Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет __Комп'ютерної інженерії та управління_____
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки_____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти ____другий (магістерський)_____
(рівень вищої освіти)

Система моделювання комбінаційних логічних схем
(тема)

Виконав: студент 2 курсу, групи СКСм-20-1

_____Солодухіна К.Є._____
(прізвище, ініціали)

Спеціальність *123 _Комп'ютерна інженерія*

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

Освітня програма_____

*Спеціалізовані комп'ютерні системи*_____
(повна назва освітньої прграми)

Керівник роботи проф. Хаханова І.В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Чумаченко С.В.
(підпис) (прізвище, ініціали)

2021 р.

Харківський національний університет радіоелектроніки

Факультет_____Комп'ютерної інженерії та управління _____

Кафедра_____Автоматизації проектування обчислювальної техніки_____

Рівень вищої освіти____другий (магістерський)_____

Спеціальність_____123_ Комп'ютерна інженерія _____
(шифр і назва)

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

Освітня програма _____Спеціалізовані комп'ютерні системи_____
(повна назва

ЗАТВЕРДЖУЮ:

Зав. кафедри ___  ___
(підпис)

«__04__»___11____ 2021 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові_____Солодухіній Катерині Євгеніївні_____
(прізвище, ім'я, по батькові)

1. Тема роботи (проекту)_ Система моделювання комбінаційних логічних схем _____

_____

Combinational logic circuits design system _____

_____

затверджена наказом по університету від «_04_»__11__2021 р. № _1635 Ст_____

2. Термін подання студентом роботи до екзаменаційної комісії _____22.12.2021_____

3. Вихідні дані до роботи (проекту)_____

Мова C#_____

SchemeCreator_____

Комбінаційні схеми_____

_____

4. Перелік питань, що потрібно опрацювати у роботі _____

Аналіз існуючих рішень архітектури програми._____

Аналіз алгоритмів роботи з комбінаційними логічними схемами: нумерація__ ліній, моделювання, розміщення елементів схеми._____

Адаптація та реалізація алгоритмів : нумерації ліній, моделювання,_____ розміщення розміщення елементів схеми._____

_____

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) _30 слайдів_ _____
_____
_____
_____
_____

6. Консультанти розділів роботи (проекту)

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

7. Дата видачі завдання 02.09.2021_____

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи (проекту)	Термін виконання етапів роботи	Примітка
1	Видача теми проекту, узгодження і затвердження теми	02.09.2021-08.09.2021	
2	Аналіз існуючих рішень архітектури програми	09.09.2021-01.10. 2021	
3	Аналіз алгоритмів роботи з комбінаційними логічними схемами: нумерація ліній, моделювання, розміщення елементів схеми	01.10.2021-31.10.2021	
4	Адаптація та реалізація алгоритмів: нумерації ліній, моделювання, розміщення розміщення елементів схеми	1.11.2021-30.11.2021	
5	Оформлення пояснювальної записки	01.12. 2021-15.12.2021	
6	Захист проекту	20.12. 2021-25.12. 2021	

Студент _____
(підпис)

Керівник роботи (проекту) _____ проф. Хаханова І.В.___
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Записка пояснювальна містить 84 сторінки., 21 рисунок, 15 джерел.

АЛГОРИТМИ РОЗМІЩЕННЯ, АЛГОРИТМИ ТРАСУВАННЯ, АЛГОРИТМИ МОДЕЛЮВАННЯ, КОМБІНАЦІЙНА СХЕМА, ЛОГІЧНИЙ ЕЛЕМЕНТ, КОМП'ЮТЕРНІ ПРОГРАМИ, БУЛЕВА АЛГЕБРА

Метою кваліфікаційної роботи є вивчення та аналіз методів побудови програм з точки зору архітектури задля підвищення якості програмного забезпечення.

Іншею задачею даного проекту є проведення аналізу існуючих, та, у разі необхідності, модифікації, чи створення власних алгоритмів, що виконують розміщення елементів, трасування ліній комбінаційної логчної схеми та її подальше моделювання.

Результатом кваліфікаційного проектування є розроблені і задокументовані програмні рішення для використання у програмах, що використовуються для демонстрації роботи комбінаційних логічних схем. Крім того, результатом даного кваліфікаційного проекту також є аналіз і пропонована архітектура для створення такого додатку.

ABSTRACT

Explanatory note: 84 pages, 21 figures, 15 sources.

PLACEMENT ALGORITHMS, TRACING ALGORITHMS, SIMULATION ALGORITHMS, COMBINATION SCHEME, LOGICAL ELEMENT, COMPUTER PROGRAMS, BOOLEAN ALGEBRA

The purpose of the qualification work is to study and analyze methods of building programs in terms of architecture to improve the quality of software.

Another task of this project is to analyze the existing and, if necessary, modify or create your own algorithms that perform the placement of elements, tracing the lines of the combinational logic circuit and its further modeling.

The result of the course design is developed and documented software solutions for use in programs used to demonstrate the operation of combinational logic circuits. In addition, the result of this course project is also an analysis and the proposed architecture for creating such an application.

ЗМІСТ

Оглавление

РЕФЕРАТ 4
ABSTRACT 5
ЗМІСТ 6
ВСТУП 8
1 ОПИС ТА ПРИНЦИП ФУНКЦІОНУВАННЯ 10
1.1 Вимоги та критерії 10
1.2 Практичне застосування 11
1.3 Мета, об’єкт та предмет дослідження 12
1.4 Приклади існуючих програм 12
1.4.1 Active-HDL 12
1.4.2 Proteus 13
1.1 Presentation Layer (рівень презентації) 14
1.2 Business Layer (рівень бізнес-логіки) 14
1.3 Database Layer (рівень даних) 15
2 АЛГОРИТМИ РОЗМІЩЕННЯ 16
2.1 Конструктивні алгоритми розміщення 16
2.2 Послідовні алгоритми розміщення по зв'язності 16
2.3 Паралельно-послідовне розміщення. Метод зворотного розміщення. .. 17
2.4 Ітераційні алгоритми розміщення 18
2.5 Алгоритм парних перестановок 18
3 ІМПЛЕМЕНТАЦІЯ АЛГОРИТМУ РОЗМІЩЕННЯ 23
3.1 Адаптація та імплементація алгоритму 23
3.2 Клас Aligner 25
3.3 Розміщення зовнішніх вихідних портів 29
3.4 Розміщення логічних елементів 31
3.5 Розміщення зовнішніх вхідних портів 36
4 РОЗРОБКА АРХІТЕКТУРИ ПРОЕКТУ 41
4.1 Критерії гарної архітектури 41
4.2 Ефективність системи 41

4.3 Гнучкість системи 41
4.4 Розширюваність системи 42
4.5 Масштабованість процесу розробки 43
4.6 Тестованість 43
4.7 Можливість повторного використання 43
4.8 Супроводжуваність 43
5 РОЗРОБКА PRESENTATION LAYER 45
5.1 Функціональність програми 45
5.2 Елементи меню 46
5.2.1 Діалог CustomGateDialog 46
5.2.2 Діалог NewExternalPortDialog 47
5.2.3 Діалог NewGateDialog 47
5.2.3 Діалог New Message 48
5.3 Динамічні елементи 48
5.4 Контейнери 51
5.5 Масштабування 53
6 РОЗРОБКА BUSINESS LAYER 56
6.1 Бібліотека стандартних елементів 56
6.2 Конструктор логічного елементу 56
6.3 Створення власних елементів 57
6.4 Алгоритм нумерації ліній 59
6.4.1 Початкова валідація 60
6.4.2 Нумерація вхідних елементів 61
6.4.3 Нумерація вихідних портів 62
6.4.4 Зовнішній цикл 62
6.4.5 Нумерація ліній 63
6.4.6 Нумерація логічних елементів 65
6.5 Алгоритм моделювання 66
6.5.1 Компілятивний спосіб 66
6.5.2 Подієвий спосіб нумерації ліній 67
6.5.3 Розробка власного алгоритму 67
6.5.4 Реалізація алгоритму за допомогою коду 69

6.6 Синхронізація з Presentation Layer 73
6.7 Синхронізація з Storage Layer 73
7 РОЗРОБКА DATABASE LAYER 75
7.1 Серіалізація та десериалізація 75
7.2 Збереження у базі даних 75
7.3 Збереження у файлі налаштування програми 77
7.4 Збереження у оперативній пам’яті 77
7.5 Автосбереження 77
7.6 Реалізація збереження у програмі 78
7.6 Реалізація завантаження у програмі 79
ВИСНОВОК 82
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 83

ВСТУП

У наш час проблема правильної та коректної розробки спеціалізованих програм для персональних комп’ютерів та мобільних телефонів стає все більш актуальною. Підставою для виникнення цієї проблеми є велике використання персональних комп'ютерів, особливо популярізація серед молоді. Завдяки саме молоді, а саме студентам наступна проблема, проблема створення освітніх програм, залишається актуальною.

Сучасні програми, що здатні симулювати роботу логічних елементів та схем, є набагато більш складними для ознайомлення, мають велику кількість додаткових функцій та потребують багато ресурсів процесора та місця на комп’ютері. Крім того, загальною проблемою є також інтерфейс користувача, що не оновлюється, та обмеження на платформи, на яких програма може коректно функціонувати, тобто непереносність на інші девайси.

Більшість таких програм є спеціалізованую і через це виникають і інши проблеми: складний інтерфейс і велика вартість.

Цільовою аудиторією цієї програми, що розглядається, є люди, які хочуть ознайомитися з приницпами роботи логічних елементів та схем, які можно побудувати на їх основі. Приклади лістингів написані на мові програмування C# версії 9.0.

У даному кваліфікаційному проекті описується теоретичнй спосіб створення програми, що проводить моделювання логічних елементів та сигналів. Програма може використовуватися як довідник функціонування логічних елементів.

Проблема сучасної розробки комбінаційних та послідовних схем стоїть доволі гостро – електронні пристрої все більше застосовуються у повсякденному житті, стають все складнішими та довершенішими, а тому кількість необхідних для їхнього функціонування елементів все збільшується.

З цього випливає, що і потрібність у програмах, що модулюють логічні

схеми та у системах автоматизованого проектування теж лишається доволі високою, адже створення фізичного пристрою є дуже витратним процесом, тому перш ніж створювати пристрій, необхідно створити його віртуальний прототип у спеціальних програмах та провести його тестування та верифікацію.

Дана пояснювальна записка містить розгорнуту інформацію про функції даного системного додатку, принцип його функціонування, опис застосованих алгоритмів відображення, симуляції і трасування, а також логічних елементів.

1 ОПИС ТА ПРИНЦИП ФУНКЦІОНУВАННЯ

1.1 Вимоги та критерії

Метою даної роботи є дослідження алгоритмів роботи з комбінаційними логічними схемами та аналіз архітектури для побудови програм для редагування таких схем.

До таких алгоритмів належать наступні алгоритми: алгоритм нумерації ліній, алгоритм моделювання роботи схеми та алгоритм корекції розміщення елементів схеми.

Розробка таких алгоритмів, звісно, передбачає створення прототипу програми, що буде їх реалізувати задля випробування на практиці, тестування та оцінки коректності та оперативності роботи.

Створені у прототипі алгоритми дозволяють проводити наступні операції:

1. створювати комбінаційні схеми за допомогою ліній та бібліотеки логічних елементів;

2. редагувати схеми;

3. проводити вирівнювання елементів схеми;

3. проводити трасування ліній;

4. симулювати роботу логічних елементів, ліній та схем у цілому;

5. зберігати та завантажувати схеми з файлів;

Алгоритми відтворюють роботу ліній та різних логічних елементів, в яких можна конфігурувати кількість інформаційних входів.

Створена схема може бути збережена та завантажена, її лінії можуть бути пронумеровані, а елементи – вирівнені згідно ієрархії, також є можливість проведення симуляції передачі електричних сигналів з входів на виходи схеми згідно законам роботи логічних елементів, що входять у склад схеми, та ліній, що їх з’єднують.

Для зручного доступу з різних пристроїв, тобто реалізації мультиплатформеності, програма була розроблена з використанням платформи UWP. Таким чином, програму можна використовувати, згідно з документацією Microsoft, на таких пристроях, як:

- персональний комп’ютер з встановленою операційною системою Windows 10;
- смартфони;
- планшети;
- Xbox One.

Однак, підтримка інших пристроїв, крім Windows 10 потребує використання Universal Windows Platform Bridges.

Мінімальними вимогами, для коректного функціонування програми на персональних комп’ютерах, таким чином, є операційна система Windows 10.

Реалізацією дослідження даної роботи є розробка програмного забезпечення, призначеного для моделювання комбінаційних логічних елементів, нумерації її ліній та її вирівнювання.

Програма дозволяє моделювати комбінаційні логічні елементи з можливістю їх подальшого змінення. Програма відтворює роботу різних логічних елементів, в яких можна конфігурувати кількість інформаційних входів. Далі проводиться симуляція електричних сигналів на входах та виходах елементів за законами їх роботи.

Програма була розроблена з використанням платформи UWP. Таким чином, програму можно використовувати з операційної системи Windows 10.

Мінімальними вимогами для коректного функціонування програми на персональних комп’ютерах є операційна система Windows 10.

1.2 Практичне застосування

Основними користувачами цієї програми є люди, які хочуть ознайомитися з приницпами роботи логічних елементів та схем, які можно

побудувати на їх основі. Більшість цих користувачів – студенти вищих освітніх закладів, коледжів та технікумів з освітньою програмой, що пов’язана зі створенням програмного та апаратного забезпечення, чи учні старших загальноосвітніх шкіл.

Програма може використовуватися як довідник з логічних елементів при кожному конкретному значенні вхідних сигналів та їх різній кількості.

1.3 Мета, об’єкт та предмет дослідження

Об'єктом наукового дослідження у даній роботі є комбінаційні логічні схеми та їх елементи.

Предметом дослідження є аналіз комбінаційних логічних схем та алгоритми аналізу та обробки зазначених схем та їх складових компонентів.

Для досягнення поставленої мети у даній роботі повинні бути вирішені наступні завдання:

1. проаналізувати існуючі алгоритми розміщення елементів схем.
2. проаналізувати існуючі алгоритми роботи зі схемами.
3. проаналізувати та розробити архітектуру програми.
4. створити власний алгоритм розміщення елементів схем.
5. реалізувати алгоритм у програмі згідно розробленій архітектурі.
6. розробити програму для побудови комбінаційних логічних схем.

