуляції, але водночас можуть суттєво вплинути на поведінку пристрою після синтезу [3, 14].

Першим правилом, яке пропонується включити до процедури, є DR-32-MAX. Його зміст полягає в тому, що цифрові автомати не повинні містити недосяжних станів. Недосяжним вважається стан, у який автомат не може перейти з жодного іншого стану за будь-якої допустимої комбінації вхідних сигналів. Наявність

такого стану може свідчити про логічну надлишковість, помилку у функції переходів або неповну узгодженість між проєктною документацією та реалізацією VHDL-коду [3].

Для FPGA-проєктів це правило має особливе значення, оскільки цифрові автомати часто реалізують керуючу логіку пристрою. Якщо в автоматі присутні недосяжні стани, це ускладнює його аналіз, підвищує складність code review та може приховувати помилки у структурі переходів. У критичних системах така ситуація є небажаною, оскільки кожен стан автомата має бути обґрунтованим, трасованим і перевіреним. Тому під час аналізу VHDL-коду необхідно будувати таблицю переходів або граф станів і перевіряти, чи всі стани можуть бути досягнуті з початкового стану.

Другим правилом є DR-33-MAX, згідно з яким для цифрового автомата має бути визначено початковий стан, а також reset-сигнал, що переводить автомат у цей стан. Це правило є критичним для синхронних цифрових систем, оскільки після запуску, перезапуску або аварійного скидання автомат повинен переходити у передбачуваний стан. Якщо початковий стан не визначено явно, поведінка пристрою після старту може залежати від особливостей синтезу, початкових значень тригерів або конкретної FPGA-платформи [3].

У контексті систем безпеки це правило має особливе значення. Наприклад, цифровий автомат, який відповідає за блокування або перехід обладнання у безпечний режим, не може мати невизначений стартовий стан. Після reset він повинен переходити у стан, який є безпечним з погляду функційної логіки системи. Тому у вдосконаленій процедурі необхідно перевіряти, чи має кожний FSM явно описану reset-гілку, чи встановлюється в ній визначений початковий стан, а також чи відповідає цей стан проєктній логіці пристрою [3, 5].

Третім правилом є DR-15-MAX, яке стосується коректності використання оператора case. У VHDL-проєктах цифрові автомати часто реалізуються саме через case, де кожна гілка відповідає окремому стану або набору умов. Відповідно до цього правила, оператор case має бути повним, не містити дубльованих або

конфліктних умов, не мати недосяжних гілок і повинен містити гілку when others [3].

Порушення цього правила може призвести до помилок під час синтезу FPGA-проєкту або до некоректної поведінки цифрового автомата. Наприклад, якщо для певного стану не описано дію, синтезатор може сформувати небажану логіку збереження попереднього значення. Якщо відсутня гілка when others, автомат може не мати передбачуваної поведінки для нештатних або помилкових значень стану. Тому під час code review необхідно окремо перевіряти всі оператори case, що використовуються у FSM, декодерах станів, блоках керування та комбінаційній логіці переходів [3, 12].

Четвертим правилом є CS-35-NRM, згідно з яким цикломатична складність VHDL-файлу, визначена за модифікованим методом McCabe, не повинна перевищувати 15. Це правило має рівень жорсткості NRM, тобто його порушення не завжди означає безумовну помилку, однак потребує аналізу та обґрунтування. Надмірна складність VHDL-коду ускладнює його розуміння, тестування, супровід і перевірку під час code review [3].

Для цифрових автоматів показник складності є особливо важливим, оскільки велика кількість станів, вкладених умов, переходів і комбінацій вхідних сигналів збільшує ризик логічних помилок. Якщо один .vhd файл містить надто складний автомат або кілька незалежних автоматів, доцільно розглянути можливість декомпозиції: винесення окремих функцій у підмодулі, спрощення логіки переходів, розділення керуючої та виконавчої частини або уточнення структури процесів. Для контролю цього правила можуть використовуватися інструменти статичного аналізу, зокрема Scientific Toolworks Understand, який підтримує аналіз метрик, залежностей і структури коду [11].

Вибрані правила мають різну спрямованість, але разом формують цілісний набір для перевірки цифрових автоматів. DR-32-MAX контролює повноту та досяжність станів, DR-33-MAX забезпечує визначеність початкової поведінки, DR-15-MAX перевіряє коректність операторів вибору, а CS-35-NRM дозволяє оцінити складність реалізації. У сукупності ці правила підвищують якість аналізу FSM і

роблять процедуру code review більш формалізованою. Узагальнення запропонованих правил наведено в таблиці 2.2.

Таблиця 2.2 – Правила, запропоновані для включення до вдосконаленої процедури

Код правила	Зміст правила	Рівень жорсткості	Причина включення до процедури	Засіб перевірки
DR-32-MAX	Цифрові автомати не повинні містити недосяжних станів	MAX	Дозволяє виявити помилки у функції переходів FSM	HDL Designer, ручний аналіз графа станів, code review
DR-33-MAX	Для FSM має бути визначено початковий стан і reset-сигнал	MAX	Забезпечує передбачувану поведінку після запуску або скидання	Code review, аналіз reset-гілки, симуляція
DR-15-MAX	Оператор case має бути повним, без дублювань, недосяжних гілок і з when others	MAX	Зменшує ризик некоректного синтезу та неповної логіки переходів	HDL Designer, Quartus Prime, code review
CS-35-NRM	Цикломатична складність .vhd файлу не повинна перевищувати 15	NRM	Дозволяє виявляти надмірно складні модулі, які потребують декомпозиції	Understand, аналіз метрик

Практична цінність запропонованого набору правил полягає в тому, що він не дублює повністю всі наявні вимоги до VHDL-коду, а виділяє найбільш важливі контрольні точки для аналізу цифрових автоматів. Це дає змогу побудувати стислий, але ефективний чек-лист перевірки, який може застосовуватися як під час ручного code review, так і під час статичного аналізу з використанням інструментальних засобів [10, 11, 12].

Запропонований підхід також дозволяє розділити відповідальність між інструментами. HDL Designer доцільно використовувати для візуалізації структури автомата, аналізу ієрархії та зв’язків між модулями. Understand варто застосовувати для оцінки складності, залежностей і загальної структури VHDL-коду. Quartus Prime може бути використаний для перевірки синтезованості та виявлення частини помилок, пов’язаних із реалізацією логіки у FPGA.

2.3 Інтеграція розробленого правил у процедуру static code analize та code review VHDL-коду

Після вибору правил DR-32-MAX, DR-33-MAX, DR-15-MAX та CS-35-NRM необхідно визначити порядок їх практичного включення до процедури перевірки VHDL-коду. Сам факт наявності правил не забезпечує підвищення якості проєкту, якщо вони не прив’язані до конкретних етапів аналізу, відповідальних дій, інструментів перевірки та форми документування результатів. Тому вдосконалення процедури полягає не лише у додаванні окремих вимог, а у формуванні послідовного механізму їх застосування під час статичного аналізу та code review [3, 14].

Запропонована процедура має бути орієнтована насамперед на аналіз цифрових автоматів, оскільки саме вони часто реалізують керуючу логіку FPGA-проєкту. У таких автоматах критичними є коректність початкового стану, повнота переходів, відсутність недосяжних станів, наявність reset-логіки та контроль складності. Якщо ці аспекти не перевіряються окремо, частина дефектів може залишитися невиявленою до етапу симуляції або синтезу, що збільшує трудовитрати на виправлення помилок.

Інтеграцію правил доцільно виконувати у вигляді окремого чек-листа для перевірки цифрових автоматів. Такий чек-лист повинен застосовуватися після первинного аналізу структури проєкту, але до фінальної симуляції та синтезу. Це дозволяє виявляти частину помилок ще на етапі статичного аналізу, коли виправлення не потребує повного повторення всього циклу тестування.

Першим етапом удосконаленої процедури є ідентифікація VHDL-модулів, які містять цифрові автомати. Для цього перевіряються файли, у яких наявні перелічувані типи станів, сигнали на зразок state, next_state, current_state, оператори case, reset-гілки та синхронні процеси зміни стану. На цьому етапі доцільно використовувати HDL Designer для перегляду структури RTL-дизайну, а також ручний аналіз коду для уточнення ролі кожного автомата [10].

Другим етапом є побудова логіки переходів цифрового автомата. Для кожного знайденого FSM потрібно визначити перелік станів, початковий стан,

умови переходів, вихідні сигнали та гілку обробки нештатних або невизначених значень. Результатом цього етапу може бути таблиця переходів або граф станів. Такий опис полегшує перевірку правил DR-32-MAX і DR-33-MAX, оскільки дозволяє явно побачити, чи всі стани досяжні та чи існує визначений перехід у початковий стан після reset [3].

Третім етапом є перевірка правила DR-32-MAX, яке вимагає відсутності недосяжних станів у цифровому автоматі. Для цього всі стани, оголошені у VHDL-коді, порівнюються зі станами, які реально можуть бути досягнуті з початкового стану через описані переходи. Якщо певний стан оголошений, але жодна гілка переходу не приводить до нього, такий стан вважається потенційним порушенням. У цьому випадку необхідно або виправити логіку переходів, або вилучити зайвий стан, або документально обґрунтувати його наявність, якщо він передбачений для майбутнього розширення функціональності. У правилах VHDL-кодування недосяжний стан прямо розглядається як ознака логічної надлишковості або некоректно заданої функції переходів FPGA electronic design [3].

