

User name:  
**Volodymyr Matiiivskyi**

Check date:  
**08.01.2023 18:48:56 EET**

Report date:  
**08.01.2023 18:56:38 EET**

Check ID:  
**1013374998**

Check type:  
**Doc vs Internet + Library**

User ID:  
**100010994**

File name: **Савенко Н. В**

Page count: **58** Word count: **11127** Character count: **78031** File size: **254.11 KB** File ID: **1013147490**

## 7.5% Matches

Highest match: **1.89%** with Internet source (<http://delphikingdom.com/asp/viewitem.asp?catalogid=838>)

7.5% Internet sources 73

Page 60

No Library sources found

## 0.3% Quotes

Quotes 1

Page 61

Exclusion of references is off

## 0% Exclusions

No exclusions

## Modifind

Text modifications detected. Find more details in the online report.

Replaced characters 1

## ВСТУП

При вивченні матеріалу, який несе певну складність в освоєнні, було б корисно показати наочно роботу якогось пристрою або процесу. Великі програмні комплекси здатні реалізувати ці процеси, але вони не можуть показати наочно роботу пристроїв чи процесів, до того ж вони мають високу вартість, вони складні у вивченні, так як вони мають широку орієнтацію і займають великий обсяг оперативної пам'яті ПК. Тому дипломну роботу присвячену розробці оригінального програмного продукту для моделювання можна вважати актуальною. Додаток, що розробляється, може бути використаний в навчальному процесі. Даний програмний продукт повинен забезпечувати:

- учням можливість виявляти недоліки в змодельованих дослідах і програмно усувати їх.

- містити завдання, для вирішення яких необхідно використовувати комп'ютер не лише для розрахунків, а для якісного і кількісного моделювання досліджуваного явища і спостереження за ним.

**Об'єкт проектування** - програмне забезпечення для моделювання процесів комп'ютерної логіки.

**Мета роботи** - розробка програмного забезпечення для моделювання процесів комп'ютерної логіки.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

1. Зробити аналіз відомих програмних засобів для моделювання.
2. Розробити вимоги до створюваного програмного продукту.
3. Провести аналіз теоретичних основ положень теорії графів і двійкової математики.
4. Розробити програмне забезпечення для моделювання процесів комп'ютерної логіки.

**Методи дослідження** - техніко-економічний з використанням комп'ютерних технологій, технічний аналіз, методи моделювання інформаційних процесів.

У першому розділі проведено аналіз програм моделювання та аналізу цифрових і аналогових пристроїв, також він присвячений аналізу вимог до створення програмного продукту.

Другий розділ присвячений аналізу теоретичних основ побудови програми моделювання, який включає аналіз положень двійкової математики та аналіз положень теорії графів.

Третій розділ присвячено етапам розробки програми, описані елементи інтерфейсу програми, їх призначення та функції наведені принципи логіки побудови програми.

## **РОЗДІЛ 1**

### **ПОРІВНЯЛЬНИЙ АНАЛІЗ ПРОГРАМ МОДЕЛЮВАННЯ ТА АНАЛІЗ ВИМОГ ДО СТВОРЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

#### **1.1 Порівняльний аналіз програм моделювання та аналізу цифрових і аналогових пристроїв**

З метою створення додатка моделювання процесів комп'ютерної логіки, був проведений аналіз найбільш використовуваних програмних комплексів в цій області. У таблиці 1.1 наведено даний аналіз.

Таблиця 1.1.

#### **Призначення і порівняння програм моделювання та аналізу цифрових, аналогових та електронних пристроїв**

<b>№ п/п</b>	<b>Найменування САПР</b>	<b>Призначення</b>
1	Electronics Workbench	Оціночне схемотехнічне моделювання та аналіз цифрових, аналогових і гібридних

		електронних пристроїв з метою ілюстрації опису процесів, що відбуваються в них, і підтвердження коректності розрахунків.
2	Electronics Workbench Multisim	Оціночне схемотехнічне моделювання та комплексний аналіз складних пристроїв радіоелектроніки і телекомунікацій, в тому числі програмованих логічних інтегральних схем (ПЛІС) і базових матричних кристалів (БМК).
3	Spectrum MicroCAP	Оціночне схемотехнічне моделювання та комплексний аналіз різних пристроїв радіоелектроніки і телекомунікацій. Аналіз локальних теплових процесів на основі моделей реальних електронних компонентів.
4	APLAC	Проектування електронних пристроїв і проведення комплексного моделювання в тимчасовій і частотній областях, в тому числі в СВЧ діапазоні
5	MicroWave Office	Розробка електронних компонентів мікрохвильового, радіо і СВЧ діапазонів. Проектування і моделювання складних пристроїв НВЧ діапазону, пристроїв цифрової обробки сигналів, елементів систем стільникового, пейджингового, транкінгового і супутникового зв'язку, радіолокації, навігації та телеметрії.

		Проектування і аналіз тривимірних моделей мікросмушкових пристроїв
6	OrCAD	<p>Проектування і аналіз аналогових, цифрових і гібридних електронних пристроїв.</p> <p>Моделювання зовнішніх факторів, що впливають і причин нестабільності функціонування електронних схем.</p> <p>Моделювання та імітація поведінки виробів в реальному світі.</p> <p>Обчислення впливів і симуляція активного та пасивного навколишнього середовища.</p>
7	MathCAD	Виконання складних математичних розрахунків і моделювання пристроїв, режимів і процесів в електронних схемах
8	MatLAB	<p>Виконання складних математичних розрахунків і моделювання електронних і електромеханічних та механічних систем</p> <p>Моделювання динамічних систем, в тому числі систем автоматичного управління, телекомунікаційних систем, сигналів і приймально-передавальних пристроїв.</p>
9	Siam	Проектування, моделювання та аналіз аналогових і цифрових систем

		автоматичного управління (САУ).
10	Altera  Max Plus	Проектування і аналіз програмованих логічних інтегральних схем (ПЛИС) і базових матричних кристалів (БМК).
11	Accel PCAD  и  Accel EDA	Проектування корпусів мікросхем, мікрозборок і електрорадіоелементів.  Проектування друкованих плат.  Підготовка структурних функціональних і принципових електричних схем.
12	Компас	Підготовка креслень і плакатів, в тому числі 3-х мірних зображень і проекцій.  Розробка технологічних процесів і технологічної документації для виробництва радіоелектронних та телекомунікаційних пристроїв і систем
13	SolidWorks	Підготовка креслень і плакатів, в тому числі 3-х мірних зображень і проекцій.  Розробка технологічних процесів і технологічної документації. Виконання твердотілого моделювання.
14	Omega PLUS	Програмне забезпечення для аналізу цілісності проекту і моделювання електромагнітної сумісності
15	BETAsoft	Програмне забезпечення для аналізу теплових процесів в радіоелектронних пристроях

З аналізу наведеного в таблиці 1.1 можна визначити вимоги створення проєктованого додатку:

1. Програма повинна мати доступний інтерфейс, враховувати всі можливі критерії та умови, що виникають в процесі роботи.
2. Програма повинна мати максимально простий інтерфейс для простоти використання.
3. Займати малий обсяг дискового простору і оперативної пам'яті.
4. Програма повинна бути придатним середовищем для виконання лабораторних робіт.