У ході виконання кваліфікаційного проекту необхідно вивчити технології створення додатку на різних архітектурних рівнях та їх особливості, провести тестування отриманого додатку.

1.4 Приклади існуючих програм

1.4.1 Active-HDL

Active-HDL – середовище розробки, моделювання та верифікації

проектів для програмованих логічних інтегральних схем, розроблена фірмою Aldec. Перша версія програми вийшла в 1997 році.

Проблема використання Active-HDL полягає в тому, що вона є занадто складною для освоєння, має застарілий та інтерфейс (рис. 1.1). Крім того, програма може використовуватися тільки на операційній системі Windows, не оновлюється з 2010 року, потребує багато місця на диску та є платною.

Це обумовлено тим, що загальною метою створення програми є вирішення більш складних задач.

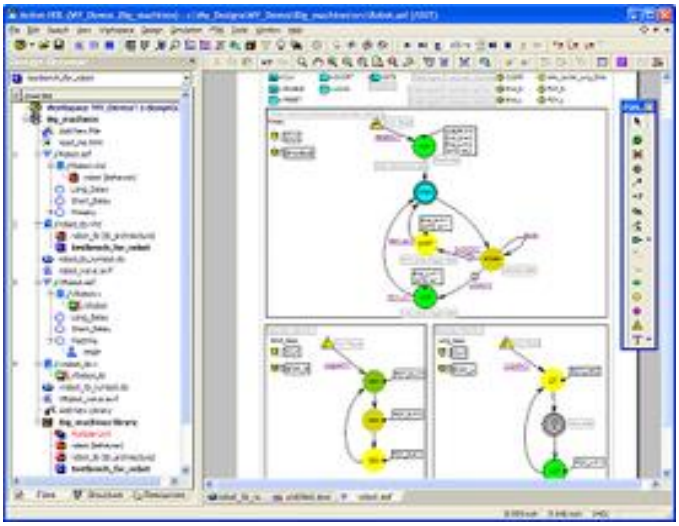


Рисунок 1.1 – Скріншот програми Active-HDL

1.4.2 Proteus

Proteus Design Suite – пакет програм для автоматизованого проектування (САПР) електронних схем. Розробка компанії Labcenter Electronics.

Proteus є більш зручною програмою для вивчення принципу роботи логічних елементів, але вона теж є занадто складною для освоєння, має застарілий інтерфейс (рис. 1.2). Крім того, програма може використовуватися тільки на операційній системі Windows та потреює багато місця на диску.

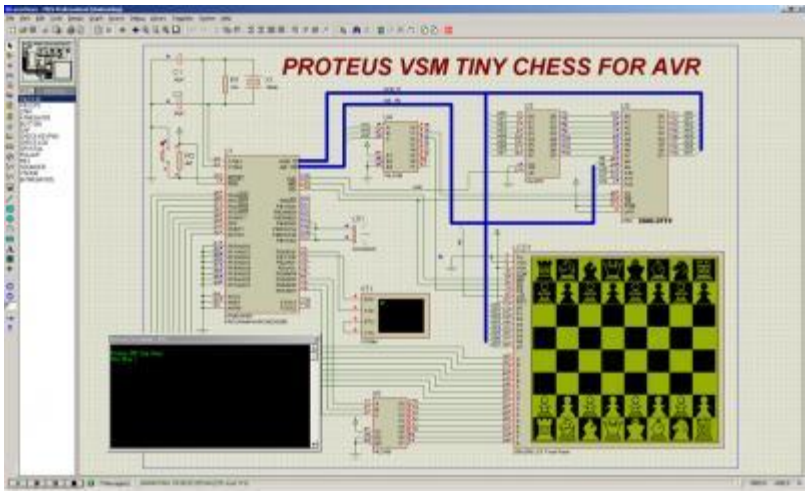


Рисунок 1.2 – Скріншот програми Proteus

Реалізація кожного проекту потребує певних алгоритмів та архітектурних рішень. У більшості випадків структуру програми можна поділити на наступні складові частини: [1]

1.1 Presentation Layer (рівень презентації)

Presentation Layer - це найвищий рівень програми. Рівень презентації відображає інформацію, пов’язану з такими послугами, як перегляд товарів, придбання та вміст кошика для покупок. Він взаємодіє з іншими рівнями, за допомогою яких видає результати на рівень браузера/клієнта та всі інші рівні в мережі. Простіше кажучи, це рівень, до якого користувачі можуть отримати безпосередній доступ (наприклад, веб-сторінка або графічний інтерфейс операційної системи).

1.2 Business Layer (рівень бізнес-логіки)

Business Layer контролює функціональність програми, виконуючи детальну обробку.

1.3 Database Layer (рівень даних)

Database Layer включає механізми збереження даних (сервери баз даних, спільні файли тощо) та рівень доступу до даних, який інкапсулює механізми збереження та виставляє дані. Рівень доступу до даних повинен забезпечувати API для рівня додатків, який надає методи управління збереженими даними без викриття, або створення залежностей від механізмів зберігання даних. Уникнення залежностей від механізмів зберігання дозволяє оновлювати або змінювати, не змінюючи, або, навіть, не знаючи про зміни клієнтів рівня додатків. Як і при поділі будь-якого рівня, існують витрати на впровадження, та часто витрати на продуктивність в обмін на покращену масштабованість та ремонтопридатність.

2 АЛГОРИТМИ РОЗМІЩЕННЯ

2.1 Конструктивні алгоритми розміщення

Серед конструктивних алгоритмів розміщення виділяють послідовні та паралельно-послідовні алгоритми.

У послідовному алгоритмі використовується n-кроковий процес прийняття рішення. На кожному кроці тут розміщується один елемент. У паралельно-послідовному алгоритмі кожному кроці розміщується група елементів (чи навіть елементи).

Серед послідовних алгоритмів розрізняють послідовні алгоритми розміщення зв'язків і матричні алгоритми. [2]

2.2 Послідовні алгоритми розміщення по зв'язності

Сутність цього багатокрокового алгоритму зводиться до послідовного розміщення чергового модуля (елемента) у певний вузол плати. Передбачається, що частина модулів (або хоча б один) заздалегідь розміщена на монтажній площині. В якості таких модулів можуть бути обрані контакти роз'єму, або модулі, фіксовані в певних позиціях відповідно до директивних вказівок розробника. При виборі чергового модуля оптимізується цільова функція, що враховує зв'язки цього модуля з безліччю розміщених раніше і нерозміщених модулів. Як така функція, наприклад, може бути обрана наступна:

$$J_{max} = OC = \sum_{j \in E_r^k} C_{ij} - \sum_{j \in E_n^k} C_{ij} \qquad (2.1)$$

де c_{ij} – елемент матриці суміжності ВНГ; E_r^k, E_n^k – відповідно безлічі

розміщених та нерозміщених на k–му кроці алгоритму модулів.

Завдання розміщення при цьому зводиться до максимізації оцінки по всіх модулях, що належать множині E_n^k.

Далі для вибраного модуля є найбільш прийнятна позиція на платі. Для вибору такої позиції використовується критерій мінімальності довжини зв'язків модуля, що розміщується з вже розміщеними модулями. При цьому, очевидно, немає необхідності розглядати всі незайняті на цьому кроці позиції, а достатньо оцінити лише велику кількість позицій, сусідніх із зайнятими. При цьому позиція p_i, в яку необхідно помістити і модуль, визначається з умови мінімізації наступного виразу:

$$F = \min_{p_i \in R^k} \sum_{j \in E_h^k} c_{ij} d_{p_j} \qquad (2.2)$$

де c_{ij} - елемент матриці суміжності ВНГ; R^k – безліч позицій, сусідніх із зайнятими, на k–му кроці. [3]

2.3 Паралельно-послідовне розміщення. Метод зворотного розміщення.

Для кожного з нерозміщених елементів ei, що належать E(I), $I = \overline{1, M}$ обчислюється деяка оцінка.

Обчислюється також деяка оцінка для кожного посадкового місця. Усі елементи та посадкові місця впорядковуються та здійснюється одночасне розміщення всіх елементів у позиції.

Нехай матриця $\mathrm{C} = \left|c_{ij}\right||m * n|$; $D = \left|d_{ij}\right||n * n|$ – матриці відстаней між позиціями.

Відповідно до зазначеного методу для кожного елемента ei розраховується сумарна кількість зв'язків i-го елемента з іншими частинами схеми $C_i = \sum_{ei \in E,\ i \neq j} C_{ij}$.

Для кожного посадкового місця обчислюється сумарна довжина відстаней j-ого посадкового місця з іншими позиціями $D_j = \sum_{p_i \in P, i \neq j} d_{ij}$.

Усі оцінки пов'язаності c_i впорядковуються за зростанням, а оцінки довжини d_{ij} - за зменшенням:

$$c_{i1} \leq c_{i2} \leq c_{i3} \leq \cdots \leq c_{in}$$
$$d_{j1} \leq d_{j2} \leq d_{j3} \leq \cdots \leq d_{jn} \quad (2.3)$$

елемент e_{i1} встановлюється позицію $Pj(1), e_{12} \rightarrow Pj(2)$ тощо. Отже, min скалярне множення двох векторів буде тоді, коли компоненти першого вектора впорядковані за зростанням, а елементи іншого за спаданням. [4]

2.4 Ітераційні алгоритми розміщення

Ці алгоритми припускають заданим деяке початкове розміщення, яке може бути отримано деяким конструктивним алгоритмом розміщення. Досліджуються кілька можливих варіантів розміщення близьких у певному сенсі до початкового та знаходять розміщення з меншим значенням цільової функції L сумарне. Знайдене розміщення приймається за вихідне та процес повторюється. Алгоритм закінчує роботу, якщо в околиці отриманого розміщення відсутні варіанти з меншими значеннями цільової функції.

Розрізняють 2 групи ітераційних алгоритмів:

- алгоритми парних перестановок;
- алгоритми групових перестановок.

2.5 Алгоритм парних перестановок

Послідовне цілеспрямоване покращення довільного початкового розміщення модулів на платі за вибраним критерієм шляхом парних

перестановок. З цією метою на кожній ітерації алгоритм здійснює обчислення сумарної довжини всіх зв'язків для всіляких парних перестановок модулів. З усієї множини перестановок, що дають негативне збільшення, вибирається підмножина, яка задовольняє наступним вимогам:

- вибрана підмножина перестановок дозволяє max зменшити сумарну довжину зв'язків;

- підмножина утворює лише незалежні перестановки в яких модулі незв'язані з модулями інших пар, що переставляються.

Далі здійснюється перестановка виділених таким чином пар модулів і перехід до наступної ітерації.

Описаний ітераційний процес є локальнооптимальним розміщенням модулів на платі. Виведемо формули для сумарної довжини L всіх зв'язків та її збільшення ΔL при переміщенні модулів.

Нехай є плоске або об'ємне плато з вузлами призначеними для установки модулів, задана матриця відстаней $D = |d_{ij}||n * n|$ де d_{ij} визначається звичайною або ортогональною метриками.

Дано деякі сукупності модулів, що підлягають розміщенню, і матриця $C = |c_{ij}||n * m|$ числа зв'язків цих модулів між собою.

Нехай на деякій ітерації є таке розміщення модулів на платі:

$$\begin{matrix} \dots 1\ 2\ 3 \dots e \dots n \\ \text{мод}\ t1\ t2\ t3 \dots t_e \dots t_n \end{matrix} \qquad (2.4)$$

t_e – номер модуля розміщеного у вузлі плати.

Поставимо у відповідність до цього варіанту розміщення матрицю зв'язків $R = |r_{ij}||n * n|$. Елемент r_{ij} дорівнює числу зв'язків між модулями t_i та t_j знаходиться на даній ітерації у вузлах і та j відповідно.

Оскільки відстань між вузлами і і k дорівнює d_{ik} то сумарна довжина зв'язків між модулями t_i і t_k:

$$l_{ik} = r_{ik} * d_{ik} \tag{2.5}$$

Звідси сумарна довжина зв'язків ti модуля розташованого в і-му вузлі пов'язані з усіма модульними схемами дорівнює:

$$L = \frac{1}{2}\sum_{i=1}^{n}\sum_{k=1}^{n}(r_{ik} * d_{ik}) \tag{2.6}$$

Примітка: нехай $i = k = \overline{1,3}$, тоді

$$L = r_{11} * d_{11} + r_{12} * d_{12} + r_{13} * d_{13} + r_{21} * d_{21} + r_{22} * d_{22} + r_{23} * d_{23} + r_{32} * d_{32} + r_{33} * d_{33} \tag{2.7}$$

Знайдемо формулу для збільшення сумарної довжини всіх зв'язків при перестановці мудулів t_i і t_j розташованих у вузлах і і j відповідно.

Для сумарної довжини зв'язків t_i і t_j з усіма іншими модулями маємо:

$$L_{ij} = \sum_{k=1}^{n}(r_{ik} * d_{ik}) + \sum_{k=1}^{n}(r_{ik} * d_{ik}) - 2 * r_{ij} * d_{ij} \tag{2.8}$$

Нехай $n = 4, i = 2, j = 3$, тоді

$$L_{23} = r_{21} * d_{21} + r_{22} * d_{22} + r_{23} * d_{23} + r_{24} * d_{24} + r_{31} * d_{31} + r_{32} * d_{32} + r_{33} * d_{33} + r_{34} * d_{34} - 2 * r_{23} * d_{23} \tag{2.9}$$

Наступним кроком знаходиться формулу для збільшення сумарної

довжини всіх зв'язків при перестановці модулів t_i і t_j розташованих у вузлах і і j відповідно. Для сумарної довжини зв'язків t_i і t_j модулів з усіма модулями, що залишилися, буде отримано:

$$L_{ij} = \sum_{k=1}^{n}(r_{ik} * d_{ik}) + \sum_{k=1}^{n}(r_{ik} * d_{ik}) \qquad (2.10)$$

Можна зауважити, що за n-число перестановок модулів t_i і t_j відповідає перестановці рядків і стовпців з номером і і j в матриці R. Віднімаючи з виразу 2.8 вираз 2.7 визначається збільшення сумарної довжини всіх зв'язків після перестановки модулів з номером t_i і t_j.

$$\Delta L_{ij} = 2 * r_{ij} * d_{ij} - \sum_{k=i}^{n}(r_{ik}r_{jk})(d_{ik}d_{jk}) \;\; i,j = \overline{1,n} \qquad (2.11)$$

Далі вводиться на розгляд матриця P=RxD:

$$p_{ij} = \sum_{k=1}^{n} r_{ik} \times d_{kj} \qquad (2.12)$$

Напівсума діагональних елементів матриці P (напівслід) дорівнює сумарній довжині всіх зв'язків, що визначається формулою 2.6. За допомогою елементів матриці P можуть легко обчислені елементи ΔL_{ij} матриці прирощень для всіх парних перестановок. З урахуванням симетричності матриці D вираз 2.8 перетворюється на вигляд:

$$\begin{gathered} VL_{ij} = 2ri_j d_{ij} + q_{ij}, \qquad i,j = \overline{1,n} \\ q_{ij} = (p_{ij} - p_{ii}) + (p_{ji} - p_{jj}) = \gamma_{ij} + \gamma_{ij}^{*} \end{gathered} \qquad (2.13)$$

Обчислюючи за виразами з 2.10 елементи матриці прирощень ΔL можна вибрати підмножину перестановок, що задовольняють перерахованим вище вимогам.