Четвертим етапом є перевірка правила DR-33-MAX. Для кожного FSM необхідно встановити, чи має він явно визначений початковий стан і reset-сигнал, який переводить автомат у цей стан. Особлива увага приділяється тому, чи описано reset-гілку до переходу за тактовим фронтом, чи не залежить початковий стан від випадкових або неініціалізованих сигналів, а також чи відповідає початковий стан логіці безпечного запуску пристрою. Для цифрових систем, що можуть застосовуватися у критичних умовах, це правило має принципове значення, оскільки після запуску або скидання пристрій повинен переходити у передбачуваний стан.

П’ятим етапом є перевірка правила DR-15-MAX, яке стосується коректності використання оператора case. Під час аналізу потрібно перевірити, чи всі можливі значення станів або керуючих сигналів оброблені, чи немає дублювання умов, чи не містить case недосяжних гілок, а також чи передбачена гілка when others. Наявність when others є важливою для забезпечення передбачуваної поведінки автомата у випадку нештатного значення стану або помилки в логіці переходів [3].

Шостим етапом є перевірка правила CS-35-NRM, яке обмежує цикломатичну складність VHDL-файлу. На цьому етапі доцільно використовувати Scientific Toolworks Understand, оскільки цей інструмент дозволяє аналізувати метрики коду, залежності між елементами проєкту та загальну структуру файлів [11]. Якщо складність файлу перевищує встановлений поріг, це не завжди означає безумовну помилку, але потребує аналізу. У такому випадку необхідно оцінити, чи можна спростити логіку, розділити автомат на кілька підмодулів, винести повторювані фрагменти у допоміжні блоки або покращити структуру процесів.

Сьомим етапом є документування результатів перевірки. Для кожного правила необхідно зафіксувати статус: “виконано”, “порушено”, “потребує обґрунтування” або “не застосовується”. Якщо виявлено порушення правила з рівнем MAX, воно має бути усунене до переходу на наступний етап. Якщо порушення стосується правила рівня NRM, воно може бути або виправлене, або залишене за наявності аргументованого пояснення. Такий підхід відповідає класифікації жорсткості правил, де MAX означає обов’язкове усунення невідповідності, а NRM – необхідність аналізу й обґрунтування [3].

Узагальнений порядок застосування правил у межах удосконаленої процедури наведено в таблиці 2.3.

Таблиця 2.3 – Порядок інтеграції правил у процедуру аналізу VHDL-коду

Етап	Дія	Правило	Інструмент або метод	Результат
1	Виявлення модулів, що містять FSM	DR-32, DR-33, DR-15	HDL Designer, ручний перегляд коду	Перелік цифрових автоматів
2	Побудова таблиці або графа переходів	DR-32, DR-33	Ручний аналіз, HDL Designer	Формалізований опис переходів
3	Перевірка досяжності станів	DR-32-MAX	Аналіз графа станів, code review	Виявлення недосяжних станів
4	Перевірка reset і початкового стану	DR-33-MAX	Аналіз синхронного процесу	Підтвердження визначеної стартової поведінки
5	Перевірка операторів case	DR-15-MAX	Code review, Quartus Prime	Виявлення неповних або конфліктних гілок
6	Аналіз складності VHDL-файлу	CS-35-NRM	Understand	Виявлення модулів, що потребують спрощення
7	Фіксація результатів перевірки	Усі вибрані правила	Чек-лист SCA/CR	Документований результат аналізу

Для практичного застосування процедури доцільно використовувати форму чек-листа. Вона дозволяє стандартизувати перевірку та зменшити залежність результату від досвіду конкретного рецензента.

Особливістю запропонованої інтеграції є розподіл перевірок між різними засобами. HDL Designer використовується переважно для структурного аналізу та візуалізації архітектури, Understand – для статичного аналізу й метрик складності, Quartus Prime – для перевірки синтезованості та виявлення частини технічних проблем, а ручний code review – для оцінки логічної коректності FSM та обґрунтованості проєктних рішень.

2.4 Очікуваний ефект від впровадження удосконаленої процедури

Запропонована процедура кодування та статичного аналізу VHDL-проєктів спрямована на підвищення якості перевірки цифрових автоматів за рахунок формалізації контрольних дій. Її основна відмінність від традиційного підходу полягає в тому, що перевірка виконується не лише після симуляції або синтезу, а ще на етапі аналізу структури коду, опису станів, reset-логіки, операторів case та складності реалізації. Такий підхід дозволяє виявляти частину дефектів раніше, коли їх виправлення потребує менших трудовитрат [3, 14].

У вихідному наборі правил VHDL-кодування правила поділяються на групи NC, CS, DR і SIM, а також мають рівні жорсткості MAX, NRM і MIN. Для правил рівня MAX невідповідність має бути усунена обов’язково, для NRM – проаналізована та або усунена, або обґрунтована, а для MIN – оцінена з погляду впливу на безпеку та якість проєкту. Така класифікація дозволяє не тільки фіксувати порушення, а й визначати їх пріоритет під час code review [3].

Основний очікуваний ефект від впровадження правил DR-32-MAX, DR-33-MAX, DR-15-MAX та CS-35-NRM полягає у зменшенні кількості логічних і структурних помилок у VHDL-коді цифрових автоматів. Ці правила охоплюють найбільш важливі аспекти FSM: досяжність станів, наявність початкового стану,

коректність reset-логіки, повноту операторів case і контроль складності. У результаті рецензент отримує не загальне завдання “перевірити код”, а конкретний набір критеріїв, за якими оцінюється якість реалізації автомата [3].

Першим результатом є підвищення передбачуваності поведінки цифрового автомата. Завдяки перевірці правила DR-33-MAX кожний FSM повинен мати явно визначений початковий стан і reset-сигнал. Це зменшує ризик неконтрольованої поведінки пристрою після запуску або перезапуску. Для FPGA-проєктів, які можуть застосовуватися у критичних системах, така вимога є особливо важливою, оскільки після скидання цифрова логіка має переходити у визначений і безпечний стан [3].

Другим результатом є зменшення надлишкової або помилкової логіки. Перевірка правила DR-32-MAX дозволяє виявляти недосяжні стани цифрового автомата. Наявність таких станів може свідчити про помилку у функції переходів, залишки попередньої реалізації або невідповідність між проєктною логікою та фактичним VHDL-кодом. Усунення недосяжних станів спрощує автомат, полегшує його тестування та робить структуру переходів більш зрозумілою для рецензента [3, 14].

Третім результатом є підвищення коректності операторів вибору. Правило DR-15-MAX вимагає перевіряти повноту case, відсутність дубльованих умов, недосяжних гілок і наявність when others. Це знижує ризик неповної реалізації логіки переходів або появи небажаної синтезованої поведінки. Для цифрових автоматів це правило має пряме значення, оскільки саме через case часто описується поведінка FSM у кожному стані [3, 12].

Четвертим результатом є покращення супроводжуваності коду. Правило CS-35-NRM дозволяє контролювати цикломатичну складність VHDL-файлів. Якщо складність перевищує встановлений поріг, це є підставою для додаткового аналізу або декомпозиції модуля. Надмірно складний файл важче перевіряти, тестувати та змінювати, тому контроль складності сприяє підвищенню якості code review. Для цього доцільно застосовувати Scientific Toolworks Understand, який підтримує аналіз метрик, залежностей і структури коду [11].

Окремим ефектом є покращення документування результатів перевірки. Використання чек-листа дозволяє фіксувати не лише факт проходження review, а й конкретні результати за кожним правилом. Це важливо для повторної перевірки, аналізу змін і пояснення прийнятих проєктних рішень. Якщо певне правило рівня NRM не виконується, але відхилення є обґрунтованим, таке пояснення має бути внесене до результатів аналізу. Це підвищує прозорість процедури та зменшує залежність від суб'єктивної оцінки окремого розробника [3, 14].

Запропонована процедура також забезпечує кращий розподіл функцій між інструментами. HDL Designer доцільно використовувати для структурного аналізу, візуалізації архітектури та пошуку взаємозв'язків між модулями. Understand – для аналізу складності, залежностей і метрик. Quartus Prime – для перевірки синтезованості, відповідності HDL-коду вимогам FPGA-реалізації та виявлення частини технічних проблем, пов'язаних зі структурою цифрової логіки [10, 11, 12].

Для оцінювання ефективності процедури можна використовувати кілька практичних критеріїв: кількість виявлених порушень за кожним правилом, кількість усунених дефектів до етапу симуляції, кількість модулів із перевищеною складністю, час, витрачений на code review, та кількість повторних ітерацій перевірки після внесення змін. Такі критерії дозволяють оцінити процедуру не лише якісно, а й кількісно.