## **1.2 Аналіз вимог до створення програмного забезпечення**

### **1.2.1 Особливості побудови інтерфейсу**

Розробка будь-якого програмного засобу насамперед має на увазі створення призначеного для користувача інтерфейсу, який є традиційним засобом взаємодії користувача з програмним продуктом. Якість виконання призначеного для користувача інтерфейсу безпосередньо впливає як на зручність роботи з програмним засобом, так і на ефективність його використання. Як правило, користувачі вимагають диференційованого підходу і, отже, програмне забезпечення, що застосовується ними, повинно мати інтерфейс, що враховує особливості користувача. Таким чином, очевидно, що головною вимогою, що пред'являються до програмного інтерфейсу, є можливість ефективного управління ресурсами і зручність використання, тобто інтерфейс повинен володіти мобільністю, що дозволяє регулювати його функціональні характеристики відповідно до вимог користувача.

#### **Принципи динамічного інтерфейсу.**

В результаті дослідження даного питання була визначена концепція побудови інтерфейсу програмних засобів виражена в принципах динамічного інтерфейсу. Були виділені наступні принципи:

- багатовіконність;
- підтримка різних пристроїв управління і введення;
- використання графічних символів для представлення об'єктів;
- гнучкість;
- об'єктна орієнтованість;
- розширюваність;
- мультимедійність;
- дружність по відношенню до користувача.

Зрозуміло, кожен з перерахованих принципів не є новим, але, будучи реалізованими разом, вони надають інтерфейсу програмного засобу нову якість, роблячи його придатним для використання широким колом користувачів. Розглянемо деякі з цих принципів більш детально.

#### Гнучкість.

Гнучкість є базовою характеристикою динамічного інтерфейсу. Рівень гнучкості призначеного для користувача інтерфейсу визначається відповіддю на питання: чи існує можливість настройки графічного інтерфейсу користувача?

Управління налаштуванням може бути лексичного рівня, наприклад, користувач визначає, де і як розташовані меню і інші інтерактивні графічні об'єкти на екрані, чи використовувати для залучення уваги користувача звуковий або світловий сигнал. Управління може бути частиною синтаксису взаємодії, тобто структура діалогу може змінюватися користувачем, з тим, щоб змінювати існуючі команди, додавати нові функціональні можливості в призначений для користувача інтерфейс.

Таким чином, основна ідея гнучкості динамічного інтерфейсу полягає в наданні користувачу можливості самостійного налаштування інтерфейсу



шляхом безпосереднього маніпулювання, з урахуванням його професійних та інших вимог, а також синтезу нового інтерфейсу.

### **Об'єктна орієнтованість.**

Об'єктно-орієнтований характер динамічного інтерфейсу на сьогоднішній день дозволяє використовувати один з найперспективніших способів, що забезпечують гнучкість призначеного для користувача інтерфейсу. Використання інтерактивного середовища, яка визначається інтерфейсними засобами, графічними символами (ікони, кнопки, меню і т.д.), дозволяє не просто розглядати інтерфейс користувача як віконну область екрану з комплектом графічних елементів і певним функціональним набором, але і виділяти в ньому окремі об'єкти, змінювати їх розташування як в рамках самого інтерфейсу, так і за його межами, а також оформлення, функції і т.п.

Типовими видами маніпуляції з об'єктами є:

- -переміщення;
- -зміна розмірів;
- -обертання і т.п.

Наприклад, за допомогою "миші", захоплюється кнопка, яка переміщується за межі вікна користувальницького інтерфейсу і фіксується в певній точці екрану, змінюється розмір шляхом "розтягування".

Важливо відзначити, що при виконанні такого роду дій велике значення має простота і ефективність реалізації завдання, тобто кінцева мета повинна досягатися найкоротшим шляхом.

### **Розширюваність.**

Особливе місце при розгляді властивостей динамічного інтерфейсу займає розширюваність. В даному аспекті під розширюваністю розуміється можливість додавання (видалення) нових функцій, використання додаткових пристроїв введення-виведення. На даний момент спостерігається стрімкий

розвиток комп'ютерної техніки, в зв'язку з чим апаратна частина комп'ютера застаріває практично за кілька років, корисне життя програмного засобу може тривати значно довше (наприклад, всім відомий Norton Commander). Таким чином, універсальність програмного продукту безпосередньо пов'язана з можливістю розширення функціонального набору, підтримкою нових пристроїв, використанням нових технологій.

#### **Дружність по відношенню до користувача.**

Ефективність використання програмного продукту, особливо на початкових етапах, багато в чому залежить від тієї допомоги, яку надає розробник. Програмний засіб має мати можливість навчання, причому навчання користувачів різного рівня підготовки. Велике значення для новачків має інтерактивна підказка, яка має безпосередній вплив на швидкість навчання і, отже, на ефективність роботи з програмним засобом. Функціональний набір призначеного для користувача інтерфейсу повинен передбачати можливість відключення інтерактивної підказки, так як для користувача більш високого рівня вона не несе корисної інформації і діє відволікаючи.

#### **1.2.2 Вимоги до розроблюваного програмного забезпечення**

До основних етапів розробки програмного забезпечення відносяться:

- - стратегічне планування;
- - аналіз вимог щодо розроблюваного програмного забезпечення;
- - проектування (попереднє і детальне);
- - кодування (програмування);
- - тестування і налагодження;
- - експлуатація та використання.

Кожному етапу відповідає набір результатів і документації, які є вихідними даними для наступного етапу. На закінчення кожного етапу

проводиться верифікація документів і рішень з метою перевірки їх відповідності початковим вимогам замовника. Етапи стратегічного планування та аналізу вимог використовуються для визначення найзагальніших вимог до програмної системи. Дані етапи передбачають вирішення наступних завдань:

- визначення доцільності розробки і порівняння з аналогами;
- визначення необхідних ресурсів для вирішення завдання;
- специфікація вимог до системи у вигляді "що вона повинна робити", але не у вигляді "як це реалізувати";
- перевірка коректності і можливості бути реалізованими вимог.