3 ІМПЛЕМЕНТАЦІЯ АЛГОРИТМУ РОЗМІЩЕННЯ

Після розгляду існуючих алгоритмів для імплементації у даному проекті було обрано метод паралельно-послідовного розміщення, а саме метод зворотного розміщення.

Метод зворотного розміщення був обраний саме через його прив’язаність до посадкових місць, яка єдина підходить для імплементації у додатках, що маштабуються. [5]

3.1 Адаптація та імплементація алгоритму

Після розгляду існуючих алгоритмів та вибору того алгоритму, що найбільш підходить для використання у системі та фреймворці (платформі), необхідно адаптувати алгоритм для використання у додатку та імплементувати.

Далі наведена діаграма алгоритму розміщення (рис. 3.1):

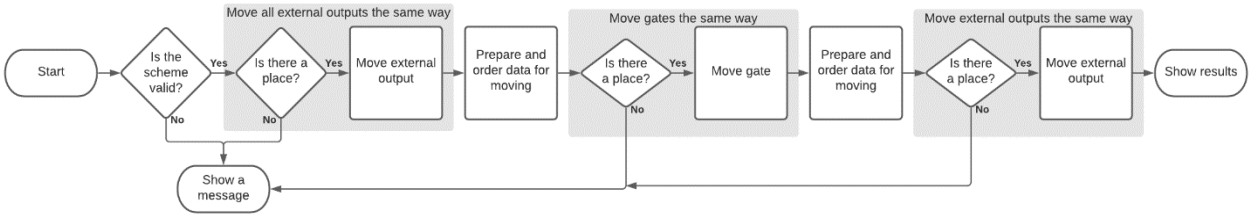


Рисунок 3.1 – Алгоритм розміщення

Механізм розміщення викликається натисканням на кнопку Align:

```
private async void AlignBt_Click(object sender, RoutedEventArgs e)
{
    Aligner liner = new(Scheme);
```

```
    try
    {
        liner.Run();
    }
    catch (DisplayableException exception)
    {
        this.Log(exception.Message);
        await new Message(exception).ShowAsync();
    }

    Scheme.ClearTracings();
}
```

Приклад 3.1 – Код обробки кнопки

У даному прикладі створюється інстанс класу Aligner, у блоці try викликається метод Run(). У наступному catch блоці оброблюється можливе виключення DisplayableException.

DisplayableException – це окреме виключення, що було створене для відображення виключень у класі Message.

Побудова повідомлення з використанням DisplayableException:

```
public Message(DisplayableException ex)
{
    InitializeComponent();

    Title = ex.Message;
    PrimaryButtonText = "OK";
    SecondaryButtonText = "";

    textBlock.Text = ex.Description;
}
```

Приклад 3.2 – Код побудови повідомлення

Якщо виключення DisplayableException виникає, програма пише

повідомлення у лог і виводить повідомлення про помилку на екран.

3.2 Клас Aligner

За алгоритм розміщення, насамперед, відповідає клас Aligner. Виконання алгоритму починається у методі Aligner.Run():

```
public void Run()
{
    this.Log("Running");

    this.Log("Validation...");
    Validate();
    this.Log("OK");

    this.Log("Moving external outputs...");
    processed = outputsAligner.MoveExternalOutputs(processed);

    this.Log("Moving gates...");
    gatesAligner.MoveGates();

    this.Log("Moving external inputs...");
    processed = inputsAligner.MoveExternalInputs(processed);

    this.Log("Done");
}
```

Приклад 3.3 – Код методу Run()

Даний метод починається и закіньчується логуванням за допомогою спецілального Log() методу, що був розглянутий раніше.

3.3 Початкова валідація

Далі у класі Aligner призводиться початкова валідація схеми за

допомогою методу Validate():

```
private void Validate()
{
    if (scheme.ExternalOutputs.Count() > SchemeView.GridSize.Height)
        throw new TooManyExternalOutputsException();

    if(!scheme.ExternalOutputs.Any())
        throw new NoExternalOutputsException();

    if (scheme.ExternalInputs.Count() > SchemeView.GridSize.Height)
        throw new TooManyExternalInputsException();

    if(!scheme.ExternalInputs.Any())
        throw new NoExternalInputsException();

    if (!AllGatesPortsHaveConnectedWires(scheme))
        throw new SomeGateHasInvalidConnection();
}
```

Приклад 3.4 – Валідація схеми

Перш за все, перевіряється, чи наявні у схемі зовнішні вхідні порти. Якщо таких немає, то схема не може бути вирівняна, бо валідна схема має мати хоча б один вхідний порт.

Наступним кроком перевіряється, чи кількість зовнішніх вхідних елементів не перевищує допустиму кількіть. Допустима кількість вхідних портів у схемі дорівнює кількості рядків ґріда, тобто кількості посадкових місць під зовнішні вхідні порти у одній колонці.

Дане обмеження обумовлено осовливістю роботи розробленого алгоритму розміщення – усі зовнішні вхідні порти завжди переміщуються на посадкові місця першої колонки.

Якщо схема перевірку не проходить, то викидується виключення NoExternalInputsException, або, у другому випадку – TooManyExternalInputsException, яке оброблюється програмою у раніше

розглянутому catch блоці, і програма виводить текст виключення на екран у вигляді модального вікна.

Наступними об’єктами перевірки стаєть зовнішні порти іншого типу – вихідного.

Спочатку перевіряється, чи наявні у схемі зовнішні вихідні порти. Якщо таких немає, то схема не може бути вирівняна, бо валідна схема має мати хоча б один вихідний порт.

Наступним кроком перевіряється, чи кількість зовнішніх вихідних елементів не перевищує допустиму кількіть. Допустима кількість вихідних портів у схемі дорівнює кількості рядків ґріда, тобто кількості посадкових місць під зовнішні вихідні порти у одній колонці.

Дане обмеження обумовлено осовливістю роботи розробленого алгоритму розміщення – усі зовнішні вихідні порти завжди переміщуються на посадкові місця останньої колонки.

Якщо схема перевірку не проходить, то викидується виключення NoExternalOutputsException, або, у другому випадку – TooManyExternalOutputsException, яке оброблюється програмою у раніше розглянутому catch блоці, і програма виводить текст виключення на екран у вигляді модального вікна.

Останньою та найскладнішою є перевірка у методі AllGatesPortsHaveConnectedWires(). Назва методу каже сама за себе – це перевірка, чи усі порти логічніх елементів мають підключені до них дроти.

```
private bool AllGatesPortsHaveConnectedWires()
{
    var gates = scheme.Gates;
    return gates.All(g =>
    {
        var connectedInWires = NavigationHelper.ConnectedInputWires(scheme,
g).Count();
        var connectedOutWires = NavigationHelper.ConnectedOutputWires(scheme,
g).Count();
```

```
            var inputsCount = g.Inputs.Count();
            var outputsCount = g.Outputs.Count();

            return connectedInWires == inputsCount && connectedOutWires >=
outputsCount;
        });
}
```

Приклад 3.5 – Метод AllGatesPortsHaveConnectedWires()

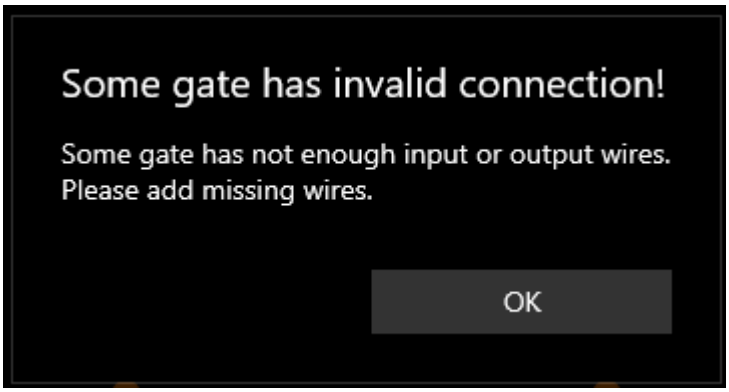
Спочатку отримуються логічні елементи схеми. Далі для кожного елементу виконується наступні кроки:

- отримуються і підраховуються дроти, що підключені до вхідних портів логічного елемента;

- отримуються і підраховуються дроти, що підключені до вихідних портів логічного елемента;

- перевірюється, чи кількість вхідних дротів дорівнює кількості портів та чи кількість вихідних дротів більша, або дорівнює кількості портів логічного елемента.

Якщо схема проходить перевірку, то виконання продовжується. Інакше – программа виводить повідомлення з виключенням SomeGateHasInvalidConnection на екран (рис. 3.2).



Приклад 3.2 – Виключення SomeGateHasInvalidConnection

3.3 Розміщення зовнішніх вихідних портів

Розміщення зовнішніх вихідних портів призводиться за допомогою методу MoveExternalOutputs() класу ExternalOutputsAligner.

```
public HashSet<ISchemeComponent>
MoveExternalOutputs(HashSet<ISchemeComponent> processed)
{
    this.Log("Running...");

    // get external outputs of the scheme.
    var ports = scheme.ExternalOutputs.ToList();

    foreach (var item in ports)
    {
        var place =
NavigationHelper.GetNotOccupiedLocationOnColumn(processed,
(int)SchemeView.GridSize.Width - 1);
        MoveExternalOutput(item, place);
        processed.Add(item);
    }

    this.Log($"Processed {ports.Count()} external outputs");
    this.Log("Done");
    return processed;
}
```

Приклад 3.6 – Метод MoveExternalOutputs()

Даний метод починається и закіньчується логуванням за допомогою спеціального Log() методу, що був розглянутий раніше.

Далі отримуються зовнішні вихідні порти схеми.

Виконання програми продовжується у циклі foreach. Спочатку у зазначеному циклі за допомогою класу NavigationHelper отримується перше вільне посадкове місце схеми. Саме цей метод і потребує перевірки кількості вихідних портів схеми, що була проведена у функції Validate().

Далі, за допомогою методу MoveExternalOutput() порт переміщується у зазначене вільне місце схеми і додається у спеціальну колекцію оброблених компонентів схеми.

```
private void MoveExternalOutput(ExternalPortView port, Vector2 newLocation)
{
    // get connected wires to the port
    var connectedWires = NavigationHelper.ConnectedWires(scheme,
port).ToList();

    // update location
    port.MatrixLocation = newLocation;

    // adjust connected wires location
    foreach (var w in connectedWires)
    {
        var c2 = w.Connection;

        c2.MatrixEnd = newLocation;
        c2.EndPoint = port.GetCenterRelativeTo(scheme);

        w.SetConnection(c2);
    }
}
```

Приклад 3.7 – Метод MoveExternalOutput()

Через те, що компоненти схеми та дроти зв'язані не напряму, а через індекси у сітці, у наведеному методі спочатку за допомогою класу NavigationHelper отримуються підключені до порту дроти, доки порт має старі координати.

Далі даний зовнішний порт отримує нові координати MatrixLocation у сітці.

Останнім кроком у циклі foreach призводиться оновлення координат дротів – отримується стара коннекшн структура, переписується її MatrixEnd координати сітки і її фізична репрезентація EndPoint. EndPoint отримується

використовуючи розширюючу функцію GetCenterRelativeTo(). Оновлена структура передається у дріт за допомогою SetConnection().

3.4 Розміщення логічних елементів

Розміщення логічних елементів призводиться за допомогою методу MoveGates() класу GatesAligner.

```
public void MoveGates()
{
    this.Log("Running");
    if(!scheme.Gates.Any())
        return;

    var gates = scheme.Gates.ToList();
    var gatesWithRanges = gates.Select(x => (gate: x, range: new
RangeHelper(scheme).GetRange(x))).ToList();
    var gatesToProcess = new
Queue<GateView>(gatesWithRanges.OrderByDescending(x => x.range).Select(g =>
g.gate));
```

Приклад 3.8 – Метод MoveGates()

Виконання цього методу починається з перевірки, чи має схема хоча б один логічний елемент, адже схема з зовнішніми портами, але без логічних елементів все ще є валідною.

Далі отримується список логічних елементів схеми, отриманий список оброблюється і сторюється нова колекція, кожен елемент що містить логічний елемент і його рендж.

Рендж логічного елемента – це його номер відносно зовнішніх входів, до яких він під’єднаний.

```
public int GetRange(GateView gate)
{
```

```
    // initial range for gates
    int range = 1;

    var gates = GetSourceGates(gate).ToList();;
    while (gates.Any())
    {
        gates = gates.SelectMany(GetSourceGates).ToList();
        range++;
    }

    return range;
}
```

Приклад 3.9 Метод GetRange()

Він обчислюється у методі GetRange() наступним чином:

- створюється змінна range з початковим значенням дорівнюючим 1. Початкове значення потрібне для того, щоб виділити першу колонку зовнішнім вхідним портам;

- ініціалізується перелік з логічних елементів, що є джерелами сигналів для даного логічного елементу, тобто з яких даний логічний елемент отримує сигнали;

- у циклі на кожному кроці перевіряється, чи містить колекція хоча б один елемент, а далі перелік оновлюється з використанням тієї ж самої функції і збільшується range лічильник.

Особливість роботи даного алгоритму у тому, що на кожному кроці він отримує перелік логічних елементів, підключених до тих, що вже були знайдені на попередньому кроці. Таким чином здійснюється покроковий обхід дерева. У алгоритмі ігноруються всі вихідні порти, бо вони не збільшують рендж.