У межах практичного застосування до VHDL-бібліотеки доцільно сформувати таблицю результатів аналізу для кожного обраного файлу. У ній слід зазначити, які правила були застосовані, чи виявлено порушення, які зміни запропоновано та який очікуваний ефект має таке вдосконалення. Це дозволить у третьому розділі перейти від теоретичного опису процедури до її практичного застосування на реальному наборі VHDL-файлів.

РОЗДІЛ 3
ЗАСТОСУВАННЯ УДОСКОНАЛЕНОЇ ПРОЦЕДУРИ ДО ВІДКРИТОЇ VHDL-БІБЛІОТЕКИ

3.1 Підготовка та проведення практичного застосування удосконаленої процедури SCA/CR

Практичне застосування удосконаленої процедури SCA/CR здійснюється з метою перевірки можливості використання сформованого у другому розділі контрольного блоку для аналізу реального VHDL-коду. Основна увага у цьому розділі приділяється не теоретичному опису правил, а їх практичному застосуванню до фрагментів, у яких реалізовано цифрові автомати, оператори вибору case, reset-логіку та складні послідовні переходи між станами.

Відповідно до процедури VHDL SCA/CR, статичний аналіз і code review використовуються для перевірки відсутності дефектів кодування, встановлення відповідності VHDL-коду правилам кодування, а також для виявлення невідповідностей між реалізацією та технічною проєктною документацією [14]. Об’єктом такої перевірки може бути як повний електронний проєкт, так і його функціонально завершені складові, функціональні бібліотеки VHDL-коду [14].

Практичне застосування процедури виконується у кілька послідовних етапів. Спочатку обираються VHDL-файли, які містять ознаки цифрових автоматів або складної керуючої логіки. Далі ці файли завантажуються до інструментального середовища для попереднього аналізу структури. Після цього виконується пошук фрагментів, до яких можуть бути застосовані контрольні пункти удосконаленої процедури: перевірка досяжності станів, наявності початкового стану та reset-сигналу, повноти операторів case, а також складності VHDL-файлу.

На першому етапі VHDL-файли завантажуються до середовища HDL Designer. Використання цього інструменту дає змогу створити проєкт для аналізу, підключити необхідні .vhd файли, переглянути структуру модулів і виконати первинну оцінку організації VHDL-коду. У процедурі SCA/CR зазначено, що HDL

Designer може застосовуватися для виявлення невідповідностей стандартизованим правилам, автоматичної перевірки стилю кодування, а також для виявлення функціональних і структурних проблем у VHDL-коді [14].

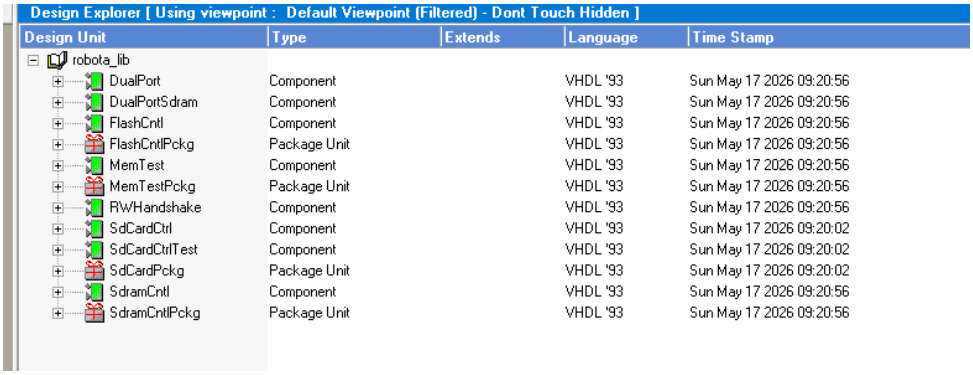

Рисунок 3.1 – Створення проєкту VHDL_Lib у середовищі HDL Designer

Після створення проєкту виконується первинний перегляд структури бібліотеки. На цьому етапі визначаються файли, у яких наявні ознаки цифрових автоматів: перелічувані типи станів, сигнали поточного або наступного стану, оператори case, синхронні процеси та reset-гілки.

На другому етапі виконується аналіз складності VHDL-коду з використанням Scientific Toolworks Understand. Цей етап є важливим для великих файлів, оскільки ручний аналіз значної кількості умовних переходів, операторів case та взаємопов’язаних процесів може бути недостатньо ефективним. У процедурі SCA/CR зазначено, що Scientific Toolworks Understand використовується для перевірки цикломатичної складності VHDL-коду за правилом CS-35-NRM [14].

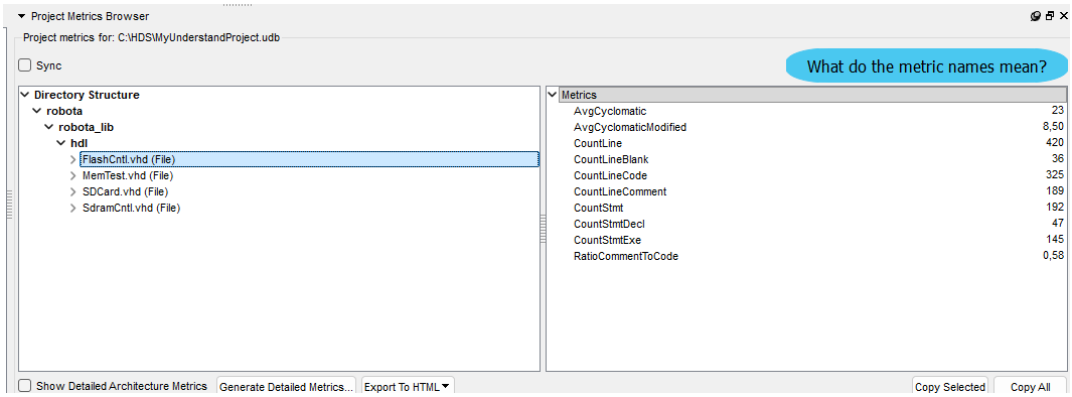

Рисунок 3.2 – Аналіз складності Scientific

Після виконання інструментального аналізу проводиться ручний code review вибраних фрагментів VHDL-коду. На цьому етапі застосовуються контрольні пункти, сформовані у другому розділі: перевірка досяжності станів цифрового автомата, перевірка наявності початкового стану та reset-сигналу, перевірка повноти операторів case, а також оцінювання складності VHDL-файлу. Ручний огляд є необхідним, оскільки не всі аспекти логіки цифрового автомата можуть бути повністю оцінені автоматизованими засобами. Зокрема, експертної перевірки потребують правильність переходів між станами, обґрунтованість початкового стану, призначення гілки when others та доцільність запропонованих змін.

Практичне застосування контрольного блоку виконується за схемою “до вдосконалення – після вдосконалення”. Спочатку фіксується вихідний фрагмент VHDL-коду, у якому виявлено ризик або недолік. Далі визначається контрольний пункт, за яким встановлено проблему. Після цього формується запропонований варіант удосконалення: явне виділення штатного стану, уточнення гілки when others, переведення автомата у визначений стан або рекомендація щодо декомпозиції складного модуля. Послідовність практичного застосування удосконаленої процедури наведено в таблиці 3.1.

Таблиця 3.1 – Послідовність практичного застосування удосконаленої процедури SCA/CR

Етап	Зміст дії	Інструмент або метод	Очікуваний результат
1	Створення проєкту та підключення VHDL-файлів	HDL Designer	Формування об’єкта аналізу
2	Перегляд структури модулів і пошук FSM-фрагментів	HDL Designer, ручний перегляд коду	Виявлення цифрових автоматів
3	Аналіз складності великих VHDL-файлів	Scientific Toolworks Understand	Визначення файлів підвищеної складності
4	Застосування контрольних пунктів до FSM-коду	Code review, чек-лист	Виявлення ризиків у логіці станів, reset і case
5	Формування пропозицій щодо вдосконалення	Порівняння “до / після”	Обґрунтовані зміни або рекомендації
6	Документування результатів перевірки	Таблиця результатів SCA/CR	Фіксація виявлених проблем і запропонованих рішень

Практичне застосування удосконаленої процедури SCA/CR передбачає поєднання інструментального аналізу та ручного code review. HDL Designer використовується для створення проєкту й первинного перегляду структури VHDL-файлів, Scientific Toolworks Understand – для аналізу складності, а ручний огляд – для перевірки логіки цифрових автоматів за контрольними пунктами. Подальший аналіз передбачає вибір конкретних фрагментів бібліотеки, фіксацію виявлених проблем, демонстрацію прикладів “до / після” та формування рекомендацій щодо підвищення визначеності, читабельності й перевірюваності VHDL-коду.

3.2 Вибір VHDL-файлів та визначення напрямів їх удосконалення

Для практичного застосування удосконаленої процедури SCA/CR було обрано окремі VHDL-файли з бібліотеки VHDL_Lib. Вибір файлів здійснювався не довільно, а з урахуванням можливості продемонструвати роботу контрольних пунктів, сформованих у другому розділі роботи. Основним критерієм відбору була наявність у файлах цифрових автоматів, операторів case, reset-логіки, гілок when others та складної послідовної логіки, яка потребує додаткового аналізу.