На етапі проектування створюється структура майбутньої програмою системи. Фази проектування:

- проектування архітектури, включає в себе визначення складу підсистем;
- специфікація підсистем, визначає специфікацію кожної підсистеми;
- проектування інтерфейсу, визначає інтерфейс кожної підсистеми, тобто метод взаємодії даної підсистеми з іншими;
- проектування компонентів, кожна підсистема розділяється на компоненти;
- проектування структур даних, визначає, де і як зберігаються дані;
- проектування алгоритмів, визначаються алгоритми обробки даних.

Етап кодування передбачає вибір мови програмування і складання тексту програми (кодування), а також, можливо, виконання тестування і налагодження окремих фрагментів.

Етап тестування і налагодження включає виконання комплексного тестування всієї програмної системи спеціальною групою і виправлення помилок.

На етапі супроводу і експлуатації програмна система здається в експлуатацію, відбувається обслуговування користувачів, можливо усунення незначних помилок (зараз це робиться повсюдно за допомогою поширення так званих **patch** - файлів).

Керуючись усіма вище розглянутими принципами розробки програми, була попередньо спроектована форма додатка, яка приведена на рис.1.1.

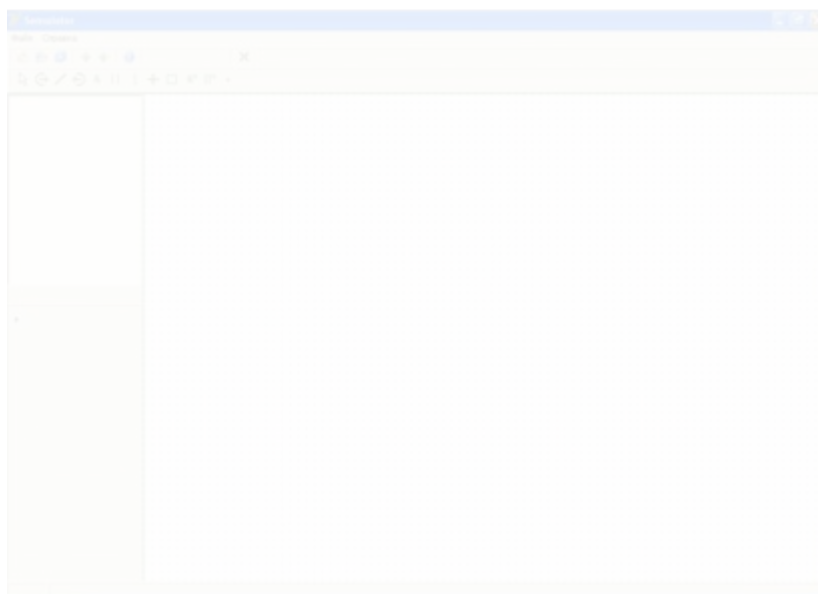


Рис. 1.1 Форма майбутнього програми моделювання процесів комп'ютерної логіки

### 1.3. Висновки до розділу

В результаті аналізу найбільш використовуваних програмних комплексів моделювання та аналізу вимог створення програмного забезпечення, а також визначення вимог до побудови інтерфейсу, була попередньо спроектована форма додатка, яка представлена на рисунку 1.1.

11

З рисунку 1.1 видно, що додаток буде відповідати тим критеріям і вимогам, які були поставлені:

1. Програма буде мати доступний інтерфейс, враховувати всі можливі критерії та умови, що виникають в процесі роботи.
2. Програма матиме максимально простий інтерфейс для простоти використання.

## **РОЗДІЛ 2**

### **АНАЛІЗ ТЕОРЕТИЧНИХ ОСНОВ ПОБУДОВИ ПРОГРАМИ МОДЕЛЮВАННЯ**

Для розробки програми моделювання процесів комп'ютерної логіки необхідно досліджувати принципи двійкової математики, зокрема, в мові програмування Object Pascal, так як всі процеси в ПК відбуваються саме на основі двійкової математики. Так само необхідно ознайомитися з теорією графів, так як для побудови логічної моделі обробки і проходження сигналу в будь-якій логічній схемі, без теорії графів просто не обійтися.

#### **2.1 Аналіз положень двійкової математики для використання їх в розроблюваній програмі**

Поряд зі звичайними логічними операціями над логічними типами, часто доводиться виконувати операції і над окремими бітами, зазвичай використовуваними, як прапори. Для ефективної роботи необхідно розуміння логічних операцій.

Паскаль підтримує такі логічні операції:

**AND** - логічне І;

**OR** - (включають) логічне АБО;

**XOR** - (виключають) логічне АБО;

**NOT** - заперечення або інверсія біта;

**SHL** - логічний зсув вліво;

**SHR** - логічний зсув вправо.

Інші логічні операції над числами в Паскаль не включені, але доступні через асемблерні вставки.

Кожен біт може мати тільки два стани БРЕХНЯ (FALSE) або ІСТИНА (TRUE).

Стан біта можна описувати і іншими словами, частина яких прийшла з математики, частина з електроніки, частина з логіки.

Для значення БРЕХНЯ, альтернативні варіанти такі - НІ, НУЛЬ, ВИМКНУТО, НЕ ВСТАНОВЛЕНО, СКИНУТО, FALSE, F, 0, - і інші.

Для значення ІСТИНА, альтернативні варіанти такі - ТАК, ОДИНИЦЯ, ВКЛЮЧЕНО, УСТАНОВЛЕНО, ЗВЕДЕНО, TRUE, T, 1, + та інші.

Розглянемо ці операції окремо

**AND** - логічне І, ця операції виглядає так

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Вираз істинно, коли істинні обидва біта. Приказка «І там І там»

**OR** - (включають) логічне АБО, ця операції виглядає так

<b>A</b>	<b>B</b>	<b>Y</b>
0	0	0
0	1	1
1	0	1
1	1	1

Вираз істинно, коли правдивий хоча б один біт. Приказка «АБО там АБО там, включаючи і там і там»

**XOR** - (виключають) логічне АБО, ця операції виглядає так

<b>A</b>	<b>B</b>	<b>Y</b>
0	0	0
0	1	1
1	0	1
1	1	0

Вираз істинно, коли правдивий тільки один біт. Приказка «АБО там АБО там, виключаючи і там і там»

**NOT** - заперечення або інверсія біта, ця операції застосовується тільки до одного біту, дія проста, поточне значення біта змінюється на протилежне.