Далі отриманий перелік сортується і з нього формується черга, у якій містяться відсортовані за ренджем логічні елементи.

Отримана черга у подальшому передається у головний цикл методу для

подальшої обробки:

```
while(gatesToProcess.Any())
{
    var gate = gatesToProcess.Dequeue();

    // will be used to set column
    int range = gatesWithRanges.Find(x => x.gate == gate).range;

    var outputWires = NavigationHelper.ConnectedOutputWires(scheme, gate);
    var destinations = outputWires.Select(w =>
NavigationHelper.GetDestination(scheme, w));
    var locations = destinations.Select(x => x.MatrixLocation);

    int minRow = (int)locations.Min(l => l.Y);

    if (minRow >= SchemeView.GridSize.Height)
        throw new TooManyGatesWithSameRangeException();

    MoveGate(gate, range, (int)minRow);
}
```

Приклад 3.9 – Головний цикл методу MoveGates()

У головному циклі спочатку з черги дістається логічний елемент, далі отриманий елемент знаходиться в іншому переліку, що був розгянутий раніше. І у отриманого елементу отримається раніше обчислений рендж.

Далі за допомогою класу NavigationHelper та його методу ConnectedOutputWires(), що був розглянутий раніше, отримається колекція з дротів, що підключені до логічного елементу, що оброблюється.

Отримані дроти використовуються для отримання destination елементів за допомогою методу GetDestination() класу NavigationHelper, з даних елементів отримуються індекси сітки.

З отриманих індексів сітки обчислюється найменше значення рядка.

Якщо обчислене значення більше, ніж кількість рядків – створюється і

викидується виключення TooManyGatesWithSameRangeException, що у подальшому оброблюється у catch блоці і виводиться у модальному діалоговому вікні.

Дане виключення потрібно для того, щоб усі створені логічні елементи з однаковим ренджем правильно відображались.

Останнім кроком обробки стає виконання методу MoveGate():

```
private void MoveGate(GateView g, int range, int index)
{
    var oldLocation = g.MatrixLocation;

    var inputWires = NavigationHelper.ConnectedInputWires(scheme,
g).ToList();
    this.Log($"inputWires = {inputWires.Count()}");
    var outputWires = NavigationHelper.ConnectedOutputWires(scheme,
g).ToList();
    this.Log($"outputWires = {outputWires.Count()}");

    var newLocation = new Vector2(range, index);
    g.MatrixLocation = newLocation;

    this.Log($"Moving gate from [{oldLocation.X}, {oldLocation.Y}] to
[{newLocation.X}, {newLocation.Y}]");
```

Приклад 3.10 – Метод MoveGate()

У наведеному методі спочатку отримається індекс логічного елементу у сітці.

Далі отриманий індекс за допомогою класу NavigationHelper і його методу ConnectedInputWires(), що був розглянутий раніше, отримує дроти, що підключені до вхідних портів логічного елемента, що оброблюється у циклі. Кількість отриманих дротів логується.

Також отриманий індекс за допомогою класу NavigationHelper і його методу ConnectedOutputWires(), що також був розглянутий раніше, отримує дроти, що підключені до вихідних портів логічного елемента, що

оброблюється у циклі. Кількість отриманих дротів теж логується.

Останнім кроком створюється нова структура для визначення нового місця розташування логічного елементу. Створена структура логується і передається у логічний елемент, що обробляється.

Далі виконання переходить у наступний цикл:

```
// adjust connected wires location
foreach (var w in inputWires)
{
    var c2 = w.Connection;

    c2.MatrixEnd = newLocation;

    if (c2.EndPort != null)
    {
        var port = g.Inputs.ToList()[c2.EndPort.Value];
        c2.EndPoint = port.GetCenterRelativeTo(scheme);
    }
    w.SetConnection(c2);
}
```

Приклад 3.11 – Цикл оновлення вхідних дротів

У наведеному циклі foreach спочатку отримується структура підключення дроту, вона отримує значення про вихідну конфігурацію з попередньо створеної нової структури newLocation.

Далі, якщо дріт був підключений до вхідних портів логічного елементу, тобто номер вхідного порту не пустий – порт обробляємого елементу дістається по номеру та координати у сітці оновлюються.

Останнім кроком даного циклу дріт отримує оновлену конекшн структуру.

Далі виконання переходить у другий цикл:

```
// adjust connected wires location
```

```
foreach (var w in outputWires)
{
    var c2 = w.Connection;

    c2.MatrixStart = newLocation;

    if (c2.StartPort != null)
    {
        // todo investigate why it's null
        var port = g.Outputs.ToList()[c2.StartPort.Value];
        c2.StartPoint = port.GetCenterRelativeTo(scheme);
    }

    w.SetConnection(c2);
}
```

Приклад 3.12 – Цикл оновлення вихідних дротів

У наведеному циклі foreach спочатку отримується структура підключення дроту, вона отримує значення про вихідну конфігурацію з попередньо створеної нової структури newLocation.

Далі, якщо дріт був підключений до вихідних портів логічного елементу, тобто номер вихідного порту не пустий – порт обробляємого елементу дістається по номеру та координати у сітці оновлюються.

Останнім кроком даного циклу дріт отримує оновлену конекшн структуру.

3.5 Розміщення зовнішніх вхідних портів

Розміщення зовнішніх вхідних портів призводиться за допомогою методу MoveExternalInputs() класу ExternalInputsAligner.

```
public HashSet<ISchemeComponent> MoveExternalInputs(HashSet<ISchemeComponent>
processed)
{
```

```
    this.Log("Running...");

    var externalInputs = scheme.ExternalInputs.ToList();
    var portsWithIndexes = new List<(ExternalPortView port, int destLoc, int
portIdx)>();

    // ...

    this.Log($"Processed {externalInputs.Count} external inputs.");
    this.Log("Done");
    return processed;
}
```

Приклад 3.13 – Метод MoveExternalInputs()

Даний метод починається и закіньчується логуванням за допомогою спеціального Log() методу, що був розглянутий раніше.

Далі отримуються зовнішні вхідні порти схеми.

Виконання програми продовжується у циклі foreach. Спочатку у зазначеному циклі за допомогою методу ConnectedWires() класу NavigationHelper отримується перелік підключених дротів.

Далі за допомогою методу GetDestination() класу NavigationHelper отримуються destination компоненти схеми. Далі перелік компонентів сортується за допомогою їх місця розташування у колонках сітки та далі – у її стовпцях. З отриманого переліку обирається перший компонент і його конфігурація розташування.

```
foreach (var i in externalInputs)
{
    var wires = NavigationHelper.ConnectedWires(scheme, i).ToList();
    var destinations = wires.Select(x =>
NavigationHelper.GetDestination(scheme, x));
    var minLocDest = destinations.OrderBy(x => x.MatrixLocation.X).ThenBy(x
=> x.MatrixLocation.Y).First();
    var minLoc = minLocDest.MatrixLocation;
```

Приклад 3.14 – Початок першого циклу методу MoveExternalInputs()

У наступному фрагменті коду обчислюється мінімальний індекс вихідного порту у логічному елементі.

Він обчислюється наступним чином:

- створюється змінна зі значенням нуль за замовчуванням;

- якщо приймаючий компонент схеми – логічний елемент, тоді отримуються всі дроти, що до нього під’єднані;

- у отриманій колекції під’єднаних вхідних дротів обчислюється мінімальний індекс порту у логічному елементі.

Дане обчислення необхідно, щоб «розплутати» дроти, підключені до одного логічного елементу. Таким чином дроти додатково будуть упорядковані за індексом порту логічного елементу, що до них підключений.

Отримані дані записуються у спеціальну колекцію portsWithIndexes, що зберігає порти з індексами.

```
    int minPortIndex = 0;
    if(minLocDest is GateView gate)
    {
        var connectingWires = wires.Where(w => gate.WireEndConnects(w));
        minPortIndex = connectingWires.Min(x => x.Connection.EndPort.Value);
    }

    portsWithIndexes.Add((i, (int)minLoc.Y, minPortIndex));
}
```

Приклад 3.15 – Кінець першого циклу методу MoveExternalInputs()

Далі створюється колекція portsToPlace.

Дана колекція створюється з portsWithIndexes, спочатку сортуючи її за допомогою місця розташування логічного елементу, а точніше його рядка у сітці, а далі – за допомогою індексу порту у логічному елементі, під’єднаному

до порту.

Таким чином, з колекції portsWithIndexes залишаються лише відсортовані у правильному порядку зовнішні порти.

```
var portsToPlace = portsWithIndexes.OrderBy(x => x.destLoc).ThenBy(x =>
x.portIdx).Select(x => x.port);

foreach (var port in portsToPlace)
{
    var place = NavigationHelper.GetNotOccupiedLocationOnColumn(processed,
0);
    MoveExternalInput(port, place);
    processed.Add(port);
}
```

Приклад 3.16 – Другий цикл методу MoveExternalInputs()

Виконання програми продовжується у циклі foreach. Спочатку у зазначеному циклі за допомогою класу NavigationHelper отримується перше вільне посадкове місце схеми. Саме цей метод і потребує перевірки кількості вхідних портів схеми, що була проведена у функції Validate().

Далі, за допомогою методу MoveExternalntput() порт переміщується у зазначене вільне місце схеми і додається у спеціальну колекцію оброблених компоненті

```
private void MoveExternalInput(ExternalPortView port, Vector2 newPosition)
{
    // get connected wires to the port by its old location
    var connectedWires = NavigationHelper.ConnectedWires(scheme,
port).ToList();

    // update location
    port.MatrixLocation = newPosition;

    // adjust connected wires location
    foreach (var w in connectedWires)
```

```
    {
        var c2 = w.Connection;

        c2.MatrixStart = newPosition;
        c2.StartPoint = port.GetCenterRelativeTo(scheme);

        w.SetConnection(c2);
    }
}
```

Приклад 3.17 – Метод MoveExternalOutput()

Через те, що компоненти схеми та дроти зв'язані не напряму, а через індекси у сітці, у наведеному методі спочатку за допомогою класу NavigationHelper отримуються підключені до порту дроти, доки порт має старі координати.

Далі даний зовнішний порт отримує нові координати MatrixLocation у сітці.

Останнім кроком у циклі foreach призводиться оновлення координат дротів – отримується стара коннекшн структура, переписується її MatrixStart координати сітки і її фізична репрезентація StartPoint. StartPoint отримується використовуючи розширюючу функцію GetCenterRelativeTo(). Оновлена структура передається у дріт за допомогою SetConnection().

4 РОЗРОБКА АРХІТЕКТУРИ ПРОЕКТУ

4.1 Критерії гарної архітектури

Взагалі кажучи, немає загальноприйнятого терміна «архітектура програмного забезпечення». Проте коли справа стосується практики, то для більшості розробників і так зрозуміло який код є хорошим, а який поганим.

Хороша архітектура це насамперед вигідна архітектура, що робить процес розробки та супроводу програми більш простим та ефективним. Програму з гарною архітектурою легше розширювати та змінювати, а також тестувати, налагоджувати та розуміти. Тобто насправді можна сформулювати список цілком розумних та універсальних критеріїв.

4.2 Ефективність системи

Насамперед програма, звичайно ж, повинна вирішувати поставлені завдання та добре виконувати свої функції, причому у різних умовах.

У даний пункт можна віднести такі характеристики, як надійність, безпека, продуктивність, здатність справлятися зі збільшенням навантаження, тобто масштабованість тощо.

4.3 Гнучкість системи

Будь-який додаток доводиться змінювати з часом – змінюються вимоги, додаються нові. Чим швидше і зручніше можна внести зміни до існуючого функціоналу, чим менше проблем і помилок це викличе – тим гнучкіша і конкурентоспроможніша система.

Тому в процесі розробки слід намагатися оцінювати те, що виходить, щодо того, як це потім, можливо, доведеться змінювати. Слід поставити

запитання: «А що буде, якщо поточне архітектурне рішення виявиться неправильним?», «Яка кількість коду зазнає при цьому змін?».

Зміна одного фрагмента системи має впливати на її інші фрагменти. По можливості, архітектурні рішення не повинні «вирубуватись у камені», і наслідки архітектурних помилок мають бути в розумній мірі обмежені.

4.4 Розширюваність системи

Можливість додавати до системи нові сутності та функції, не порушуючи її основної структури

На початковому етапі в систему має сенс закладати лише основний та найнеобхідніший функціонал, що описаний у принципі YAGNI – you ain't gonna need it, «Вам це не знадобиться». Але при цьому архітектура повинна дозволяти легко нарощувати додатковий функціонал при потребі. Причому так, щоб внесення найімовірніших змін вимагало найменших зусиль.

Вимога, щоб архітектура системи мала гнучкість і розширюваність, а тобто була здатна до змін та еволюції, є настільки важливою, що вона навіть сформульована у вигляді окремого принципу – «Принципу відкритості/закритості».

Дана вимога є другим з п'яти принципів SOLID і має назву Open-Closed Principle. Сутність її у наступному: програмні сутності (класи, модулі, функції тощо) повинні бути відкритими для розширення, але закритими для модифікації. [6]

Іншими словами: має бути можливість розширити/змінити поведінку системи без зміни/переписування вже існуючих частин системи.

Це означає, що додаток слід проектувати так, щоб зміна його поведінки і додавання нової функціональності досягалась за рахунок написання нового коду (розширення), і при цьому не доводилося б змінювати вже існуючий код.

У такому разі поява нових вимог не спричинить модифікацію існуючої логіки, а зможе бути реалізована насамперед за рахунок її розширення. Саме

цей принцип є основою «плагінної архітектури» (Plugin Architecture). Про те, за рахунок яких технік це може бути досягнуто, буде наведено далі. [7]

4.5 Масштабованість процесу розробки

Можливість скоротити термін розробки рахунок додавання до проекту нових людей. Архітектура повинна дозволяти розпаралелити процес розробки, так щоб багато людей могли працювати над програмою одночасно.

4.6 Тестованість

Код, який легше тестувати, міститиме менше помилок та надійніше працюватиме. Але тести не лише покращують якість коду. Багато розробників приходять до висновку, що вимога «хорошої тестованості» є також спрямовуючою силою, що автоматично веде до гарного дизайну, і одночасно одним з найважливіших критеріїв, що дозволяють оцінити його якість.

Існує ціла методологія розробки програм на основі тестів, яка так і називається – «Розробка через тестування», або «Test-Driven Development, TDD».