Процедура VHDL SCA/CR передбачає можливість аналізу як повного електронного проєкту, так і його функціонально завершених складових, зокрема функціональних бібліотек VHDL-коду [14]. Тому використання бібліотеки VHDL_Lib як об’єкта практичного аналізу є доцільним, оскільки вона містить реальні VHDL-модулі, придатні для перевірки за контрольними пунктами удосконаленої процедури.

У межах практичної частини було обрано чотири файли: MemTest.vhd, FlashCntl.vhd, SdramCntl.vhd та SDCard.vhd. Кожен із них використовується для демонстрації окремого напряму вдосконалення. Файл MemTest.vhd застосовується для перевірки повноти опису станів цифрового автомата. Файл FlashCntl.vhd використовується для аналізу поведінки гілки when others. Файл SdramCntl.vhd дає

змогу продемонструвати необхідність побудови таблиці переходів для складного автомата. Файл SDCard.vhd використовується для аналізу великої кількості основних і службових станів FSM.

Вибір цих файлів пов’язаний із тим, що вони дозволяють показати не лише факт виконання перевірки, а й практичний результат удосконалення. У кожному випадку аналіз виконується за схемою: виявлення фрагмента коду, визначення потенційного ризику, застосування контрольного пункту, формування пропозиції щодо покращення та оцінювання очікуваного ефекту.

Файл MemTest.vhd – це цифровий пристрій для автоматичної перевірки оперативної пам'яті за технологією вбудованого самотестування (BIST). Він послідовно записує псевдовипадкові дані у заданий діапазон адрес (режим LOAD), зчитує їх назад (режим COMPARE) і порівнює з оригіналом. Якщо виявляється помилка, пристрій миттєво сигналізує про це через апаратний прапорець err_o.

В основі модуля лежить скінченний автомат (FSM), який керує логікою тестів та генератором випадкових чисел randGen. У бакалаврській роботі цей елемент використовується як об'єкт дослідження якості апаратного коду. Він навмисно містить інженерні помилки в структурі автомата та скидання тригерів, які зафіксовані в коментарях як порушення DR-32-MAX та DR-33-MAX.

Для виявлення та аналізу цих дефектів у дипломному проекті застосовуються спеціальні інструменти. Програмний комплекс Siemens HDL Designer використовується для візуалізації структури автомата та перевірки правил проектування. Аналітична платформа SciTools Understand дозволяє провести статичний аналіз вихідного тексту, оцінити його складність та виявити недосяжні стани автомата (наприклад, ізольований стан EMPTY_PIPE).

Файл FlashCntl.vhd обрано як об'єкт дослідження складного контролера Flash-пам’яті з метою перевірки стійкості його скінченного автомата (FSM). Під час аналізу виявлено, що у критичній гілці when others використовується порожня дія null. З погляду надійності цифрових систем це оцінюється як серйозний ризик непередбачуваної поведінки пристрою у разі виникнення апаратного збою в ПЛІС. Замість модифікації самого коду, у роботі обґрунтовано теоретичну необхідність

перенаправлення цієї гілки у безпечний початковий стан (RAM_BLK_INIT) для усунення потенційного зависання автомата.

Для перевірки структурної складності та архітектурної стійкості цифрового автомата (FSM) у файлі FlashCntl.vhd було додатково залучено інструмент статичного аналізу SciTools Understand. На основі галузевого стандарту метрик Hersteller Initiative Software (HIS) у конфігурації перевірки коду було виставлено гранично допустиме нормативне значення цикломатичної складності (Cyclomatic Complexity), що дорівнює 15. За результатами автоматизованого сканування (правило CS-35-NRM) у процесі обробки логіки автомата у файлі FlashCntl.vhd (рядок 182) зафіксовано критичне порушення: показник складності становить 45, що втричі перевищує встановлений ліміт. Такий високий рівень комплексності підтверджує наявність надмірно розгалуженої логіки у процесі керування, що безпосередньо корелює із виявленим ризиком використання порожньої дії null у захисній гілці when others. Отримані метричні дані з інструменту Understand разом із результатами лінтингу HDL Designer підтверджують критичну необхідність перенаправлення логіки автомата у безпечний початковий стан для зниження структурної складності та забезпечення апаратної надійності пристрою.

Файл SdramCntl.vhd – це апаратний контролер для керування зовнішньою мікросхемою синхронної динамічної пам'яті (SDRAM). Його головна функція полягає в автоматизації складних часових діаграм пам'яті та перетворенні високорівневих запитів на читання й запис у фізичні сигнали керування (строби адрес рядків і колонок, вибір кристалу та дозвіл запису). Модуль має гнучку систему параметрів, що дозволяє адаптувати тактову частоту й розрядність шин під конкретну конфігурацію заліза.

В основі архітектури лежить скінченний автомат (FSM), який повністю координує цикли ініціалізації, активації рядів, регенерації даних та переходу в режим енергозбереження (SELFREFRESH). У бакалаврській роботі цей файл досліджується як приклад відмовостійкого проектування, оскільки його критична гілка when others реалізована інженерно правильно: у разі будь-якого апаратного збою автомат примусово повертається у безпечний стан очікування стабілізації

частоти (INITWAIT). Через велику кількість внутрішніх таймерів та лічильників затримок модуль має високу цикломатичну складність, що робить його ключовим об'єктом аналізу надійності.

Файл SDCard.vhd обрано як приклад контролера зовнішнього пристрою, у якому цифровий автомат може мати значну кількість основних і допоміжних станів. У такому випадку звичайний перегляд оператора case може бути недостатнім, оскільки частина станів відповідає за виконання команд, а інша частина – за передавання бітів, очікування відповіді або формування службових сигналів. Тому напрям удосконалення для цього файлу полягає у групуванні станів FSM за функційним призначенням. Це спрощує code review і дозволяє окремо перевірити основну командну логіку та допоміжні службові переходи. Узагальнення вибраних файлів і напрямів їх удосконалення наведено в таблиці 3.2.

Таблиця 3.2 – Напрями практичного вдосконалення вибраних VHDL-файлів

Файл	Виявлений аспект для аналізу	Контрольний пункт процедури	Напрям удосконалення	Очікуваний результат
MemTest.vhd	Штатний завершальний стан FSM може оброблятися через when others	Перевірка повноти case	Явно виділити завершальний стан в окрему гілку case	Підвищення читабельності та прозорості FSM
FlashCntl.vhd	Гілка when others може не задавати визначеної поведінки	Перевірка коректності case та нештатної поведінки FSM	Додати перехід у визначений початковий або безпечний стан	Підвищення передбачуваності поведінки автомата
SdramCntl.vhd	Складна логіка переходів між станами пам’яті	Перевірка досяжності станів FSM	Побудувати таблицю або граф переходів	Спрощення code review та перевірки досяжності станів
SDCard.vhd	Велика кількість основних і службових станів	Перевірка структури FSM	Згрупувати стани за функційним призначенням	Підвищення структурованості аналізу автомата

Визначено не лише перелік VHDL-файлів для практичного аналізу, а й конкретні напрями їх удосконалення. Обрані файли дозволяють продемонструвати різні форми застосування удосконаленої процедури: локальне уточнення VHDL-коду, покращення поведінки гілки when others, формалізацію аналізу переходів

FSM та структуризацію складного цифрового автомата. Подальший підрозділ присвячено безпосередньому застосуванню цих напрямів до фрагментів коду та порівнянню результатів до і після вдосконалення.

3.3 Демонстрація практичного застосування контрольних пунктів до VHDL-коду

Після вибору VHDL-файлів та визначення напрямів їх удосконалення було виконано практичне застосування контрольних пунктів, сформованих у другому розділі роботи. Основною метою цього етапу є демонстрація того, як удосконалена процедура SCA/CR дозволяє виявляти фрагменти VHDL-коду, що потребують уточнення, підвищення визначеності або додаткового аналізу.

Практичне застосування виконувалося за схемою: фіксація вихідного фрагмента коду, визначення потенційного ризику, застосування відповідного контрольного пункту, формування пропозиції щодо вдосконалення та порівняння результату до і після зміни. Такий підхід дозволяє не лише констатувати наявність певного недоліку, а й показати, яким чином запропонована процедура сприяє підвищенню якості VHDL-коду.

Першим прикладом практичного застосування процедури SCA є комплексний аналіз фрагмента цифрового автомата у файлі MemTest.vhd. Під час автоматизованого сканування проєкту утиліта лінтингу HDL Designer згенерувала попередження щодо структури оператора case, а вбудований аналізатор правил кодування (Code Check) у середовищі SciTools Understand зафіксував цей фрагмент як відхилення від оптимального стандарту проектування FSM. В аналізованій структурі завершальний штатний стан автомата обробляється через захисну гілку when others. Такий підхід формально є синтаксично коректним і не призводить до помилок компіляції, проте інструменти автоматичного аналізу класифікують його як дефект стилю кодування, що знижує прозорість кінцевого автомата (FSM),

оскільки гілка when others має використовуватися виключно для нештатних або непередбачених значень стану.