<b>A</b>	<b>Y</b>
0	1
1	0

**SHL** - логічний зсув вліво, операції застосовується тільки до групи бітів, одного з цілочисельних типів Паскаля, наприклад до байту, слову і т.д.

Зрушення байта вліво на два розряди.

Розряд	В	В	В	В	В	В	В	В
и	7	6	5	4	3	2	1	0
До	1			1	1	1		1000
Після	0	1	1	1		1	00	0

Байт зміщується вліво на один або більше розрядів, позиції праворуч заміщуються нулями, позиції зліва губляться.

**SHR** - логічний зсув вправо, операції застосовується тільки до групи бітів, одного з цілочисельних типів Паскаля, наприклад до байту, слову і т.д.

Зрушення байта вправо на один розряд.

Розряди	В	В	В	В	В	В	В	В
	7	6	5	4	3	2	1	0
До	1			1	1	1		1000
Після	0	1	0	0	1	1	1	0

Зрушення байта вправо на два розряди.

Розряд	В	В	В	В	В	В	В	В
и	7	6	5	4	3	2	1	0
До	1			1	1	1		1000
Після	0		01			1	1	100

Байт зміщується вправо на один або більше розрядів, позиції зліва заміщуються нулями, позиції праворуч губляться.

На цьому опис операцій закінчується, і переходимо до практичних прикладів. Але спочатку трохи слів про нотації. Числа в символьній формі прийнято відображати, так що б молодші розряди були праворуч, а рядки зліва, при цьому якщо використовується вирівнювання, то воно теж підпорядковується цим правилам. Нумерація розрядів починається з нуля відповідно до ступеню розряду і описується формулою  $K \cdot M^N$ , де K це



коефіцієнт в діапазоні від 0 до  $M-1$ ,  $M$  це основа числа, а  $N$  це ступінь. Число в ступені 0 для всіх основ дорівнює 1.

Подивимося на прикладі такої таблиці для чотирьох основних основ.

Для числа 100

Основа	Значення	Формула
2	4	$1*2^2 + 0*2^1 + 0*2^0$
8	64	$1*8^2 + 0*8^1 + 0*8^0$
10	100	$1*10^2 + 0*10^1 + 0*10^0$
16	256	$1*16^2 + 0*16^1 + 0*16^0$

Для числа 123

Основа	Значення	Формула
2	X	Не допустима комбінація
8	83	$1*8^2 + 2*8^1 + 3*8^0$
10	123	$1*10^2 + 2*10^1 + 3*10^0$
16	291	$1*16^2 + 2*16^1 + 3*16^0$

#### Установка біта.

Для установки окремого біта або групи бітів використовується операція АБО, використання ілюструється нижче наведеним кодом у вигляді окремої функції і результатом виконання у вигляді таблиці.

```
function SetBit (Src: Integer; bit: Integer): Integer;
begin
  Result := Src or (1 shl Bit);
end;
```

Тут відбувається наступне:

Спочатку ми розраховуємо позицію біта -  $(1 \text{ shl Bit})$ , потім встановлюємо отриманий біт і повертаємо результат через визначену змінну Result.

Приклад використання:

DummyValue: = SetBit (DummyValue, 2);

Розряди	В	В	В	В	В	В	В	В
	7	6	5	4	3	2	1	0
До (1)	1		1	100	1	0	1	
Після	1	0	0	1	1	1	0	1
До (2)	1	0	0	1	1	0	0	1
Після	1	0	0	1	1	1	0	1

Як бачимо, незалежно від початкового стану біта, після виконання операції біт стає рівним одиниці.

#### Скидання біта.

Для скидання окремого біта або групи бітів використовується операція І спільно з інверсною маскою, використання ілюструється нижче наведеним кодом у вигляді окремої функції і результатом виконання у вигляді таблиці.

```
function ResetBit (Src: Integer; bit: Integer): Integer;
begin
  Result: = Src and not (1 shl Bit);
end;
```

Тут відбувається наступне:

Спочатку ми розраховуємо позицію біта -  $(1 \text{ shl Bit})$ , потім за допомогою операції NOT інвертуємо отриману маску, встановлюючи не доторкані біти маски в одиницю, а задіяний біт в нуль, потім скидаємо цей біт, а результат повертаємо через визначену змінну Result.

Приклад використання:

DummyValue: = ResetBit (DummyValue, 2);

Розряди	В	В	В	В	В	В	В	В
	7	6	5	4	3	2	1	0
До (1)	1		1	100	1	0	1	
Після	1	0	0	1	1	0	0	1
До (2)	1	0	0	1	1	0	0	1
Після	1	0	0	1	1	0	0	1

Як бачимо, незалежно від початкового стану біта, після виконання операції біт стає рівним нулю.

### Перемикання біта.

Для перемикання окремого біта або групи бітів використовується операція виключають АБО, використання ілюструється нижче наведеним кодом у вигляді окремої функції і результатом виконання у вигляді таблиці.

```
function InvertBit (Src: Integer; bit: Integer): Integer;
begin
  Result: = Src xor (1 shl Bit);
end;
```

Тут відбувається наступне:

Спочатку ми розраховуємо позицію біта - (1 shl Bit), потім за допомогою операції XOR перемикаємо біт, а результат повертаємо через визначену змінну Result.

Приклад використання:

DummyValue: = InvertBit (DummyValue, 2);

Розряди	B7	B6	B5	B4	B3	B2	B1	B0
До (1)	1		1	100	1	0	1	
Після	1	0	0	1	1	0	0	1
До (2)	1	0	0	1	1	0	0	1
Після	1	0	0	1	1	1	0	1

Як бачимо, стан біта B2 змінюється на протилежне.

### Помилки при роботі з бітами.

Наприклад, для складання біт ми можемо використовувати два варіанти або операцію + або операцію OR. Перший варіант є помилковим.

AnyValue + 2, якщо біт два встановлений, то в результаті цієї операції відбудеться перенесення в наступний розряд, а сам біт виявиться скинутим замість його установки, так можна робити тільки якщо є впевненість в результаті, то якщо заздалегідь відомо початкове значення. А от у випадку використання варіанта AnyValue or 2, такої помилки не відбудеться. Теж відноситься до операції віднімання для скидання біта. faAnyFiles - faDirectory помилки не дасть, а ось AnyFlags - AnyBit може, дати правильний варіант, а може ні. Зате AnyFlags and not AnyBit завжди дасть те що саме, використання цієї техніки буде правильніше і для роботи з атрибутами файлів - faAnyFiles and not faDirectory.