4.7 Можливість повторного використання

Систему бажано проектувати так, щоб її фрагменти можна було використовувати повторно в інших системах.

4.8 Супроводжуваність

Простіше кажучи – добре структурований, читаний та зрозумілий код.

Над програмою, як правило, працює безліч людей – одні йдуть, приходять нові. Після написання супроводжувати програму теж, як правило,

доводиться людям, які не брали участь у її розробці.

Тому хороша архітектура має давати можливість відносно легко та швидко розібратися у системі новим людям.

Проект має бути добре структурований, не містити дублювання, мати добре оформлений код та бажано документацію.

І, по можливості, в системі краще застосовувати стандартні, загальноприйняті звичні рішення для програмістів. «Чим екзотичніша система, тим складніше її зрозуміти іншим», – як згадується у «Принципі найменшого подиву», або «Principle of least astonishment». Зазвичай, він використовується щодо інтерфейсу користувача, але застосовний і до написання коду).

5 РОЗРОБКА PRESENTATION LAYER

Для реалізації даного рівня необхідно вирішити наступні питання: відрисування та масштабування статичних та динамічних елементів; приорітизація елементів на екрані.

5.1 Функціональність програми

Після запуску на екрані з’являється сітка з посадкових місць на якій необхідно розміщувати вхідні, вихідні порти та логічні елементи.

Для кожної можливої в програмі операції існує окрема кнопка в пункті меню, що знаходиться зверху, над схемою, після натисканні якої, обробник подій виконує одну з наступних дій:

- створення нової схеми;
- збереження схеми;
- відновлення схеми;
- вирівнювання елементів схеми;
- запуск процесу трасування;
- симуляція роботи схеми.

Друга частина операцій проводиться за допомогою взаємодії з елементами гриду:

- додавання нового логічного елементу;
- видалення логічного елементу;
- додавання нового зовнішнього порту;
- зміна значення зовнішнього порту;
- видалення зовнішнього порту;
- створення лінії між елементами;
- видалення лінії між елементами.

5.2 Елементи меню

У даний розділ можна включити головне вікно програми, а також діалогові вікна, що використовуються для збору даних користувача.

Діалогові вікна знаходяться в окремій папці Dialogs. Програма містить наступні діалогові вікна:

5.2.1 Діалог CustomGateDialog

Даний діалог (рис. 5.1) потрібен для створення окремого елементу користувача, а саме для дефінування функціонування.

Рисунок 5.1 – CustomGateDialog

Даний діалог дозволяє сконфігурувати принцип роботи логічного елементу.

Діалог має поле для введення назви нового логічного елементу, ToggleSwitch для встановлення значення виходу за замовчуванням та має ґрід для дефініції виключень.

Якщо значення вхідних виходів дорівнюють значенням, що були задефіновані у ґріді, то значення буде зворотним від значення, що було обране

у полі Default output.

При натисканні кнопки OK створюється сконфігурований елемент, а при натисканні кнопки Cancel створення елементу не призводиться.

5.2.2 Діалог NewExternalPortDialog

Наступний діалог (рис 5.2) використовується для створення зовнішнього порту схеми:

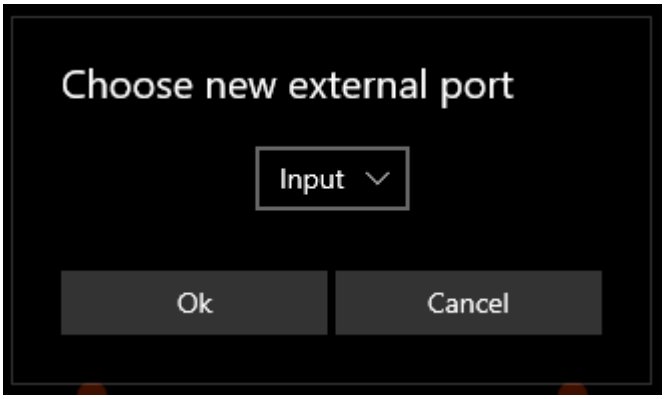


Рисунок 5.2 – NewExternalPortDialog

Лукап даного діалогу містить усього 2 значення: Input та Output. При натисканні кнопки OK створюється зовнішній порт, а при натисканні кнопки Cancel створення елементу не призводиться.

5.2.3 Діалог NewGateDialog

Наступний діалог (рис. 5.3) використовується для створення зовнішнього порту схеми:

Рисунок 5.3 – NewGateDialog

Якщо був обраний логічний елемент з бібліотеки стандартних елементів – при натисканні кнопки OK створюється сконфігурований елемент.

У іншому випадку буде відкрито додаткове вікно CustomGateDialog, яке було розглянуто у попередньому розділі.

При натисканні кнопки Cancel створення елементу не призводиться.

5.2.3 Діалог New Message

Message (рис. 5.4) – це діалог загального призначення. Він може використовуватися для показу будь-яких інформаційних даних, запросів на підтвердження дії користувачем.

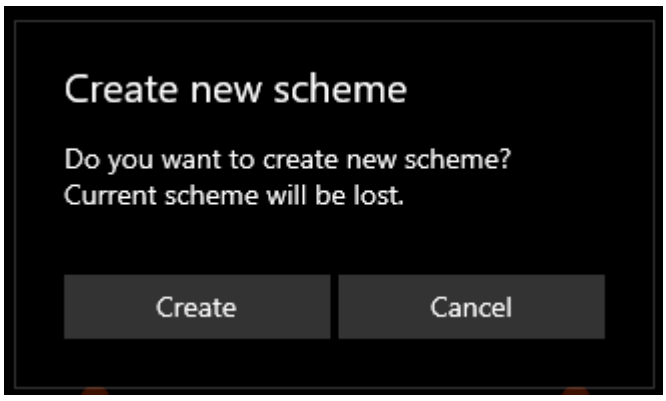


Рисунок 5.4 – Message

5.3 Динамічні елементи

У даний розділ можна включити візуальні елементи, чий вигляд та розташування повністю залежить від користувача, тобто вигляд, що не є детермінований розробником програми.

До таких елементів належать:

- Логічні елементи та їх частини: вхідні та вихідні порти, тіло елементу (рис. 5.5);

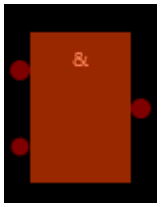


Рисунок 5.5 – Логічний елемент

- Зовнішні порти схеми (рис. 5.6):

Рисунок 5.6 – Зовнішні порти схеми

- Лінії з’єднання (рис. 5.7):

Рисунок 5.7 – Лінія з’єднання

- Посадкові місця для розташування елементів (рис. 5.8):

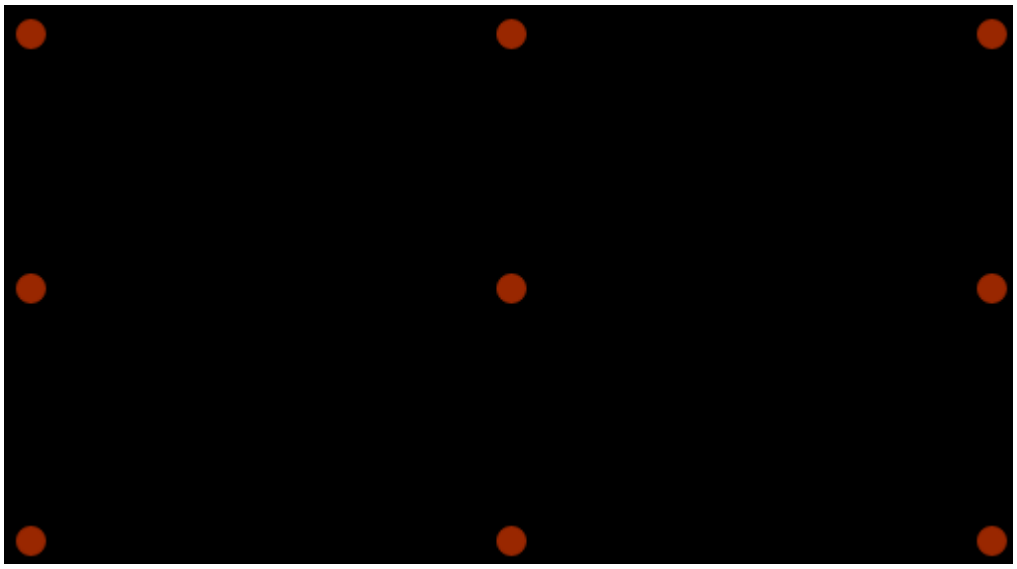


Рисунок 5.8 – Посадкові місця схеми

Також у даний розділ можна додати і службові компоненти інтерфейсу користувача для функціонування програми:

- Лейбли для позначення номеру лінії (рис. 5.9):

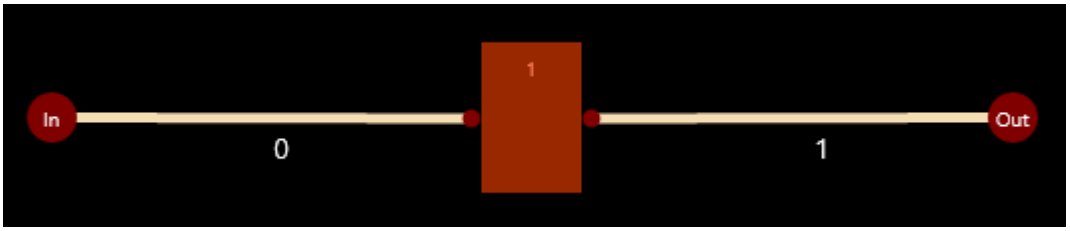


Рисунок 5.9 – Лінії з лейблами

- Елемент PortConfigurationView (рис. 5.10) для дефінування правил роботи створеного користувачем логічного елементу типу Custom:

Рисунок 5.10 – PortConfigurationView

5.4 Контейнери

Програма має спеціальні контейнери для зберігання логічних елементів. Кожен такий контейнер має постфікс Layer і реалізує інтерфейс ILayer.

```
public interface ILayer<T> where T : UserControl
{
    public IEnumerable<T> Items { get; }

    public void Add(T item);
    public void Clear();
}
```

Приклад 5.1 – Код інтерфейсу ILayer.cs

Властивість Items потрібна, щоб мати доступ до об’єктів ґріду. Метод Add() додає об’єкт до колекції елементів. Метод Clear() видаляє всі об’єкти з колекції.

Даний інтерфейс реалізують такі класи, як: PortLayer, ExceptionsLayer, ExternalPortsLayer, GateLayer, GatePortsLayer, PortConfigurationLayer, TraceLayer та WireLayer.

Через те, що UWP не дає можливості розширювати функціонал контролів за допомогою наслідування, для спрощення реалізації даних методів у всіх Layer контролів, було розроблено наступні розширюючі методи:

- Методи для спрощення роботи з властивістю Items:

```
public static IEnumerable<T> GetItems<T>(this Grid grid)
    where T : UIElement
{
    return grid.Children.Select(c => c as T);
}

public static void SetItems<T>(this Grid grid, IEnumerable<T> items)
```

```
    where T : UIElement
{
    grid.Children.Clear();
    items.ToList().ForEach(x => grid.Add(x));
}
```

Приклад 5.2 – Розширюючі методи GetItems() та SetItems()

```
public IEnumerable<PortConfigurationView> Items =>
Grid.GetItems<PortConfigurationView>();
```

Приклад 5.3 – Використання у Layer класі

- Метод для спрощення роботи з методом Add():

```
public static void Add<T>(this Grid grid, T item)
    where T : UIElement
{
    grid.Children.Add(item);
}
```

Приклад 5.4 – Розширюючий метод Add()

```
public void Add(DotView e) => Grid.Add(e);
```

Приклад 5.5 – Використання у Layer класі

- Метод для спрощення роботи з методом Clear():

```
public static void Clear(this Grid grid)
{
    grid.Children.Clear();
}
```

Приклад 5.6 – Розширюючий метод Clear()

```
public void Add(DotView e) => Grid.Add(e);
```

Приклад 5.7 – Використання у Layer класі

5.5 Масштабування

Перш за все, для вирішення даної проблеми необхідно задати деякі константи, що будуть обмежувати можливі розміри головного вікна. Головне вікно не може занадто малим, бо програма не зможе коректно відобразити елементи. Для даного випадку розмір вікна повинен бути розрахований наступним чином:

```
var MinWindowSize = new Size
{
        Width = Math.Max(Menu.Width, Grid.Width),
        Height = Menu.Height + Grid.Height
};
```

Приклад 5.8 – Приклад розрахунку мінімального розміру вікна

Наступний лістинг розраховує мінімальний розмір головного гріду, що містить схему:

```
var MinGridSize = new Size
{
        Width = Row.Width * Grid.ColumnsCount,
        Height = Column.Height * Grid.RowsCount
};
```

Приклад 5.9 – Приклад розрахунку мінімального розміру гріду

Наступний лістинг розраховує мінімальний розмір стовпця та рядку

гріду, що містить схему. У даному розрахунку використовується розмір логічного елементу – найбільшого з динамічних об’єктів. Крім того, розміри логічного елементу перемножуються на коефіцієнт 2, щоб залишити достатньо вільного місця для ліній.

```
var MinCellSize = new Size
{
        Width = Gate.Width * 2,
        Height = Gate.Height * 2
};
```

Приклад 5.10 – Приклад розрахунку мінімального розміру клітинки

Розмір логічного елементу потрібен бути розрахуваний лише один раз, у зв’язку з тим, що даний елемент містить текст, а його масштабування є нерівномірне і досить важкою проблемою.

```
var gateSize = new Size
{
        Width = GateBody.Width + GatePort.Size.Width * 2,
        Height = GateBody.Height + GatePort.Size.Height * 2
};
```

Приклад 5.11 – Приклад розрахунку мінімального розміру логічного елементу

У свою чергу, всі інші розміри елементів потрібно дефінувати власноруч, за допомогою констант. За допомогою емпіричних обчислень, було обрано наступні розміри елементів:

```
private readonly Size traceLabelSize = new(20, 20);
private readonly Size traceLabelOffset = new(0, 5);

private readonly Size externalPortSize = new(25, 25);
```

```
public static readonly Size LogicGateSize = new(50, 75);

private readonly Size dotSize = new(15, 15);

public static readonly Size GatePortSize = new(10, 10);

private const double WireThickness = 5.0;
```

Приклад 5.12 – Дефініція обраних розмірів елементів

Також є необхідним дефінувати розмір сітки за замовчуванням, був обраний розмір сітки, що дорівнює 8, тобто 64 вільних посадкових місця для елементів.