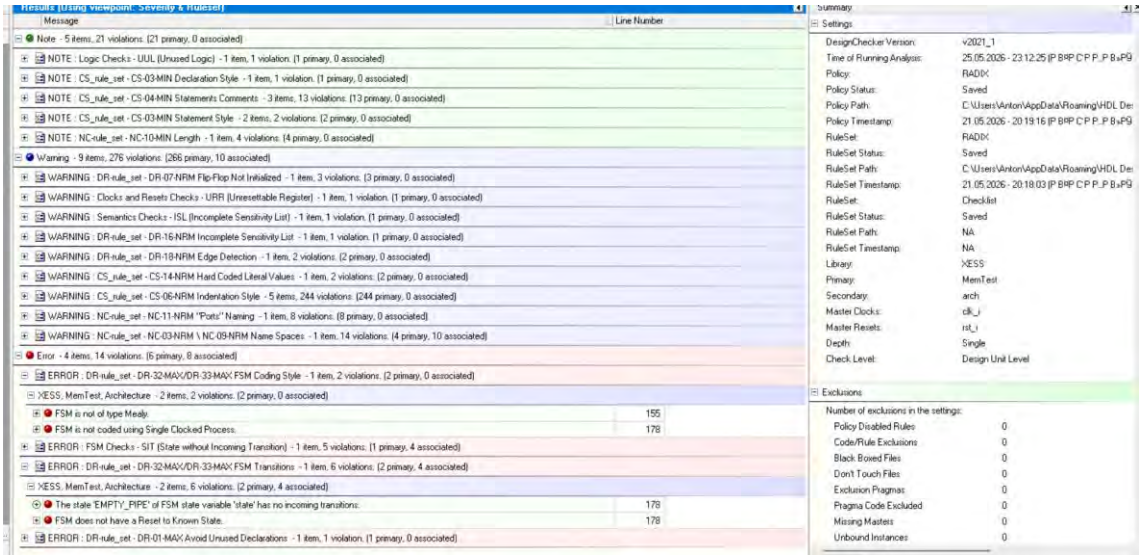

Рисунок 3.3 – Результат аналізу структури FSM та фіксація попередження в середовищі HDL Designer

Для перехресної верифікації та аналізу метрик цей же фрагмент коду було досліджено у середовищі SciTools Understand. За результатами автоматизованого сканування та перевірки правил кодування (Code Check) у програмі Understand для цього блоку не було виявлено жодних помилок чи критичних порушень стандартів (отримано статус Compliant). Це свідчить про те, що архітектурна структура та метрики складності даного компонента повністю вкладаються у встановлені нормативи проєкту.

Другим прикладом практичного застосування є аналіз фрагмента файлу FlashCntl.vhd, який містить логіку складного контролера Flash-пам’яті. Під час проведення автоматизованого статичного аналізу (SCA) особливу увагу було приділено захисній гілці when others кінцевого автомата (FSM). У досліджуваному модулі у разі виникнення нештатного значення стану в цій гілці використовується порожня дія null, тобто не задається жодної активної реакції системи. З погляду вдосконаленої процедури SCA/CR така реалізація класифікується як серйозний ризик непередбачуваної поведінки або зависання пристрою у разі виникнення

апаратного збою в ПЛІС, оскільки автомат не переводиться у визначений безпечний стан.

Для детальної перевірки структурної складності цього процесу було залучено інструмент SciTools Understand. На основі галузевого стандарту метрик Hersteller Initiative Software (HIS) у конфігурації перевірки коду було виставлено гранично допустиме нормативне значення цикломатичної складності (Cyclomatic Complexity), що дорівнює 15. За результатами автоматизованого сканування (правило HIS_04) у рядку 182 для процесу керування автомата зафіксовано критичне порушення: показник складності становить 45, що втричі перевищує встановлений ліміт.

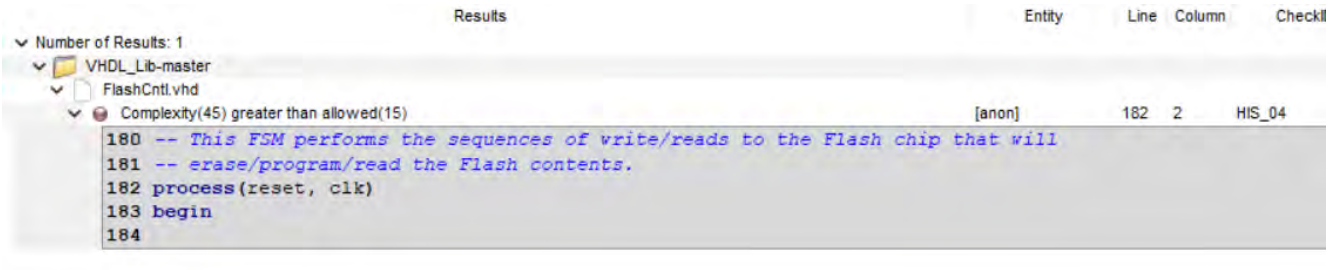

Рисунок 3.4 – Результати фіксації порушення цикломатичної складності та аналізу захисної логіки FSM у SciTools Understand для файлу FlashCntl.vhd

Виявлене відхилення метрик та код процесу проаналізовано за контрольним пунктом перевірки оператора case у нештатних умовах. Надмірна складність логіки у поєднанні з відсутністю реакції в when others ускладнює верифікацію, залишаючи поведінку системи при збої невизначеною.

На основі аналізу (без прямого втручання в код) сформовано проєктну рекомендацію для етапу Code Review: перенаправити гілку when others до безпечного початкового стану (RAM_BLK_INIT) та встановити службові сигнали у передбачувані значення.

Ця рекомендація демонструє головний принцип SCA-перевірки: захисна гілка FSM повинна мати чітко визначену реакцію. Це дозволяє знизити зафіксовану в Understand цикломатичну складність та підвищити апаратну надійність системи.

Третім прикладом є SCA-аналіз складного модуля SdramCntl.vhd. Через значну кількість станів автомата (FSM) ручний перегляд коду тут є неефективним, тому аналіз було формалізовано за допомогою автоматизованих інструментів.

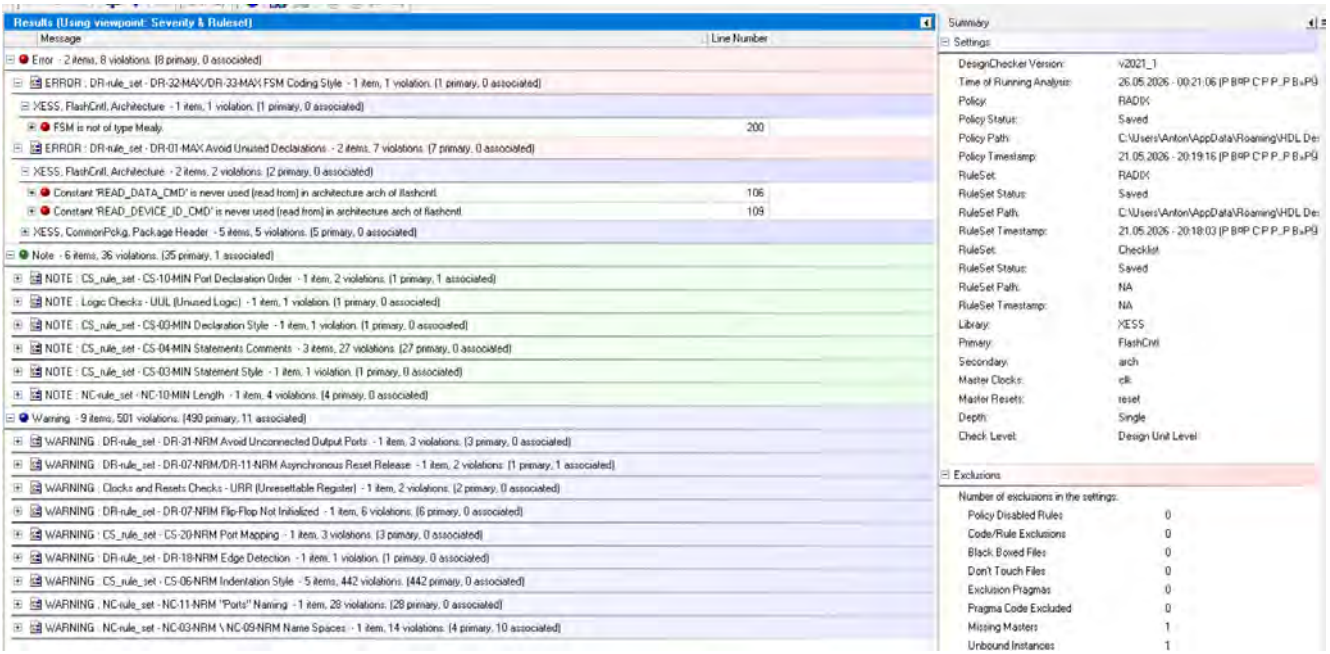

Рисунок 3.5 – Формування рекомендацій щодо обробки нештатного стану контролера у середовищі HDL Designer

Під час сканування проєкту у SciTools Understand (правило HIS_04) для файлу SdramCntl.vhd було зафіксовано критичне перевищення нормативного ліміту цикломатичної складності (15): показник склав 38. Це підтверджує наявність надзвичайно заплутаної логіки переходів, що суттєво підвищує ризик апаратних збоїв.