Ще одна розповсюджена помилка, це логічна при виконанні операцій над групами біт. Наприклад невіруючих виконувати операцію порівняння над наступною конструкцією AnyFlags and 5 <> 0, якщо істина повинна бути при установці обох біт, треба писати так AnyFlags and 5 = 5,

зате якщо влаштовує істина при установці будь-якого з біт, вираз AnyFlags and 5 <> 0 буде вірним.

## 2.2 Аналіз положень теорії графів

Завдання обходу графа, тобто, побудови маршруту, що проходить по всіх ребрах графа, добре відома. Її спеціальним випадком є завдання китайського листоноші, в якій обхід повинен мати мінімальну довжину або мінімальна вага для графа із заданими вагами дуг. Для орієнтованого графа завдання ускладнюється, оскільки маршрут повинен проходити кожне орієнтоване ребро (дугу) тільки в напрямку його орієнтації.

У більшості робіт передбачається, що граф заданий явно до побудови його обходу. Більш складним є випадок, коли до початку роботи про графа нічого невідомо і ми отримуємо інформацію про пристрій графа в процесі його обходу. Це відома задача обходу лабіринту людиною або пристроєм, що знаходиться всередині нього і не мають плану лабіринту. Дузі графа відповідає коридор лабіринту, а вершині - перехрестя. Перебуваючи на перехресті, ми бачимо що виходять з нього коридори, але ми не знаємо, куди веде той чи інший коридор, до тих пір, поки не пройшли по ньому до наступного перехрестя. Для виконання нашого завдання ми, по-перше, маємо деяку внутрішню пам'ять (блокнот в руках людини), куди можемо записувати отриману інформацію про пройденої частини лабіринту, і, подруге, робити позначки в пройдених перехрестях і коридорах. Орієнтованому графу відповідає лабіринт, в якому кожен коридор закритий з обох сторін дверима: вхідні двері відкриваються тільки з перехрестя, а вихідна - тільки зсередини коридору, що дозволяє рухатися по кожному коридору тільки в одному напрямку.

Алгоритм, який працює на такому не заданому заздалегідь графі, будемо називати ненадлишковим алгоритмом. Такі алгоритми називаються в літературі також online-алгоритмами. Його окремим випадком, коли внутрішня пам'ять обмежена кінцевим числом станів, є робот (кінцевий автомат) на графі як різновид машини Тьюринга. Замість стрічки у нас є граф, комірки стрічки відповідає вершина графа, а рух вліво або вправо

замінюється на перехід по одній з дуг, що виходять з поточної вершини графа. (Щоб автомат робота був кінцевим, граф повинен мати обмеження зверху на полустепені виходу вершини - число дуг, що виходять. Це обмеження можна зняти, якщо кожній дузі також поставити у відповідність комірку і комірки всіх дуг із загальним початком зв'язати в цикл.)

Останнім часом завдання обходу орієнтованих графів стало особливо актуальним у зв'язку з тестуванням кінцевих автоматів, точніше, об'єктів, що розглядаються як кінцеві автомати. Написано досить багато робіт про тестування кінцевих автоматів. Наприклад, ґрунтовний огляд таких робіт представлений в статтях Lee і Yannakakis і Bochmann і Petrenko.

Автомат визначається безліччю своїх станів і переходів. Перехід автомата - це четвірка  $(v, x, y, v')$ , де  $v$  - престан,  $x$  - стимул,  $y$  - реакція,  $v'$  - постстан. Зазвичай автомат задається графом станів, вершини якого - стани, а дуги - переходи. Автомат (граф станів) усюди визначено, якщо в кожному стані  $v$  допустимо кожен стимул  $x$ , тобто, існує хоча б один перехід виду  $(v, x, y, v')$ . В іншому випадку, автомат частково визначено. Автомат (граф станів) детермінований, якщо престан і стимул однозначно визначають реакцію і постстан.

Якщо граф станів автомата відомий, то зазвичай цікавляться питаннями, чи задовольняє він тим чи іншим вимогам. Ці завдання вирішуються аналітичними методами і про тестування зазвичай не йдеться. Тестування потрібно тоді, коли граф станів автомата невідомий. Автомат розглядається як «чорний ящик»: ми можемо подавати на автомат стимули і отримувати відповідну інформацію про виконаний перехід, тобто, в загальному випадку, про реакцію і постстан. Завданням тестування є перевірка того, що тестований автомат задовольняє наперед заданим специфікаційні вимогам. Це тестування відповідності (conformance testing) в широкому сенсі. У загальному випадку специфікація не має на увазі перевірки кожного переходу автомата. Якщо, наприклад, ми хочемо

перевірити, що число станів автомата не менш заданого, то тестування припиняється, як тільки ми в цьому переконуємося, і залишилися неперевіреними переходи нас не цікавлять. Однак, такі вимоги є скоріше винятком, ніж правилом. Зазвичай нас цікавить повна функціональність автомата, і нам потрібна перевірка кожного його переходу. Таке тестування спирається на наступні припущення.

**Зміна стану.** Стан змінюється тільки в результаті тестового впливу, тобто, подачі стимулу на автомат. З одного боку, це означає, що автомат знаходиться під виключним управлінням тесту і ніхто «не заважає» тестування, точніше, такий сторонній вплив не змінює спостережувану функціональність автомата. (Інформацію про тестування за наявності таких «перешкод» можна знайти в роботах по тестуванню «сірим ящиком» або «напівкерованих» тестуванню.) З іншого боку, це означає, що у тесту немає інших засобів змінити стан автомата. Якби безліч станів було відомо (невідомі тільки переходи) і була можливість довільно змінювати стан за допомогою прямого запису або спеціальної (нетестуємої) операції, завдання перебору всіх переходів автомата стала б очевидною. Апріорне знання про безліч станів реалізації - сильна вимога; зазвичай ми дізнаємося про неї тільки в процесі тестування і тільки «поелементно», тобто, про існування стану ми дізнаємося тільки після переходу в нього.

**Допустимість стимулів.** У кожен момент часу ми можемо тим чи іншим способом дізнатися, які стимули можна подавати на автомат. Зрозуміло, що в іншому випадку ні про яке тестування не може йти мова. Зауважимо, що в багатьох роботах ця проблема обходиться припущенням, що автомат усюди визначено, тобто, у всіх станах допустимі всі стимули з його алфавіту стимулів. Слід також зазначити, що на практиці нам важлива допустимість тільки «цікавлячих» нас стимулів, з яких проводиться тестування, тобто, стимулів, визначених у моделі. Якщо в реалізації допустимі якісь інші стимули, то це не впливає на тестування. Фактично, це

означає, що для реалізації  $R$  і моделі  $M$  тестується підавтомат  $R(M) \dot{I}R$ , який визначається переходами по модельним стимулам і станами, досяжними з початкового по таким переходам.