Якщо ж платформа, з використанням якої виконується розробка програми не дозволює змінення розміру вікна, як у випадку UWP, можна заборонити користувачу змінення режиму відображення вікна.

6 РОЗРОБКА BUSINESS LAYER

6.1 Бібліотека стандартних елементів

Бібліотека стандартних елементів (рис. 6.1) містить наступні типи комбінаційних елементів: Buffer, Not, And, Nand, Or, Nor, Xor, Xnor. [8]

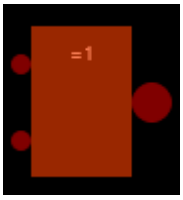


Рисунок 6.1 – Логічний елемент

6.2 Конструктор логічного елементу

Для створення кожного логічного елементу потрібно отримати наступні його параметри від користувача: тип, кількість входів та кількість виходів (рис. 6.2).

Оскільки усі елементи з бібліотеки стандартних елементів мають лише один вихід, налаштування виходу усієї бібліотеки стандартних елементів буде завжди дорівнювати одиниці.

Елементи Buffer і Not мають тільки один вхід, тому кількість входів у них дефінується таким самим чином, як і кількість виходів.

Інші елементи мають більш складні налаштування виходів. Мінімальні і максимальні значення дорівнюють константам, що дефінуються у коді чи у файлі констант.

Теоретично, елементи зі стандартної бібліотеки можуть мати будь-яку кількість входів і все-одно будуть функціонувати коректно. Адже обчислення їх виходного значення призводиться з використанням функції підчас

виконання програми. Таким чином, програма не потребує зберігання додаткових конфігураційних даних.

Але, все ж таки, для відображення існуючих елементів та для коректного відображення вхідних портів потрібно обрати максимальне число. Таким чином, було обране обмежувальне число, що дорівнює 4.

Рисунок 6.2 – Меню створення логічного елемента

6.3 Створення власних елементів

Перш за все, аналогічно зі створенням елементу з шаблонів стандартної бібліотеки, створений користувачем елемент потребує вибору кількості входів та виходів елементів.

Різниця полягає у тому, що користувач повинен обрати інший тип логічного елементу – Custom.

Якщо був обраний даний тип логічного елементу, то програма відобразить додаткове вікно для дефінування правил його роботи.

Перш за все, створення власного елементу потребує дефінування унікальної назви для розрізнення створених елементів.

Крім того, назва не може повторювати вже існуючі назви логічних елементів. Таким чином, користувач не може створити копію існуючого елементу.

Замість того, щоб вибрати тип елементу з бібліотеки за допомогою

випадаючого списку, необхідно, щоб користувач створив нове, унікальне ім’я для створеного елементу.

Ім’я елемента не може повторювати існуючі пресети бібліотеки шаблонів і вже існуючі назви елементів, створених користувачем, бо буде добавлене у той же список і призведе до проблем з ідентифікацією усіх елементів. Таким чином, при введенні назви елементу потрібно одразу ж проводити валідацію.

Додатково назва елементу має містити тільки латинські літери та арабські цифри. Крім того, максимальна довжина має бути задефінована.

У даному випадку вона, наприклад, може дорівнювати 10.

Наприклад, у бібліотеці стандартних елементів існує такий логічний елемент, як NOT. І, згідно логіці програми, користувач не може створити елемент зі співпадаючою назвою у будь-якому регистрі.

```
public bool Validate(string newName)
{
if(gate.Type == null || gate.Type.AreCharactersValid())
        return false;
if(standartGates.Any(gate => gate.Type.ToString().ToLower() ==
newName.ToLower()))
        return false;
else if(userGates.Any(gate => gate.Name.ToLower() == newName. .ToLower()))
        return false;
else return true;
}
```

Приклад 6.1 – Приклад валідації назви

Крім того, є різниця і в визначенні кількості портів. Перш за все, потрібно обрати максимальну кількість портів. Цей вибір є необхідним, адже при обранні занадто великої кількості портів може виникнути проблема з розташуванням портів і складності дефінування значень користувачем.

Тому, аналогічно зі стандартними шаблонами, була обрана цифра 4 для

кількості входів і виходів.

Логічний елемент конфігурується за допомогою окромого меню (рис. 6.3).

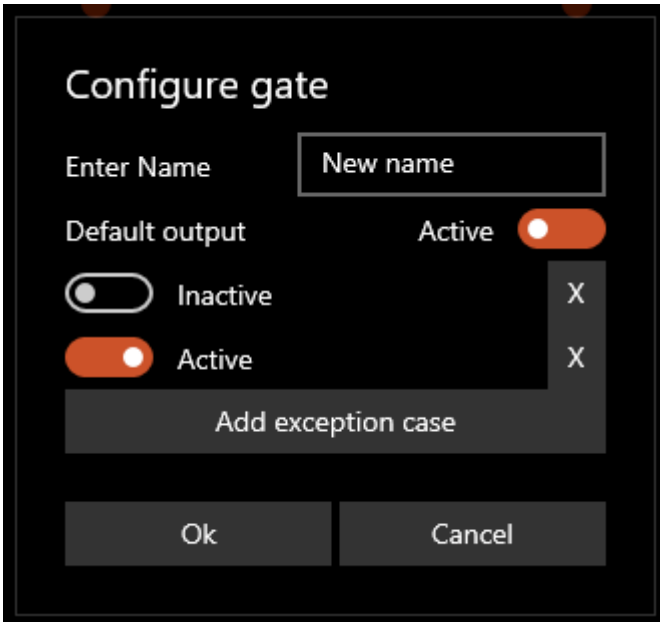


Рисунок 6.3 – Меню конфігурування логічного елемента

6.4 Алгоритм нумерації ліній

Функціональність даної програми включає в себе нумерацію ліній, що поєднують логічні елементи схеми (рис. 6.4).

Перш за все, головним принципом нумерації ліній є те, що чим віддаленіше знаходиться лінія від входу, тим більший номер їй присвоюється, таким чином, був розроблений алгоритм нумерації ліній.

Алгоритм нумерації ліній складається з декількох кроків:

- початкова валідація;
- перш за все нумеруються вхідні логічні елементи,
- далі нумеруються лінії, що ідуть з трасованих елементів схеми,
- далі нумеруються внутрішні лінії, що під’єднуються до трасованих елементів,

- далі кроки 2 та 3 повторюються до тих пір, доки всі елементи та лінії будуть трасовані,

- Останніми трасуються вихідні елементи схеми.

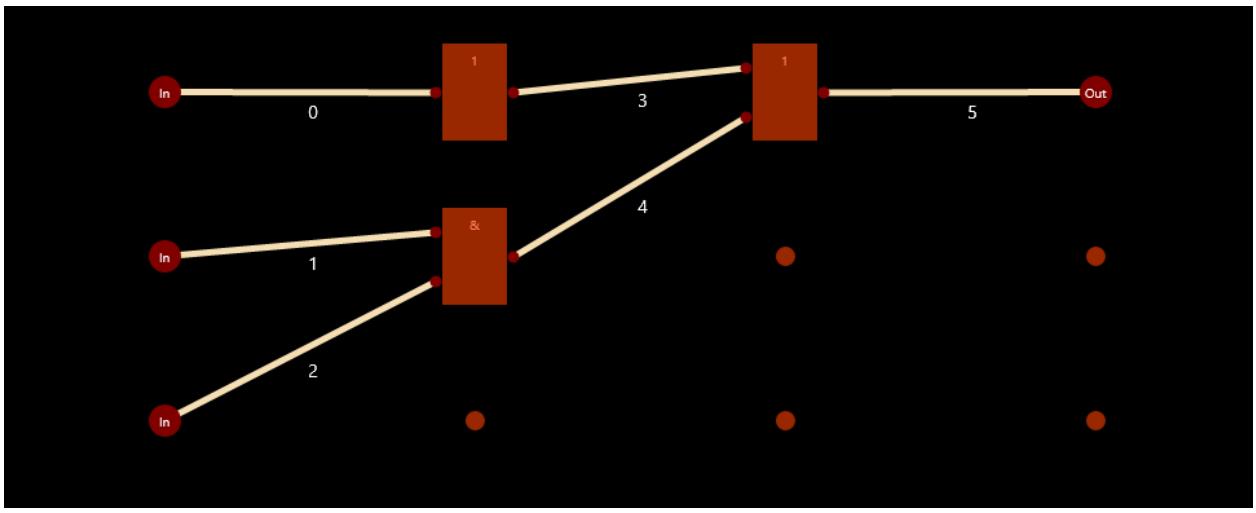


Рисунок 6.4 – Приклад нумерації ліній схеми

6.4.1 Початкова валідація

Далі описаний алгоритм початкової валідації:

```
public HistoryService? Run()
{
    this.Log("Running...");

    (int exPorts, int gates, int wires) total = (externalPorts.Count(),
gates.Count(), wires.Count());

    // check if there is nothing to trace
    if (total.exPorts == 0 || total.wires == 0)
    {
        new Message(Messages.NothingToTrace).ShowAsync();
        return null;
    }
```

Приклад 6.2 – Початкова валідація

Дана частина коду починається з логування початку роботи функції. Наступим кроком створюється кортеж з кількості вихідних портів, логічних елементів та ліній.

Далі проводиться перевірка, чи є у схемі вихідні порти та лінії з’єднання. Логічні елементи не перевіряються, бо мінімальною вимогою для функціонування схеми є те, щоб схема мала вихідні порти та дроти для з’єднання.

Якщо логічна схема не є валідною, то буде показано наступне інформаційне вікно (рис. 6.5):

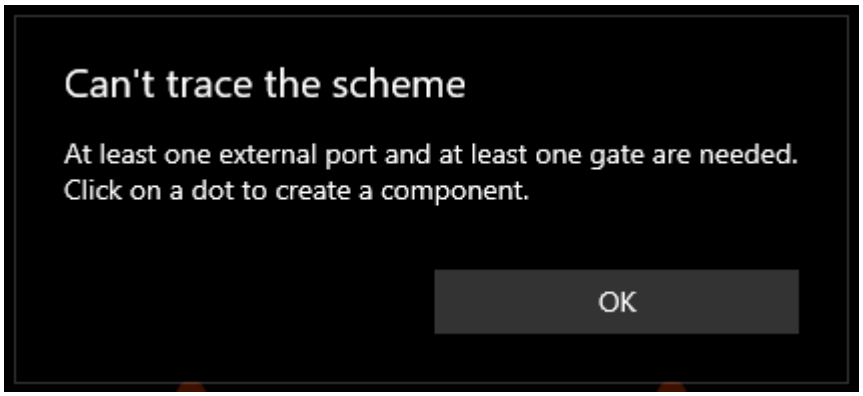


Рисунок 6.5 – Помилка валідації

6.4.2 Нумерація вхідних елементів

У даному розділі міститься код, що проводить нумерацію початкових елементів. Елементи всіх інших типів оброблюються в інших методах, які будуть розглянуті у подальших розділах та прикладах.

Виклик метода трасування у загальному методі:

```
// trace external inputs first
this.Log("Tracing external inputs...");
service.AddToHistory(TraceExternalPorts(PortType.Input));
```

Приклад 6.3 – Виклик метода трасування вхідних портів

Даний метод використовує наступну функцію для трасування зовнішніх портів:

```
private IEnumerable<ExternalPortView> TraceExternalPorts(PortType type) =>
externalPorts.Where(x => x.Type == type);
```

Приклад 6.4 – Функція TraceExternalPorts()

6.4.3 Нумерація вихідних портів

Нумерація вихідних портів проводиться аналогічно нумерації вхідних портів.

Наступний приклад містить код виклику методу TraceExternalPorts(), що була розглянута у попередньому прикладі:

```
// external outputs should be traced last
this.Log("Tracing external outputs...");
service.AddToHistory(TraceExternalPorts(PortType.Output));
```

Приклад 6.5 – Виклик метода трасування вихідних портів

6.4.4 Зовнішній цикл

Даний розділ розглядає обробку логічних елементів схеми. Кожну ітерацію перевіряється, чи всі логічні елементи та лінії були оброблені.

Наступний метод AllGatesAndWiresTraced проводить дану перевірку:

```
private bool AllGatesAndWiresTraced((int gates, int wires) total) =>
    total.gates == service.Count(nameof(GateView)) && total.wires ==
service.Count(nameof(WireView));
```

Приклад 6.6 – Метод перевірки трасуваних компонентів

У тілі циклу ініціалізується флаг anyTraced, що потрібен для виключення проблеми входження програми у безкінечний цикл.

Такий випадок можливий у разі існування помилок у схемі.

Для вирішення цієї проблеми і був доданий флаг anyTraced.

```
while (!AllGatesAndWiresTraced((total.gates, total.wires)))
{
    // flag used to handle logic errors in the scheme
    bool wiresTraced || gatesTraced = false;

    //...

    anyTraced = wiresTraced || gatesTraced;

    if (!anyTraced)
        throw new TracingErrorException;
}
```

Приклад 6.7 – Тіло циклу трасування

Флаг отримує значення з двох інших флагів: wiresTraced та gatesTraced. Якщо хоча б один з них дорівнює істині, anyTraced теж буде отримає значення істини.

Якщо під кінець циклу флаг не буде дорівнювати істині, то буде викинуто спеціальне виключення TracingErrorException.

6.4.5 Нумерація ліній

Даний розділ описує код нумерації ліній, що з’єднують логічні елементи.

Наступний код спочатку логує початок операції, далі викликає GetWiresToTrace(), що енумерує отримує список ліній. Далі перевіряється, чи містить список ліній хоча б один елемент, тобто чи буде хоча б один елемент трасований на даній ітерації. Отриманий список додається до історії

трасування за допомогою сервісу HistoryService.

```
// tracing wires
this.Log("Tracing wires...");
var tracedWires = GetWiresToTrace().ToList();
bool wiresTraced = tracedWires.Any();
service.AddToHistory(tracedWires);
```

Приклад 6.8 – Код нумерації ліній

Наступний код демонструє метод GetWiresToTrace(), що повертає лінії схеми, що можуть бути пронумеровані на даній ітерації циклу.

```
private IEnumerable<WireView> GetWiresToTrace()
{
    var exPorts = externalPorts
            .Where(p => p.Type == PortType.Input);

    var tracedGates = service.GetAll(nameof(GateView))
        .Select(x => x.TracedObject as GateView);

    var tracedWires = service.GetAll(nameof(WireView))
        .Select(x => x.TracedObject as WireView);

    // check by point OR by matrix location or external point
    return wires.Except(tracedWires)
        .Where(w => tracedGates.Any(g => g.WireStartConnects(w))
            || exPorts.Any(p => p.MatrixLocation ==
w.Connection.MatrixStart));
}
```

Приклад 6.9 – Метод GetWiresToTrace()

Алгоритм дії даного методу GetWiresToTrace():

- Отримує усі пронумеровані компоненти схеми;
- Повернути всі непронумеровані лінії схеми, що під’єднані до

компонентів, отриманих на попередньому кроці.