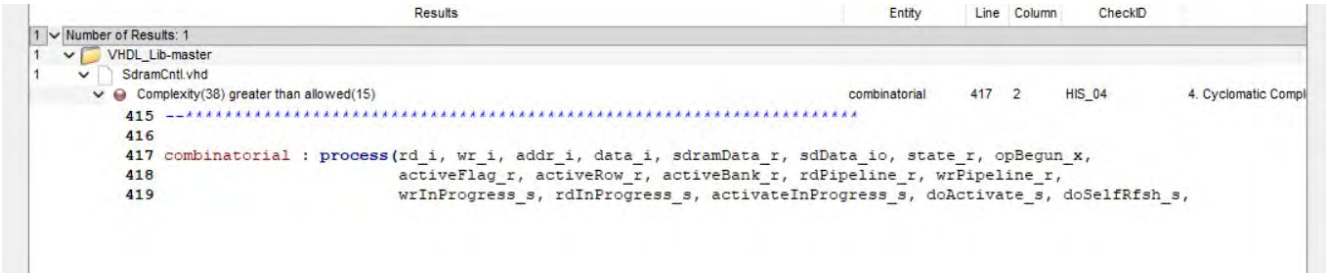

Рисунок 3.6 – Фіксація надмірної складності для файлу SdramCntl.vhd у SciTools Understand

Для верифікації такої структури файл було імпортовано до середовища HDL Designer. За допомогою інструменту аналізу діаграм заплутаний VHDL-код автомата було автоматично візуалізовано у вигляді графу станів. На основі цієї графічної моделі сформовано детальну таблицю переходів між штатними режимами (ініціалізація, читання, запис, рефреш) та службовими станами.

Рисунок 3.7 – Приклад візуалізації графу та формалізації переходів FSM в HDL Designer для процедури Code Review

Четвертим прикладом є аналіз модуля SDCard.vhd. Статичне сканування у середовищі SciTools Understand зафіксувало для цього файла найвищий показник цикломатичної складності – 60, що в чотири рази перевищує встановлений ліміт (15). Це підтверджує наявність величезного масиву заплутаних зв'язків та переходів у структурі автомата.

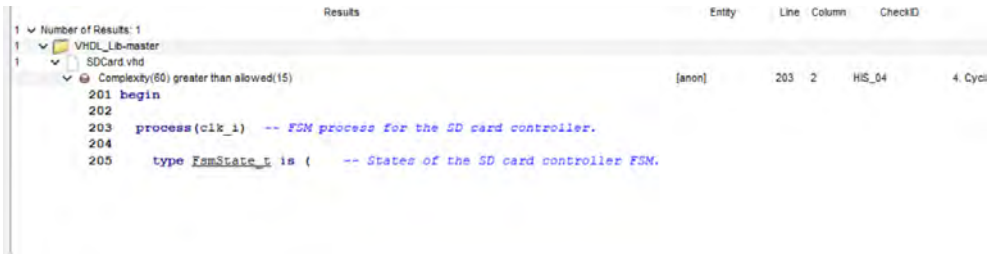

Рисунок 3.8 – Фіксація критичної складності для файлу SDCard.vhd у SciTools Understand

Для усунення виявлених дефектів та спрощення структури в HDL Designer було застосовано інструмент функційного групування станів. На основі графічного аналізу FSM блоки станів було розділено на окремі підгрупи: ініціалізація карти,

передавання команд, приймання відповідей та службове формування сигналів. Такий розподіл дозволяє виконувати процедуру Code Review не як суцільний перегляд великого оператора case, а як послідовну перевірку окремих логічних блоків.

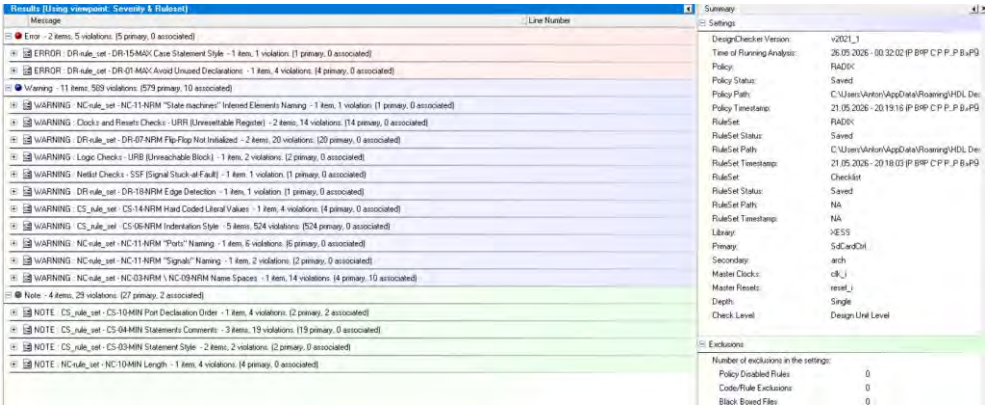

Рисунок 3.9 – Функційне групування станів FSM в HDL Designer

Узагальнення результатів практичного застосування контрольних пунктів наведено в таблиці 3.3.

Таблиця 3.3 – Результати практичного застосування контрольних пунктів до VHDL-файлів

Файл	Виявлений ризик	Застосований контрольний пункт	Запропоноване вдосконалення	Очікуваний ефект
MemTest.vhd	Штатний завершальний стан обробляється через when others	Перевірка повноти case	Явно виділити завершальний стан в окрему гілку case	Підвищення читабельності та прозорості FSM
FlashCntl.vhd	Гілка when others не задає визначеної реакції	Перевірка коректності case та нештатної поведінки FSM	Додати перехід у визначений початковий або безпечний стан	Підвищення передбачуваності поведінки автомата
SdramCntl.vhd	Складна логіка переходів між станами	Перевірка досяжності станів FSM	Побудувати таблицю або граф переходів	Спрощення code review та перевірки досяжності станів
SDCard.vhd	Велика кількість основних і службових станів	Перевірка структури FSM	Згрупувати стани за функційним призначенням	Підвищення структурованості аналізу автомата

Порівняння стану фрагментів до та після застосування удосконаленої процедури наведено в таблиці 3.4.

Таблиця 3.4 – Порівняння результатів до та після вдосконалення

Приклад	До вдосконалення	Після вдосконалення	Результат
MemTest.vhd	Штатний стан FSM фактично прихований у when others	Стан описаний окремою гілкою case	Логіка автомата стала зрозумілішою
FlashCntl.vhd	Нештатна гілка не задає активної реакції	Запропоновано визначений recovery-перехід	Поведінка автомата стала більш передбачуваною
SdramCntl.vhd	Перевірка переходів виконується загальним переглядом коду	Запропоновано таблицю переходів FSM	Review переходів став формалізованим
SDCard.vhd	Великий FSM аналізується як суцільний оператор case	Запропоновано групування станів	Аналіз автомата став структурованішим

Отримані результати показують, що удосконалена процедура SCA/CR дозволяє не лише виявляти потенційні недоліки VHDL-коду, а й формувати конкретні рекомендації щодо їх усунення. Для компактних модулів доцільними є локальні зміни у фрагментах case, а для складних модулів – процедурне вдосконалення аналізу шляхом побудови таблиць переходів, групування станів і застосування інструментальної оцінки складності.

3.4 Економічне обґрунтування впровадження удосконаленої процедури

Економічне обґрунтування впровадження удосконаленої процедури SCA/CR виконується з метою оцінювання доцільності застосування запропонованого контрольного блоку для аналізу VHDL-коду цифрових автоматів. Основний економічний ефект від упровадження процедури полягає у скороченні трудовитрат на code review, зменшенні кількості повторних перевірок і ранньому виявленні дефектів, пов’язаних із FSM-логікою, операторами case, reset-поведінкою та складністю VHDL-файлів.

Відповідно до методичних рекомендацій, економічне обґрунтування результатів дослідження у галузі інформаційних технологій може базуватися на оцінюванні витрат, очікуваного ефекту та доцільності впровадження запропонованого рішення [16]. У межах цієї роботи запропонованим рішенням є не новий програмний продукт, а вдосконалення процедури перевірки VHDL-коду шляхом додавання контрольного блоку для цифрових автоматів. Тому основним показником ефективності є зменшення часу, необхідного для аналізу коду та підготовки зауважень за результатами SCA/CR.

Процедура VHDL SCA/CR передбачає планування часу на code review залежно від обсягу VHDL-коду. У процедурі зазначено, що на виконання VHDL CR та аналіз результатів SCA/CR рекомендовано планувати час відповідно до обсягу коду – орієнтовно 100–150 рядків коду на годину [14]. Це дає змогу використати трудомісткість перевірки як базовий показник для економічного розрахунку.

До впровадження удосконаленої процедури перевірка VHDL-файлів виконується переважно шляхом загального перегляду коду. У такому випадку рецензент витрачає додатковий час на пошук цифрових автоматів, визначення станів, аналіз гілок case, перевірку reset-логіки та виявлення фрагментів із підвищеною складністю. Після впровадження контрольного блоку перевірка стає більш формалізованою: для кожного файлу заздалегідь визначаються контрольні пункти, а результати фіксуються у таблиці SCA/CR. Це скорочує час review і зменшує ймовірність пропуску типових дефектів.