Спостережливість реакцій. Після подачі стимулу на автомат ми можемо бачити реакцію, що видається ним. Власне кажучи, при тестуванні ми перевіряємо якраз цю реакцію. В іншому випадку, автомат робив би якісь переходи, але ми не змогли б дізнатися, правильні вони чи ні. З іншого боку, про правильність переходів можна судити також по постстанам за умови, що вони спостережувані.

Окремо стоїть питання про спостережливості станів автомата. Якщо в будь-який момент часу ми можемо дізнатися стан автомата, прочитавши його або отримавши у відповідь на спеціальну операцію `status message` (передбачається, що операція не змінює стан), то таке тестування будемо називати тестуванням з відкритим станом. В іншому випадку, будемо говорити про тестування з прихованим станом.

Спеціальний випадок тестування відповідності (у вузькому сенсі), коли специфікація - це модельний автомат, експліцитно заданий своїм графом станів, і перевіряється еквівалентність реалізаційного (тестованого) автомата модельному. Два стану (одного або різних автоматів) еквівалентні, якщо будь-яка послідовність стимулів, допустима, починаючи з одного стану, допустима, починаючи з іншого стану, і викликає одну і ту ж послідовність реакцій. Автомати еквівалентні, якщо кожному стану одного автомата відповідає еквівалентний йому стан іншого автомата. Модельний автомат, тим самим, описує клас еквівалентних йому реалізаційних автоматів.

При наявності модельного автомата тестування з відкритим станом зводиться до обходу модельного графу, при якому кожен модельний перехід супроводжується подачею на реалізаційний автомат того ж стимулу і перевіркою того, що одержувані реалізаційні реакція і постстан такі ж, як в



модельному переході. Зауважимо, що в такій постановці завдання, оскільки модельний граф відомий, не потрібно ненадлишковості алгоритму обходу. Якщо реалізаційний стан нам не доступний (тестування з прихованим станом), доводиться вводити спеціальні обмеження на реалізацію і модель і вдаватися до набагато більш складних методів перевіряючих послідовностей (checking sequence). Обхід модельного графа в цьому випадку є недостатнім, але, очевидно, також необхідний, і також для цього не потрібно ненадлишковості алгоритму обходу.

На жаль, на практиці специфікації, по-перше, не описують явно модельний автомат і існує проблема його вираженості, а, по-друге, реалізація повинна бути еквівалентною не повному модельному автомату, а деякому його підавтомату, заздалегідь невідомому.

Найбільш широко поширений випадок - це імпліцитні специфікації у вигляді перед і постумовою. Передумова - предикат над престаном і стимулом - визначає допустимість стимулів в станах, а постусловієм - предикат над престаном, стимулом, реакцією і постстаном - визначає можливі переходи. Виразність автомата з таких специфікацій зводиться до вирішення системи рівнянь загального вигляду і, в загальному випадку, не має задовільного рішення. Але справа не тільки в цьому.

Зазвичай вважається, що специфікація описує можливі, але не обов'язкові, переходи автомата. Якщо специфікація допускає кілька переходів з даного стимулу з даного престану, то це не обов'язково означає недетермінізм реалізації. Допускається, щоб реалізація мала хоча б один з таких переходів, але не обов'язково всі; детермінована реалізація повинна мати рівно один такий перехід. Фактично, це означає, що модельному автомату відповідає не один клас еквівалентних реалізаційних автоматів, а сімейство таких класів. Реалізаційний автомат, точніше, як сказано вище, його підавтомат  $R(M) \rightarrow R$ , який визначається модельними стимулами, еквівалентний деякому підавтомату  $M(R)$  специфікаційного автомата  $M$ ,

24

причому в кожному стані  $M(R)$  допустимі всі стимули, які в цьому стані допустимі в  $M$ , але серед всіх переходів в  $M$  з даного стану з даного стимулу не всі повинні бути присутніми в  $M(R)$ . (Іноді в цьому випадку говорять про квазі-еквівалентності  $R$  і  $M(R)$  і редукції  $R$  до  $M$ . Зрозуміло, що в цьому випадку, по-перше, робота по вираженості специфікаційного автомата  $M$  може виявитися надмірною, так як нам потрібно тільки його підавтомат  $M(R)$ , число станів і переходів якого може бути в багато разів меншим, ніж в  $M$ . По-друге, сам  $M(R)$  заздалегідь невідомий, оскільки визначається не тільки  $M$ , але і  $R$ .

Можна також відзначити, що недетермінований специфікаційний автомат  $M$  може використовуватися для тестування детермінованих реалізаційних автоматів  $R$ . У цьому випадку експлікований підавтомат  $M(R)$  також детермінований. Це важливо, оскільки тестування детермінованих автоматів набагато простіше тестування недетермінованих.

Таким чином, потреба в ненадлишковому алгоритмі обходу графа станів виникає природним чином. Тестування з відкритим станом, фактично, зводиться до такого алгоритму, а для тестування з прихованим станом воно необхідне (хоча й не дуже).

### Поняття графа і алгоритму руху по графу.

Орієнтованим графом (далі просто графом)  $G$  будемо називати сукупність трьох об'єктів:  $VG$  - безліч вершин,  $XG$  - безліч стимулів,  $EG \cup VG'XG'VG$  - безліч дуг.

Стимул  $x$  допустимо в вершині  $a$ , якщо в графі існує дуга  $(a, x, b)$ . Для дуги  $(a, x, b)$  вершину  $a$  будемо називати початком дуги, вершину  $b$  - кінцем дуги, стимул  $x$  - розфарбуванням дуги. Якщо стимул дуги є несуттєвим, ми будемо також замість  $(a, x, b)$  писати просто  $(a, b)$ .

Зауваження: При тестуванні граф розглядається як граф станів автомата. Однак, при вивченні алгоритмів обходу ми не використовуємо

розмальовку дуг графа реакціями, оскільки для алгоритму досить при проході будь дуги вміти визначати кінець дуги (постстан), що починається в даній вершині (престан), і розфарбованої даними стимулом. При тестуванні з відкритим станом ми визначаємо постстан безпосередньо, а при тестуванні з прихованим станом постстан іноді можна обчислити по реакції, але ми відносимо це до способу визначення постстану, зовнішньому для алгоритму обходу, а не до самого алгоритму обходу. Граф називається кінцевим, якщо безлічі вершин і дуг кінцеві. Число вершин і дуг кінцевого графа позначимо, відповідно,  $n$  і  $k$ .