6.4.6 Нумерація логічних елементів

Даний розділ описує код нумерації логічних елементів схеми.

Наступний код спочатку логує початок операції, далі викликає TraceGates(), що енумерує список отриманих результатів. Далі перевіряється, чи містить список результатів хоча б один елемент, тобто чи був хоча б один елемент трасований. Отриманий список додається до історії трасування за допомогою сервісу HistoryService.

```
// tracing gates
this.Log("Tracing gates...");
var tracedGates = GetGatesToTrace().ToList();
bool gatesTraced = tracedGates.Any();
service.AddToHistory(tracedGates);
```

Приклад 6.10 – Код нумерації логічних елементів

Наступний код демонструє метод GetGatesToTrace(), що повертає логічні елементи схеми, що можуть бути пронумеровані на даній ітерації циклу.

```
private IEnumerable<GateView> GetGatesToTrace()
{
    var tracedWires = service.GetAll(nameof(WireView))
        .Select(x => x.TracedObject as WireView);

    var tracedGates = service.GetAll(nameof(GateView))
        .Select(x => x.TracedObject as GateView);

    var notTracedGates = gates.Except(tracedGates);

    // select only gates connected to traced wires
    return notTracedGates
        .Where(gate => tracedWires
```

```
            .Any(wire => gate.WireEndConnects(wire)));
}
```

Приклад 6.11 – Метод GetGatesToTrace()

Розглянутий метод функціонує наступним чином:

- Отримує всі трасовані лінії і логічні елементи;

- Повертає не трасовані логічні елементи, що під’єднані до вже трасованих ліній схеми.

Далі наведена діаграма роботи алгоритму нумерації ліній (рис. 6.6):

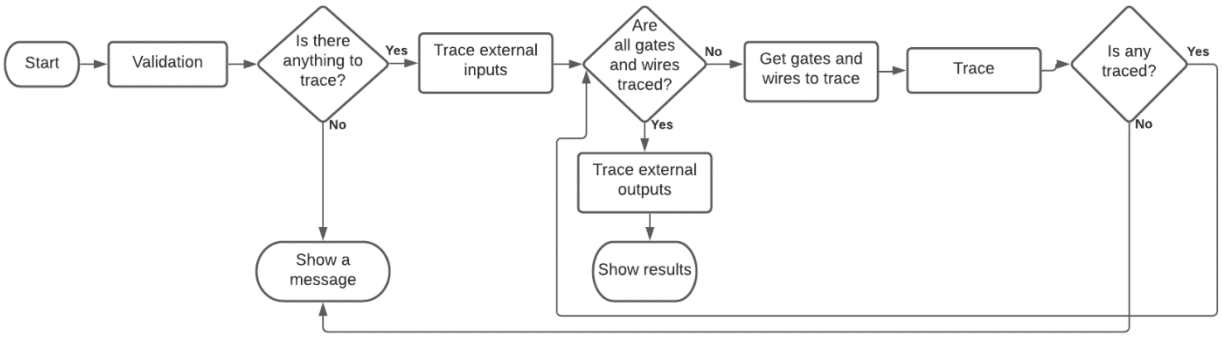


Рисунок 6.6 – Алгоритм нумерації ліній

6.5 Алгоритм моделювання

Функціональність даної програми включає в себе також і моделювання роботи схеми, а саме обробку сигналу при надходженні у логічний елемент та трансфер з виходу одного елементу до входу іншого.

Перед тим, як розпочинати розробку власного алгоритму, чи вибір одного з існуючих, потрібно провести аналіз існуючих рішень.

Існує два способи моделювання логічних схем – компілятивний і подієвий. [9]

6.5.1 Компілятивний спосіб

Компілятивний спосіб транслюється в послідовність операторів мови програмування або машинних інструкцій.

Даний спосіб моделювання вимагає попереднього ранжирування рівнів схеми і розриву зворотних зв'язків для послідовних схем. Особливістю компілятивного методу є те, що кожен логічний елемент моделюється заново на кожній ітерації моделювання, навіть, якщо зміни сигналів на його входах не відбулося.

Метод відрізняється своїм високим швидкодією, однак має істотні недоліки – цей метод не враховує затримок елементів і при кожній зміні схеми, компіляція повинна проводиться заново.

Таким чином, компілятивний спосіб моделювання, насамперед, використовується при моделюванні комбінаційних і синхронних послідовних схем.

6.5.2 Подієвий спосіб нумерації ліній

Подієвий спосіб заснований на відстеженні подій, пов'язаних зі зміною значень сигналів на входах елементів схеми.

Даний метод не має обмежень, пов'язаних з моделюванням тимчасових затримок.

У подієвому методі моделювання логічного елемента відбувається тільки в разі, коли на його входах відбулася подія. Такою подією є зміна значення сигналу хоча-б на одному його вході, а також зміна внутрішнього стану елементів пам'яті.

6.5.3 Розробка власного алгоритму

Оскільки метою даної програми є моделювання лише комбінаційних схем без можливості обробки зворотних зв'язків, і в зв'язку з тим, що реалізація традиційних методів моделювання, компіляційного і подієвого, занадто складна конкретно для даного випадку, то було прийнято рішення створити власний алгоритм моделювання.

Розроблений алгоритм має кілька ступенів виконання:

1. З колекції вибираються зовнішні входи в плату і дроти, які до них підключені.

2. Від входів значення сигналу передається на дроти.

3. Проводиться вибірка проводів, які отримали будь-яке значення від джерела, до якого вони підключені.

4. З колекції вибираються порти логічних елементів або зовнішні виходи схеми, які підключені до даних проводам.

5. Значення з обраних проводів передається на входи підключених до них елементів або зовнішніх виходів схеми.

6. Якщо всі входи елемента отримали значення, то елемент оброблює отримане значення і подає на вихід.

7. Проводиться вибірка проводів, які ще не отримали ніякого значення і елементів, які сформували вихідний сигнал.

8. Сигнал елементів передається на підключення до них дроти.

9. Перевірка наявності значення на вихідних портах схеми і повторення кроків 3-8 до тих пір, доки всі вихідні порти не будуть зберігати якесь значення сигналу.

Далі наведена діаграма алгоритму моделювання схеми:

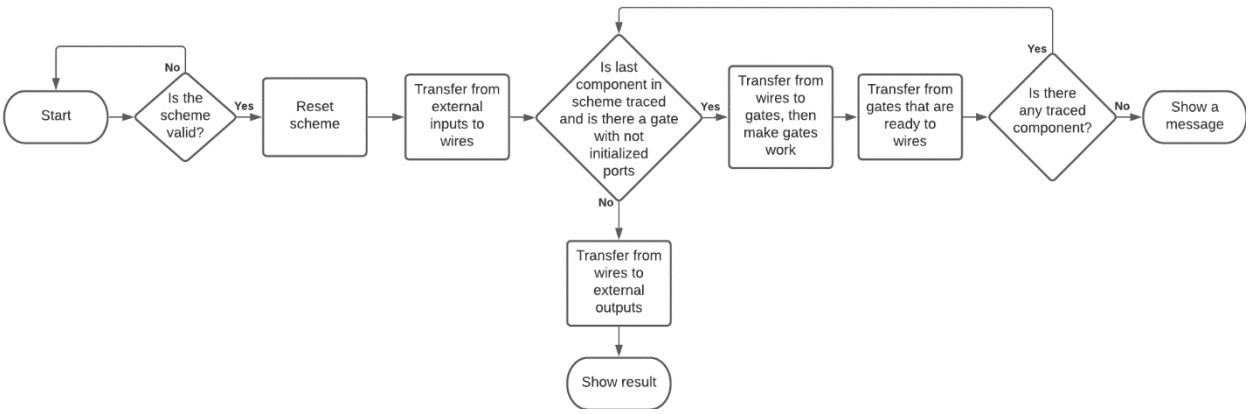


Рисунок 6.7 – Алгоритм моделювання схеми

6.5.4 Реалізація алгоритму за допомогою коду

Даний розділ містить опис конкретної реалізації методу моделювання роботи комбінаційної схеми.

У наступному лістингу приводиться сигнатура метода моделювання роботи схеми. Перш за все, метод перевіряє, чи є у схемі помилки, допущені користувачем, і повертає тип конкретної помилки, що допустив користувач. Далі за допомогою поверненого типу помилки можна сповістити користувача про знайдені помилки. [10]

Першу помилку, «exInsNotInited» можна отримати, якщо вхідні порти не були ініціалізовані, тобто подальша передача сигналів не може відбутися коректно.

Другу помилку, «gatesNotConnected» можна отримати, якщо при перевірці схеми буде виявлено, до одного з елементів не підключено жодної лінії. Таким чином, елемент або не має лінії, звідки отримати значення, або не має лінії, куди це значення можна передати.

```
public static WorkAlgorithmResult Visualize(Scheme scheme)
{
    if (scheme.gateController.GetFirstNotInitedGate() != null)
        return WorkAlgorithmResult.exInsNotInited;
    else if (!scheme.IsAllConnected())
        return WorkAlgorithmResult.gatesNotConnected;
    //...
}
```

Приклад 6.12 – Сигнатура метода моделювання

Наступний лістинг містить ініціалізацію черги з лініями за допомогою даних схеми, ініціалізацію лічильника і цикл обробки ліній схеми. Даний цикл працює до тих пір, доки всі лінії схеми не будуть оброблені та, відповідно, видалені з черги.

Ітерація циклу починається з того, що з черги ліній вибирається одна

лінія схеми і починається її обробка.

При позитивному результаті виконання моделюювання, тобто при відсутності помилок у схемі, метод повертає відповідний статус «correct».

```
    Queue<Wire> wires = new Queue<Wire>(scheme.lineController.Wires);
    int connectionNotFound = 0;
    while (wires.Count != 0)
    {
        Wire wire = wires.Dequeue();
        //...
}

    return WorkAlgorithmResult.correct;
}
```

Приклад 6.13 – Цикл обробки схеми

Наступний лістинг містить обробку елемента, що під’єднується до стартової точки лінії, що оброблюється.

Якщо даний елемент є зовнішнім і має тип IN, то лічильник обнулюється і лінія отримує значення з виходу стартового елемента.

```
Gate startGate = scheme.gateController.GetGateByWireStart(wire.Start);
if(startGate.Type == GateEnum.IN)
{
connectionNotFound = 0;
      wire.isActive = startGate.Values[0];
}
```

Приклад 6.14 – Передача значення зі стартового елементу IN

Наступний фрагмент коду містить перевірку на те, що усі вхідні порти логічного елементу ініціалізовані, тобто немає жодного вхідного порту елемента, що не отримав значення сигналу.

У такому випадку елемент передає значення лінії і лічильник приймає

значення 0.

```
else
{
if(startGate.FirstFreeValueBoxIndex() == -1)
      {
        connectionNotFound = 0;
            wire.isActive = startGate.Values[0];
}

//...
}
```

Приклад 6.15 – Передача значення з логічного елементу

У наступному лістингу оброблюється ситуація, коли елемент не готовий видати значення лінії.

У такому випадку лічильник збільшується і лінія потрапляє назад до черги. Оскільки черга працює за принципом FIFO (first-in-first-out), то обробка даної лінії буде призведена останньою.

Якщо ж кількість ліній у черзі стає більшою, ніж значення лічильнику, то програма повертає результат «schemeIsntCorrect», який означає, що у схемі допущено логічні помилки, що заважають моделюванню її роботи, але програма не може визначити їх тип.

Основною причиною отримання даної помилки можна вважати, зворотні зв’язки. Якщо у схемі є хоч один зворотній зв’язок, це призводить до неможливості її коректного моделювання.

Наприклад, якщо підключити лінію від виходу елемента до його входу, елемент не зможе отримати значення на вхід тому, що підключена лінія теж не може отримати значення, бо елемент не може його сгенерувати, адже не має значень на вході.

Таким чином, було винайдено рішення для виключення можливості входу програми у безкінечний цикл, адже, якщо програма буде мати ситуацію,

аналогічну поданій у прикладі, вона неминуче увійде в безкінечний цикл і це призведе до її зависання.

```
else
{
connectionNotFound++;
	wires.Enqueue(wire);

	if (connectionNotFound > wires.Count)
		return WorkAlgorithmResult.schemeIsntCorrect;

continue;
}
```

Приклад 6.16 – Обробка іншого випадку

У наступному фрагменті коду призводиться вибірка і обробка елементу, що є під'єднаний до лінії, що оброблюється.

Аналогічно з попередньою обробкою елемента, що підключений до стартової точки лінії, елемент, під'єднаний до кінечної точки перевіряється на тип. Якщо він має тип «OUT», тоді лічильник обнулюється і елемент приймає значення, що зберігається у лінії.

```
Gate endGate = scheme.gateController.GetGateByWireEnd(wire.End);

if (endGate.Type == GateEnum.OUT)
{
connectionNotFound = 0;
	endGate.Values[0] = wire.isActive;
}
```

Приклад 6.17 – Вібірка та обробка вихідного елементу

Якщо ж даний елемент є логічним елементом, тоді отримується індекс його вільного порту і туди записується значення, яке було отримано з

під’єднаної до нього лінії. Також обнулюється лічильник.

Останнім кроком даного методу є обробка випадку, коли останній порт елементу отримав своє значення з лінії. Тоді елемент формує вихідне значення.

```
else
{
int box = endGate.FirstFreeValueBoxIndex();

      connectionNotFound = 0;
endGate.Values[box] = wire.isActive;

      if (box == endGate.Inputs - 1)
        endGate.Work();
}
```

Приклад 3.18 – Обробка логічного елементу

6.6 Синхронізація з Presentation Layer

Даний рівень, як уже було описано раніше, відповідає за відображення даних на екрані, але він зберігає свої дочірні елементи у спеціальній колекції, що має свої особливості роботи.