Економічний ефект від упровадження удосконаленої процедури можна визначити за формулою:

$$E = \left(T_{\text{до}} - T_{\text{після}}\right) \times C_{\text{год}} \times N, \tag{3.1}$$

де E– економічний ефект від упровадження процедури, грн;

$T_{\text{до}}$– трудомісткість перевірки до впровадження контрольного блоку, год;

$T_{\text{після}}$– трудомісткість перевірки після впровадження контрольного блоку, год;

$C_{\text{год}}$– вартість однієї години роботи спеціаліста, грн/год;

N– кількість ітерацій перевірки або кількість модулів, до якихзастосовується процедура.

Для розрахунку прийнято, що до впровадження контрольного блоку рецензент перевіряє VHDL-код із продуктивністю приблизно 100 рядків на годину. Після впровадження удосконаленої процедури продуктивність аналізу підвищується до 150 рядків на годину за рахунок використання формалізованого чек-листа, попереднього виділення FSM-фрагментів, фіксації типових ризиків і використання результатів Scientific Toolworks Understand.

Для демонстраційного розрахунку прийнято обсяг аналізованого VHDL-коду $L = 1500$ рядків. Такий обсяг відповідає перевірці кількох вибраних модулів бібліотеки, що містять цифрові автомати та складну керуючу логіку.

Трудомісткість перевірки до впровадження процедури:

$$T_{\text{до}} = \frac{L}{Q_{\text{до}}} = \frac{1500}{100} = 15 \text{ год.}$$

Трудомісткість перевірки після впровадження процедури:

$$T_{\text{після}} = \frac{L}{Q_{\text{після}}} = \frac{1500}{150} = 10 \text{ год.}$$

Економія часу становить:

$$\Delta T = T_{\text{до}} - T_{\text{після}} = 15 - 10 = 5 \text{ год.}$$

За умовної вартості години роботи спеціаліста $C_{\text{год}} = 250$ грн/год економічний ефект для однієї ітерації перевірки становить:

$$E = 5 \times 250 = 1250 \text{ грн.}$$

Результати розрахунку наведено в таблиці 3.5. Оскільки процедура SCA/CR може виконуватися неодноразово, зокрема після внесення змін у VHDL-код, економічний ефект зростає зі збільшенням кількості ітерацій перевірки. Якщо

прийняти, що протягом роботи з проєктом виконується три ітерації аналізу, загальний економічний ефект становитиме:

$$E_{\text{заг}} = 1250 \times 3 = 3750 \text{ грн.}$$

Таблиця 3.5 – Розрахунок економії часу від упровадження удосконаленої процедури

Показник	Позначення	Значення
Обсяг VHDL-коду для аналізу	L	1500 рядків
Продуктивність перевірки до вдосконалення	$Q_{\text{до}}$	100 рядків/год
Продуктивність перевірки після вдосконалення	$Q_{\text{після}}$	150 рядків/год
Трудомісткість до вдосконалення	$T_{\text{до}}$	15 год
Трудомісткість після вдосконалення	$T_{\text{після}}$	10 год
Економія часу	ΔT	5 год
Вартість години роботи спеціаліста	$C_{\text{год}}$	250 грн/год
Економічний ефект за одну ітерацію	E	1250 грн

Для впровадження удосконаленої процедури також необхідні певні організаційні витрати. До них належать підготовка чек-листа перевірки, налаштування шаблону таблиці SCA/CR, попереднє визначення контрольних пунктів і підготовка прикладів оформлення результатів. Приймаючи витрати на підготовку процедури на рівні 3 годин роботи спеціаліста, отримаємо:

$$C_{\text{впров}} = 3 \times 250 = 750 \text{ грн.}$$

Чистий економічний ефект після трьох ітерацій перевірки становить:

$$E_{\text{чист}} = E_{\text{заг}} - C_{\text{впров}} = 3750 - 750 = 3000 \text{ грн.}$$

Результати узагальненого розрахунку наведено в таблиці 3.6.

Таблиця 3.6 – Узагальнений економічний ефект від упровадження удосконаленої процедури

Показник	Значення
Економічний ефект за одну ітерацію перевірки	1250 грн
Кількість ітерацій перевірки	3 од
Загальний економічний ефект	3750 грн
Витрати на підготовку чек-листа та шаблону перевірки	750 грн
Чистий економічний ефект	3000 грн

Крім прямого економічного ефекту, удосконалена процедура забезпечує непрямий ефект. Він полягає у підвищенні якості VHDL-коду, зменшенні ризику

пропуску дефектів цифрових автоматів, скороченні кількості повторних виправлень і підвищенні зручності супроводу проєкту. Особливо важливим є те, що типові проблеми FSM-коду виявляються на ранньому етапі, до функційного тестування або подальшої інтеграції модуля в більший FPGA-проєкт.

Удосконалена процедура також зменшує залежність результатів code review від суб'єктивного досвіду окремого рецензента. За наявності контрольного блоку перевірка виконується за єдиною схемою: досяжність станів, початковий стан і reset, повнота case, складність файлу. Це дає змогу стандартизувати результати перевірки та спростити документування виявлених зауважень.

Таким чином, економічне обґрунтування підтверджує доцільність упровадження удосконаленої процедури SCA/CR. Запропонований контрольний блок потребує незначних організаційних витрат на підготовку, однак дозволяє скоротити трудомісткість перевірки, підвищити якість аналізу VHDL-коду та зменшити ризик виявлення дефектів на пізніх етапах розробки. Отриманий розрахунок показує, що навіть за умовної оцінки чистий економічний ефект після кількох ітерацій перевірки становить 3000 грн, а додатковий якісний ефект проявляється у підвищенні надійності, читабельності та супроводжуваності VHDL-коду.

ВИСНОВКИ

У результаті виконання даної роботи було досліджено особливості процесу кодування та верифікації цифрових систем, реалізованих мовою опису апаратури VHDL, а також проаналізовано сучасні підходи до підвищення ефективності розробки HDL-проєктів.

У першому розділі роботи було розглянуто теоретичні основи мови VHDL, її ключові концепції та області застосування. Встановлено, що VHDL є потужним інструментом для моделювання та реалізації цифрових систем, який дозволяє працювати на різних рівнях абстракції. Разом із тим було визначено основні проблеми традиційного підходу до розробки HDL-проєктів, зокрема складність структури систем, труднощі виявлення помилок, обмеженість симуляційного підходу та відсутність ефективних засобів статичного аналізу.

Також було проведено огляд сучасних інструментів розробки та аналізу HDL-коду. Особливу увагу приділено середовищу HDL Designer та інструменту Scientific Toolworks Understand. Встановлено, що використання таких засобів дозволяє значно підвищити ефективність розробки, забезпечити кращу структуризацію проєктів та покращити якість коду. На основі аналізу підходів до верифікації було зроблено висновок, що найбільш ефективним є комбінований підхід, який поєднує симуляцію, статичний аналіз та перевірку коду.

У другому розділі було сформульовано основні вимоги до процесу розробки VHDL-проєктів, серед яких ключовими є читабельність, модульність, можливість повторного використання компонентів та контроль якості коду. Було описано структуровану процедуру кодування, що базується на декомпозиції системи, формуванні інтерфейсів та поетапній реалізації функціональних блоків.

Окрему увагу приділено інтеграції HDL Designer у процес розробки, що дозволяє перейти до графічного та ієрархічного проєктування систем. Встановлено, що використання цього інструменту забезпечує покращення організації проєкту, зменшення кількості помилок на етапі створення структури та підвищення зручності роботи з великими системами. Також було розглянуто

можливості Scientific Toolworks Understand, який забезпечує глибокий статичний аналіз коду, дозволяє виявляти залежності між модулями та оцінювати складність системи без необхідності її виконання.

У третьому розділі роботи було запропоновано удосконалений підхід до кодування та верифікації VHDL-проєктів, що базується на інтеграції інструментів HDL Designer 2021.1 та Scientific Toolworks Understand у єдиний процес розробки. Основною ідеєю запропонованого підходу є перехід від традиційного коду-орієнтованого методу до структурно-орієнтованого, який передбачає попереднє формування архітектури системи, автоматизовану генерацію коду та виконання статичного аналізу на ранніх етапах.

Запропонований підхід дозволяє:

- зменшити кількість помилок за рахунок їх раннього виявлення;
- підвищити якість та читабельність коду;
- покращити керованість проєкту;
- скоротити час розробки;
- забезпечити кращу масштабованість системи.

Практична значущість роботи полягає у можливості використання розробленого підходу при створенні цифрових систем різної складності, зокрема при проєктуванні FPGA та ASIC. Запропонована методика може бути застосована як у навчальних цілях, так і в інженерній практиці.

Таким чином, поставлена мета роботи досягнута, а отримані результати підтверджують доцільність використання сучасних інструментів та структурованого підходу до розробки VHDL-проєктів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. IEEE Std 1076-2019. IEEE Standard for VHDL Language Reference Manual. IEEE, 2019. URL: https://standards.ieee.org/ieee/1076/5179/ (дата звернення: 18.04.2026).