Граф називається детермінованим, якщо кінець дуги однозначно визначається її початком і допустимим в ньому стимулом: для дуг  $(a, x, b)$  і  $(a', x', b')$  з  $a = a'$  і  $x = x'$  слід  $b = b'$ . У даній статті ми будемо розглядати тільки кінцеві детерміновані графи.

Дуги  $(a, b)$  і  $(a', b')$  називаються суміжними, якщо кінець першої дуги збігається з початком другої дуги:  $b = a'$ . Маршрутом  $P$  довжини  $n$  в графі  $G$  називається послідовність  $n$  суміжних дуг: для  $i = 1..n-1$  дуга  $P[i]$  суміжно з дугою  $P[i + 1]$ . Початок  $a$  першої дуги маршруту будемо називати його початком, кінець  $b$  останньої дуги маршруту - його кінцем, сам маршрут -  $[a, b]$  -Маршрут. Порожній послідовності дуг відповідає маршрут нульової довжини, початок і кінець якого збігаються. Маршрут будемо називати обходом, якщо він містить всі дуги графа.

Алгоритмом руху по графу будемо називати алгоритм, який в процесі своєї роботи будує маршрут в графі. Формально такий алгоритм можна визначити як спеціальний вид машини з абстрактним станом (машина Гуревича, ASM - Abstract State Machine), в якому зовнішні операції частково специфіковані завданням графа, на якому відбувається робота алгоритму, і поточної вершини в ньому. Для наших цілей достатньо вказати, що алгоритму надаються дві спеціальні зовнішні операції `status ()`, яка повертає ідентифікатор поточної вершини, і `call (x)`, яка здійснює перехід з поточної

вершини  $a$  по дузі зі стимулом  $x$ . Для детермінованого графа така дуга  $(a, x, b)$  єдина (єдина вершина  $b$ ). Передумовою операції  $\text{call}(x)$  є допустимість стимулу  $x$  в поточній вершині  $a$ . Маршрут будується алгоритмом як послідовність дуг, прохідних послідовними викликами операції  $\text{call}$ . Слід зазначити, що ніяка зовнішня операція (не кажучи вже про внутрішню) не змінює сам граф, і єдиною модифікуючою операцією, яка може змінити поточну вершину, є операція  $\text{call}$ .

Ненадлишковим алгоритмом будемо називати алгоритм руху по графу, який залежить тільки від пройденої частини графа і допустимості стимулів в поточній вершині. Допустимість стимулів алгоритм може визначити за допомогою спеціальної зовнішньої операції  $\text{next}()$ , яка повертає стимул, неспеціфікованим чином обраний серед необраних раніше стимулів, допустимих в поточній вершині, здійснюючи тим самим ітерацію стимулів в вершині. Якщо всі стимули, допустимі в поточній вершині, вже вибиралися, будемо вважати, що  $\text{next}()$  повертає «порожній» символ  $\epsilon$ .

Вільним алгоритмом будемо називати ненадлишкових алгоритм, який дізнається про допустимість стимулу, ще не випробуваного в поточній вершині  $a$ , що не заздалегідь, а одночасно з проходом по дузі, розфарбованої цим стимулом. Інакше кажучи, вільний алгоритм здійснює первинний прохід по будь-якій ще не пройденій дузі з початком в поточній вершині, використовуючи поєднану зовнішню операцію  $\text{nextcall}()$ :  $x = \text{next}()$ ; if  $x \neq \epsilon$  then  $\text{call}(x)$ ; return  $x$  else return  $\epsilon$  end. Ця операція неспеціфікованим чином вибирає ще не випробуваний в поточній вершині  $a$  стимул  $x$  і проходить по дузі  $(a, x, b)$ . Якщо всі стимули вже випробувані в поточній вершині, повертається порожній символ  $\epsilon$ . Для вторинного проходу по дузі  $(a, x, b)$ , як і раніше, використовується операція  $\text{call}(x)$  в той момент, коли поточної вершиною є вершина  $a$ .

Алгоритм призначений для вирішення того чи іншого завдання; в даному розділі таким завданням є обхід графа. Нас будуть цікавити тільки

такі алгоритми, які зупиняються через кінцеве число кроків. У момент зупинки алгоритм може повідомити, виконаний обхід чи ні. Можливий також випадок, коли побудований маршрут є обходом, але алгоритм про це «не знає». Протилежний випадок, коли обхід не виконано і алгоритм «знає», що в даному графі взагалі немає обходу. Подібну інформацію, яка надається алгоритмом в момент зупинки, будемо називати вердиктом алгоритму. Ми будемо говорити, що вердикт достовірний, якщо повідомлена в ньому інформація відповідає дійсності.

Робота алгоритму, взагалі кажучи, залежить від виконання зовнішніх операцій; для ненадлишкових алгоритмів - від операції next, яка визначена неоднозначно. Будемо говорити, що алгоритм робить гарантований обхід даного графа з даної початковою вершиною, якщо це відбувається при будь-якому допустимому виконанні всіх зовнішніх операцій; для ненадлишкових алгоритмів - незалежно від ітерації стимулів в вершинах (next).

#### Досяжні графи.

Оскільки взаємна досяжність вершин - це відношення еквівалентності, в загальному випадку граф  $G$  розбивається на компоненти сильної зв'язності, на безлічі яких досяжність є відношенням часткового порядку. Компонент є підграфом графа  $G$ , безліч вершин якого - це клас еквівалентності, а дуги - всі дуги графа  $G$ , початок і кінець яких належать до цього класу. Компонент, якому належить вершина  $a$ , будемо позначати  $K(a)$ . Дугу графа  $G$ , початок і кінець якої відносяться до різних компонентів, будемо називати сполучною.

Для графа  $G$  його фактор-графом по відношенню взаємної досяжності будемо називати граф  $F(G)$ , вершинами якого є компоненти сильної зв'язності графа  $G$ , а дуга  $(A, x, B)$ , де  $A, B$  - компоненти графа  $G$ , існує тоді і тільки тоді, коли в графі  $G$  існує еднальна дуга  $(a, x, b)$ . Досяжним графом називається граф, всі вершини якого досяжні з виділеної початкової вершини. Надалі,

спеціально не обумовлюючи це, ми будемо розглядати тільки досяжні графи, оскільки тільки в них, очевидно, можливий обхід.