Перш за все, усі дочірні елементи, що зберігаються у контейнерах, зберігаються за допомогою одного базового типу. Якщо усі елементи різних типів зберігаються у одному контейнері, вибірка та кастування їх до потрібного типу стає досить складною проблемою. Але, якщо потрібно їх взаємодія і розміщення в одному контейнері, то становиться необхідним вирішити цю проблему.

Найпростіше рішення, яке було обране – копіювання і збереження елементів у більш гнучких колекціях. [11]

6.7 Синхронізація з Storage Layer

Необхідність синхронізіції з даним рівнем обумовлена тим, що час роботи програми завжди обмежений, а створювати одну й ту ж схему з нуля є дуже незручним, а іноді може бути і зовсім неможливим, якщо вона складається з великої кількості елементів.

Проблема, пов’язана з елементами Presentation Layer полягає в тому, що елементи цього рівня мають занадто багато даних, та циклічних зв’язків, щоб їх можна було серіалізувати, десериалізувати, чи зберігати у базі даних, використовуючи Entity Framework, чи будь-яку іншу систему.

Таким чином, перед збереженням та відтворенням будь-яких класів, що відносяться до інтерфейсу користувача, є необхідним етап їх перетворення на більш прості, так звані Data Trasfer Object, або DTO-класи, що мають лише необхідні дані, а інші дані ігноруються, або відтворюються з контексту. [12]

7 РОЗРОБКА DATABASE LAYER

7.1 Серіалізація та десериалізація

Серіалізація – процес перетворення будь-якої структури даних у послідовність бітів. Зворотною до операції серіалізації є операція десеріалізації – відновлення початкового стану структури даних із бітової послідовності.

Будь-якій зі схем серіалізації властиво те, що кодування даних послідовно за визначенням, і вибірка будь-якої частини серіалізованої структури даних вимагає, щоб весь об'єкт був зчитаний від початку до кінця і був відновлений. У багатьох програмах така лінійність корисна, тому що дозволяє використовувати прості інтерфейси введення/виведення загального призначення для збереження і передачі стану об'єкта. У додатках, де важлива висока продуктивність, можливо буде доречніше використовувати складнішу, нелінійну організацію зберігання даних.

Даний спосіб підходить для збереження невеликих об’ємів даних, при незалежності від даних інших користувачів. Також, цей спосіб не може використовуватись для зберігання секретних даних, для яких важливо забезпечити безпечне зберігання. Прикладом даного типу даних є персональні дані користувача такі, як пароль, номер телефона, почтова адреса тощо.

В контексті розробки даної програми, даний спосіб підходить для збереження розміру шрифтів, теми програми тощо. [13]

7.2 Збереження у базі даних

Бази даних – це сукупність відомостей (про реальні об'єкти, процеси, події або явища), що ставляться до певної теми або завдання, організована таким чином, щоб забезпечити зручне подання цієї сукупності, як у цілому,

так і будь-якої її частини.

Основними характеристиками є:

1. Скорочення надмірності.

При відсутності централізованої БД кожен додаток буде окремо зберігати свої дані. Нерідко одні й ті ж дані використовуються кілька різних програм. Наприклад, і список співробітників у відділі кадрів, і список співробітників, записаних в бібліотеку підприємства, містять ім'я, адреса, паспортні дані. У централізованої БД такі дані можна об'єднати з повним або частковим усуненням надмірності.

2. Можливість усунення суперечливості.

Найчастіше протиріччя є наслідком надмірності. Якщо однакові дані про одну людину представлені в двох різних записах і це "роздвоєння" не враховано, то рано чи пізно два записи можуть перестати узгоджуватися: наприклад в одну запис внесуть зміни, а в іншу - немає. В цьому випадку БД стане суперечливою. Протиріч можна уникнути, усуваючи надмірність або контролюючи її. В останньому випадку може використовуватися множинне оновлення, коли при внесенні змін в одну із записів автоматично будуть змінені і записи, пов'язані з нею. [14]

3. Можливість спільного доступу до даних.

При наявності централізованої БД співробітники різних підрозділів відповідно до їх повноважень можуть спільно використовувати ці дані.

4. Можливість дотримання стандартів.

Впровадження єдиних стандартів по обробці даних набагато простіше здійснити в централізованій системі.

5. Можливість введення обмежень для забезпечення безпеки.

При централізованому зберіганні та обробці даних простіше розробити і впровадити правила розмежування доступу до них.

6. Можливість забезпечення цілісності даних.

Завдання забезпечення цілісності полягає в забезпеченні правильності і точності даних в БД. Виникнення протиріч - це приклад порушення цілісності.

Інший приклад: при централізованому зберіганні простіше організувати процедури резервного копіювання та відновлення БД.

У той же час, невдале проектування і впровадження системи БД, може спричинити ряд проблем. Основна з них пов'язана з тим, що вся робота інформаційної системи підприємства буде залежати від системи БД, і збої в ній можуть привести до катастрофічних наслідків.

В контексті розробки даної програми, даний спосіб підходить для збереження елементів та ліній схеми.

7.3 Збереження у файлі налаштування програми

Файл налаштування програми передбачає використання його для зберігання невеликих розмірів даних, таких як, наприклад, персональні налаштування користувача на даному конкретному комп’ютері, мапінги клавіш даного пристрою тощо.

В контексті розробки даної програми, даний спосіб підходить для збереження розміру шрифтів, теми програми тощо. [15]

7.4 Збереження у оперативній пам’яті

Даний спосіб передбачає те, що дані зберігаються лише підчас роботи програми, тому не може бути використаний для довготривалого зберігання інформації.

В контексті розробки даної програми, даний спосіб підходить для збереження елементів та ліній схеми підчас роботи.

7.5 Автосбереження

Автозбереження – це функція, що автоматично виконує зберігання даних. Частота залежить від налаштувань, наприклад, вона може відбуватись

кожні кілька секунд, якщо користувач зробив якісь зміни у конфігурації схеми, що є активною у даний момент часу.

Функція автозбереження є дуже зручною, якщо користувачу не зручно кожен раз зберігати стан документа або схеми, або якщо у нього можливі якісь проблеми з роботою пристрою, наприклад, розряджується батарея.

Але, є і негативні сторони даної функції – якщо відкритий файл чутливий до змін і вони повинні робитись за спеціальних умов, автоматичне збереження може переписати правильну версію тою, що не була, наприклад, прийнята або затверджена керівництвом.

7.6 Реалізація збереження у програмі

Для реалізації у даній програмі був обраний засіб серіалізації та десеріалізації.

Таким чином, сериалізацію було імплементовано наступним чином:

```
public static async Task Save(SchemeView scheme)
{
    StorageFolder folder = ApplicationData.Current.LocalFolder;

    var serialized = PrepareForSerialization(scheme);

    await Save(folder, GatePath,
JsonConvert.SerializeObject(serialized.gates));
    await Save(folder, PortPath,
JsonConvert.SerializeObject(serialized.ports));
    await Save(folder, WirePath,
JsonConvert.SerializeObject(serialized.wires));
}
```

Приклад 7.1 – Метод збереження схеми

У зазначеному методі отримується локальна директорія програми, далі для збереження кожного типу об’єкту схеми використовується метод

PrepareForSerialization(). Кожна колекція зберігається за допомогою функції Save().

Наступна функція, PrepareForSerialization(), виглядає наступним чином:

```
private static (IEnumerable<GateDto> gates, IEnumerable<PortDto> ports,
IEnumerable<WireDto> wires) PrepareForSerialization(SchemeView scheme)
{
    var gates = scheme.Gates.Select(gate => new GateDto(gate));
    var ports = scheme.ExternalPorts.Select(port => new PortDto(port));
    var wires = scheme.Wires.Select(wire => new WireDto(wire));

    return (gates, ports, wires);
}
```

Приклад 7.2 – Конвертація даних схеми для збереження

Функція Save() спочатку отримує файл, у який буде призводитись запис і пише передані дані у зазначений файл, дана функція виглядає наступним чином:

```
private static async Task Save(StorageFolder folder, string path, string
serializedData)
{
    StorageFile file = await folder.CreateFileAsync(path,
CreationCollisionOption.OpenIfExists);
    await FileIO.WriteTextAsync(file, serializedData);
}
```

Приклад 7.3 – Метод Save()

7.6 Реалізація завантаження у програмі

Завантаження схеми починається з очищення її попередніх даних, і отримання локальної директорії, у якій знаходиться додаток:

```
public static async Task Load(SchemeView scheme)
{
    scheme.Recreate();

    StorageFolder folder = ApplicationData.Current.LocalFolder;
```

Приклад 7.4 – Початок функції завантаження даних схеми

Далі проводиться завантаження логічних елементів.

Перш за все отримується директорія з логічними елементами, далі десеріалізуються дані з файлу у форматі списку з GateDto. Отриманий список перетворюється у список з GateView та додається до схеми.

```
//serialization of gates
StorageFile gateFile = await folder.CreateFileAsync(GatePath,
CreationCollisionOption.OpenIfExists);

var gates = JsonConvert.DeserializeObject<List<GateDto>>(await
FileIO.ReadTextAsync(gateFile));
var gatesToAdd = new List<GateView>(gates.Select(g => new GateView(g.Type,
g.Location, g.Inputs, g.Outputs)));
gatesToAdd.ForEach(scheme.AddToView);
```

Приклад 7.5 – Завантаження логічних елементів

Далі проводиться завантаження зовнішніх портів.

Перш за все отримується директорія з зовнішніма портами, далі десеріалізуються дані з файлу у форматі списку з PortDto. Отриманий список перетворюється у список з ExternalPortView та додається до схеми.

```
// serialization of ports
StorageFile portFile = await folder.CreateFileAsync(PortPath,
CreationCollisionOption.OpenIfExists);

var ports = JsonConvert.DeserializeObject<List<PortDto>>(await
FileIO.ReadTextAsync(portFile));
```

```
var portsToAdd = new List<ExternalPortView>(ports.Select(p => new
ExternalPortView(p.Type, p.Location)));
portsToAdd.ForEach(scheme.AddToView);
```

Приклад 7.6 – Завантаження зовнішніх портів

Далі проводиться завантаження дротів.

Перш за все отримується директорія з дротами, далі десеріалізуються дані з файлу у форматі списку з WireDto. Отриманий список перетворюється у список з WireView та додається до схеми.

```
//serialization of wires
StorageFile wireFile = await folder.CreateFileAsync(WirePath,
CreationCollisionOption.OpenIfExists);

var wires = JsonConvert.DeserializeObject<List<WireDto>>(await
FileIO.ReadTextAsync(wireFile));
var wiresToAdd = new List<WireView>(wires.Select(w =>
{
    WireView wireView = new();
    wireView.SetConnection(w.Connection);
    return wireView;
}));
wiresToAdd.ForEach(scheme.AddToView);
```

Приклад 7.7 – Завантаження дротів

ВИСНОВОК

В результаті виконання кваліфікаційного проекту була запропонована архітектура для створення системи для моделювання комбінаційних логічних схем, і трасування ліній даних схем. Крім цього були вивчені принципи роботи логічних елементів.

В ході дослідження виконано аналіз існуючих алгоритмів моделювання комбінаційних логічних схем. Після проведення аналізу алгоритмів був пропонований власний алгоритм.

Додатково, були проаналізовані існуючі алгоритми трасування ліній схеми. Після проведення аналізу алгоритмів також був пропонований власний алгоритм.

Дослідження архітектурних рішень призвело до вибору відповідної архітектури слоїв, з яких складається програма.

Аналіз архітектури дозволив інтегрувати імплементацію різних алгоритмів трасування і моделювання комбінаційних логічних схем. Існуючі приклади програмних лістингів були протестовані на швидкість та коректність масштабування та роботи.

В ході кваліфікаційної роботи був створений приклад програми, що дозволяє створювати та редагувати комбінаційні схеми. Програма дозволяє проводити трасування, проводити візуалізацію роботи ліній та різних логічних елементів, в яких можна конфігурувати кількість інформаційних входів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Хабр [Електронний ресурс] – Режим доступу: https://habr.com/ru/post/276593/

2. Гимади Э. Х. Выбор оптимальных шкал в одном классе задач типа размещения, унификации и стандартизации // Управляемые системы: Сб. науч. тр. Новосибирск: Ин-т математики СО АН СССР, 1970. Вып. 6. С. 57-70.

3. Трубин В. А., Шарифов Ф. А. Простейшая многоэтапная задача размещения на древовидной сети // Кибернетика и системный анализ. 1992. N 6. С. 128-135.

4. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. М: Мир, 1982.

5. Прайс, М. C# 8 и .NET Core. Разработка и оптимизация [Текст] : пер. с англ. – К. : Прайс; М. : Реал-бук, 1998. – 462 с.

6. Аналіз застосування концепції SOLID [Текст] : матеріали 23-го междунар. молодіж. форум IV, 22 марта 2021 р. Харків / редкол. : І. В. Рубан (отв. ред.). – Харків : Темп, 2021. – 57 с.

7. Об’єктно-орієнтоване програмування [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Об%27єктно-орієнтоване_програмування#cite_ref-MeyerCh3_4-0

8. Логічні елементи [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/Логічні_елементи#Послідовні_цифрові_пристрої

9. Алгоритми моделювання логічних схем [Текст] : матеріали IV междунар. науч.-практ. конф., 22 марта 2021 р. Харків / редкол. : І. В. Рубан (отв. ред.). – Харків : Темп, 2021. – 57 с.

10. Мова програмування C# [Електронний ресурс] – Режим доступу: https://metanit.com/sharp/tutorial/1.1.php

11. Албахари, Д. C# 9.0: Полный справочник [Текст] : пер. с англ. – К. : Албахари; М. : O'Reilly, 2021. – 1061 с.

12. Мова програмування C# [Електронний ресурс] – Режим доступу: https://ru.wikipedia.org/wiki/C_Sharp

13. Гриффитс, И. Программируем на C# 8.0 [Текст] : пер. с англ. – К. : Гриффитс; М. : Apress, 2021. – 1383 с.

14. Українсько-англійський словник з радіоелектроніки [Електронний ресурс] – Режим доступу: https://e2u.org.ua/s?w=object-oriented&dicts=16&highlight=on&filter_lines=on

15. Троелсен, Э., Джепикс, Ф. C# 9 и .NET 5 для профессионалов [Текст] : пер. с англ. – К. : Вакlер; М. : Реал-бук, 1998. – 462 с.