2. IEEE Std 1076-2008. IEEE Standard VHDL Language Reference Manual. IEEE, 2008. URL: https://standards.ieee.org/ieee/1076/3666/ (дата звернення: 18.04.2026).

3. Ashenden P. J. The Designer’s Guide to VHDL. Morgan Kaufmann, 2008. URL: https://www.sciencedirect.com/book/monograph/9780120887859/the-designers-guide-to-vhdl (дата звернення: 18.04.2026).

4. Pedroni V. A. Circuit Design with VHDL. MIT Press, 2020. URL: https://mitpress.mit.edu/9780262042642/circuit-design-with-vhdl/ (дата звернення: 18.04.2026).

5. Chu P. P. FPGA Prototyping by VHDL Examples. Wiley, 2008. URL: https://onlinelibrary.wiley.com/doi/book/10.1002/9780470231630 (дата звернення: 18.04.2026).

6. Brown S., Vranesic Z. Fundamentals of Digital Logic with VHDL Design. McGraw-Hill. URL: https://books.google.com/books/about/Fundamentals_of_Digital_Logic_with_VHDL.html?id=CDpGAQAAIAAJ (дата звернення: 18.04.2026).

7. Zwolinski M. Digital System Design with VHDL. Pearson Education, 2004. URL: https://books.google.com/books/about/Digital_System_Design_with_VHDL.html?id=16usKXZiCNMC (дата звернення: 18.04.2026).

8. Perry D. L. VHDL: Programming by Example. McGraw-Hill, 2002. URL: https://books.google.com/books/about/VHDL.html?id=PBYgxSATL5YC (дата звернення: 18.04.2026).

9. Wakerly J. F. Digital Design: Principles and Practices. Pearson. URL: https://books.google.com/books/about/Digital_Design.html?id=DXQYlCoyvTYC (дата звернення: 18.04.2026).

10. Gajski D. D. Principles of Digital Design. Prentice Hall, 1997. URL: https://philpapers.org/rec/GAJPOD (дата звернення: 18.04.2026).

11. Siemens EDA. HDL Designer. Official product information. URL: https://www.siemens.com/en-us/products/ic/hdl-designer/
(дата звернення: 18.04.2026).

12. Siemens EDA. HDL Author. Official product information. URL: https://www.siemens.com/en-us/products/ic/questa-one/design-solutions/hdl-author/
(дата звернення: 18.04.2026).

13. SciTools. Understand: The Software Developer’s Multi-Tool. URL: https://scitools.com/ (дата звернення: 18.04.2026).

14. SciTools. Understand Metrics. URL: https://scitools.com/metrics (дата звернення: 18.04.2026).

15. SciTools. Understand User Guide and Reference Manual. URL: https://docs.scitools.com/manuals/pdf/understand.pdf (дата звернення: 18.04.2026).

16. Intel Corporation. Recommended HDL Coding Styles. Intel Quartus Prime Pro Edition User Guide: Design Recommendations. URL: https://manuals.plus/m/69e302ef92f8e3ac9555cadc83803985d81c707dfb3ec9bfd8e669abfc232b44 (дата звернення: 18.04.2026).

17. Intel Corporation. Intel Quartus Prime Pro Edition User Guide: Design Recommendations. URL: https://www.intel.com/programmable/technical-pdfs/683082.pdf (дата звернення: 18.04.2026).

18. AMD Xilinx. HDL Coding Practices to Accelerate Design Performance. WP231. URL: https://docs.amd.com/v/u/en-US/wp231 (дата звернення: 18.04.2026).

19. AMD Xilinx. Vivado Design Suite User Guide: Synthesis. UG901. URL: https://docs.amd.com/r/en-US/ug901-vivado-synthesis (дата звернення: 18.04.2026).

20. AMD Xilinx. Vivado Design Suite User Guide: Design Analysis and Closure Techniques. UG906. URL: https://docs.amd.com/r/en-US/ug906-vivado-design-analysis (дата звернення: 18.04.2026).

21. U.S. Nuclear Regulatory Commission. NUREG/CR-7006: Review Guidelines for Field-Programmable Gate Arrays in Nuclear Power Plant Safety Systems. NRC, 2010. URL: https://www.nrc.gov/reading-rm/doc-collections/nuregs/contract/cr7006/index (дата звернення: 18.04.2026).

22. IEC 62566:2012. Nuclear Power Plants – Instrumentation and Control Important to Safety – Development of HDL-Programmed Integrated Circuits for Systems Performing Category A Functions. URL: https://standards.iteh.ai/catalog/standards/iec/bfef3e19-c4c2-493a-ab78-163478a25595/iec-62566-2012 (дата звернення: 18.04.2026).

23. IEC 62566-2:2020. Nuclear Power Plants – Instrumentation and Control Important to Safety – Development of HDL-Programmed Integrated Circuits – Part 2. URL: https://webstore.iec.ch/en/publication/30737 (дата звернення: 18.04.2026).

24. IEC 61508-3:2010. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems – Part 3: Software Requirements. URL: https://webstore.iec.ch/en/publication/5517 (дата звернення: 18.04.2026).

25. IEC 61508-7:2010. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems – Part 7: Overview of Techniques and Measures. URL: https://webstore.iec.ch/en/publication/5521 (дата звернення: 18.04.2026).

26. IEC 60880:2006. Nuclear Power Plants – Instrumentation and Control Systems Important to Safety – Software Aspects for Computer-Based Systems Performing Category A Functions. URL: https://webstore.iec.ch/en/publication/3795 (дата звернення: 18.04.2026).

27. IEC 62138:2018. Nuclear Power Plants – Instrumentation and Control Systems Important to Safety – Software Aspects for Computer-Based Systems Performing Category B or C Functions. URL: https://webstore.iec.ch/en/publication/30172 (дата звернення: 18.04.2026).

28. IEC 61226:2009. Nuclear Power Plants – Instrumentation and Control Important to Safety – Classification of Instrumentation and Control Functions. URL: https://webstore.iec.ch/en/publication/4941 (дата звернення: 18.04.2026).

29. IAEA. Design of Instrumentation and Control Systems for Nuclear Power Plants. IAEA Safety Standards Series No. SSG-39. Vienna: IAEA, 2016. URL: https://www-pub.iaea.org/MTCD/Publications/PDF/Pub1694_web.pdf (дата звернення: 18.04.2026).

30. IAEA. Computer Security of Instrumentation and Control Systems at Nuclear Facilities. IAEA Nuclear Security Series No. 33-T. URL: https://www.iaea.org/publications/11184/computer-security-of-instrumentation-and-control-systems-at-nuclear-facilities (дата звернення: 18.04.2026).

31. IAEA. Application of Field Programmable Gate Arrays in Instrumentation and Control Systems of Nuclear Power Plants. IAEA Nuclear Energy Series No. NP-T-3.17. URL: https://www-pub.iaea.org/MTCD/Publications/PDF/Pub1701_web.pdf (дата звернення: 18.04.2026).

32. RadICS. RadICS Digital Safety I&C Platform. URL: https://radics.tech/ic-safety-platform-radics/ (дата звернення: 18.04.2026).

33. Radiy. RadICS Platform. URL: https://radiy.com/en/radics-platform/ (дата звернення: 18.04.2026).

34. World Nuclear Association. Hardware Description Language Programmed Device Technology in Nuclear Power Plants. URL: https://world-nuclear.org/our-association/publications/working-group-reports/hardware-description-language-%28hdl%29-programmed-dev (дата звернення: 18.04.2026).

35. XESS Corp. VHDL_Lib: Library of VHDL Components That Are Useful in Larger Designs. GitHub repository. URL: https://github.com/xesscorp/VHDL_Lib (дата звернення: 18.04.2026).

36. XESS Corp. MemTest.vhd. VHDL_Lib repository. URL: https://github.com/xesscorp/VHDL_Lib/blob/master/MemTest.vhd (дата звернення: 18.04.2026).

37. XESS Corp. FlashCntl.vhd. VHDL_Lib repository. URL: https://github.com/xesscorp/VHDL_Lib/blob/master/FlashCntl.vhd (дата звернення: 18.04.2026).

38. XESS Corp. SdramCntl.vhd. VHDL_Lib repository. URL: https://github.com/xesscorp/VHDL_Lib/blob/master/SdramCntl.vhd (дата звернення: 18.04.2026).

39. XESS Corp. SDCard.vhd. VHDL_Lib repository. URL: https://github.com/xesscorp/VHDL_Lib/blob/master/SDCard.vhd (дата звернення: 18.04.2026).

40. Копішинська О. П., Уткін Ю. В., Дивнич О. Д. Методичні рекомендації щодо виконання кваліфікаційної роботи для здобувачів вищої освіти за освітньо-професійною програмою “Інформаційні управляючі системи” спеціальності 126 Інформаційні системи та технології. Полтава: ПДАУ, 2022. URL: https://www.pdau.edu.ua/sites/default/files/node/10014/metodrekomendkvalifrobitbak2022.pdf (дата звернення: 18.04.2026).