Граф називається ациклічним, якщо в ньому немає циклічних маршрутів;

джерелом називається вершина, до якої не входять дуги;

стоком називається вершина, з якої не виходять дуги.

Для того щоб граф  $G$  був досяжним графом з початковою вершиною  $v_0$ , необхідно і достатньо, щоб його фактор-граф  $F(G)$  був ациклічним графом з одним джерелом  $K(v_0)$ .

Необхідність. Фактор-граф  $F(G)$ , очевидно, є ациклічним. В досяжному графі  $G$  для будь-якої вершини  $v$  існує  $[v_0, v]$  -Маршрут. Залишаючи в ньому тільки сполучні дуги і замінюючи їх відповідними фактор-дугами, отримуємо  $[K(v_0), K(v)]$  - маршрут в фактор-графі, тобто,  $F(G)$  також є досяжним графом і отже, має тільки один джерело  $K(v_0)$ .

**Достатність.**

У ациклічному фактор-графі  $F(G)$  з одним джерелом  $K(v_0)$  для кожної вершини  $v$  існує  $[K(v_0), K(v)]$  - маршрут як послідовність фактор-дуг, відповідних сполучною дуг  $(a_i, b_i)$ ,  $i = 1..t$ . Позначаючи  $b_0 = v_0$  і  $a_{t+1} = v$ , отримуємо, що для  $i = 0..t$ , вершини  $b_i$  і  $a_{i+1}$  належать одному компоненту  $i$ , отже, в графі  $G$  існує  $[b_i, a_{i+1}]$  -Маршрут  $P_i$ .  $[V_0, v]$  -Маршрут в графі  $G$  будується як конкатенація  $P_0^{(a_1, b_1)} \dots P_{t-1}^{(a_t, b_t)} P_t$ . Ї

**Графи 1-го роду.**

Графом 1-го роду будемо називати граф з лінійним порядком досяжності компонентів, в якому з кожного не останнього компонента виходить тільки одна єдина дуга і вона веде в наступний компонент. За замовчуванням, початкова вершина  $v_0$  належить першому компоненту.

Іншими словами, фактор-граф такого графа складається з одного ациклічного маршруту з початком в компоненті початкової вершини  $K(v_0)$  (рис.2.1).



Рис. 2.1. Граф

Пройденим графом маршруту будемо називати підграф, що складається з дуг маршруту і інцидентних їм вершин.

**Теорема:** 1) Пройдений граф маршруту є графом 1-го роду, початок і кінець маршруту належать, відповідно, його першому і останньому компонентам.

2)  $[a, b]$  -обхід існує для і тільки для графа 1-го роду, в якому вершини  $a$  і  $b$  належать, відповідно, першому і останньому компонентам; мінімальна довжина обходу дорівнює  $O(nk)$ .

3) Для будь-яких можливих  $n$  і  $k$  існує граф 1-го роду з  $n$  вершинами і  $k^3k$  дугами, будь який обхід якого має довжину  $W(nk')$ .

4) Обхід з будь-якої початкової вершини  $v_0$  існує для і тільки для сильно-зв'язкових графів.

**Затвердження:** 1) безпосередньо впливає з того, що всі вершини пройденного графа лінійно упорядковуються маршрутом в порядку досяжності. 2) З 1) витікає існування обходу тільки для графів 1-го роду. Назад, для графа 1-го роду з єднальними дугами  $(a_i, b_i)$ ,  $i = 1..t-1$ , позначаючи  $b_0 = a$  і  $a_t = b$ , будуємо  $[b_{i-1}, a_i]$  -обхід  $P_i$   $i$ -го компонента для  $i = 1..t$  і  $[a, b]$  -обхід графа як конкатенацію  $P_1^{a_1, b_1} \dots P_{t-1}^{a_{t-1}, b_{t-1}} P_t^{a_t, b_t}$ . Довжина цього обходу не перевищує  $t-1 + \sum_{i=1}^t O(n_i k_i)$ , де  $n_i$ ,  $k_i$  - число вершин і дуг  $i$ -го компонента, тобто, дорівнює  $O(nk)$ .

**Графи 2-го роду і вільні алгоритми.**

Для маршруту в графі вершину будемо називати повністю пройденою, якщо в цьому маршруті пройдені усі, виходящі з вершини дуги.

Графом 2-го роду будемо називати такий граф 1-го роду, в якому всі компоненти, крім, можливо, останнього, складаються з однієї вершини і не містять дуг, крім однієї сполучної дуги, що веде в наступний компонент (рис.2.2).



Рис. 2.2. Приклад графу

**Теорема:** 1) Будь-який вільний алгоритм може гарантовано обійти тільки граф 2-го роду з початковою вершиною  $v_0$  з першого компонента і зупинкою в вершині з останнього компонента. 2) Існує вільний алгоритм  $A_1$ , який зупиняється на будь-якому графі, проходячи маршрут довжиною  $O(nk)$ , і гарантовано обходить всі графи 2-го роду з початковою вершиною  $v_0$  з першого компонента. За Теоремою, обхід існує тільки для графа 1-го роду. Якщо граф не 2-го роду, то деякий не останній компонент графа або складається більш, ніж з однієї вершини, або з його єдиною вершини виходить не тільки єдина дуга. У першому випадку, в силу сильної зв'язності компонента, є маршрут з початку сполучною дуги в іншу вершину компонента, отже, з початку сполучною дуги  $(a, x, b)$  виходить ще одна дуга  $(a, x', b')$ , а у другому випадку наявність таких двох дуг явно постулюється. Коли алгоритм вперше виявляється в вершині  $a$ , в ній ще жоден стимул не випробуваний, і тому вільний алгоритм повинен застосувати операцію `nexthcall`. Оскільки алгоритм повинен гарантовано обходити граф, тобто, незалежно від виконання цієї операції, припустимо, що вона вибирає стимул  $x$ . В цьому випадку ми пройдемо по сполучній дузі  $(a, x, b)$  в наступний компонент  $i$ , отже, більше не зможемо повернутися в вершину  $a$ , і дуга  $(a, x', b')$  залишиться непройденою.



2) У процесі роботи алгоритму A1 ми будемо зберігати опис пройденого графа, запам'ятовуючи всі пройдені дуги. Крім того, будемо позначати вершини, в яких операція nextcall повернула порожній символ  $\epsilon$ , тобто, вершини, які гарантовано повністю пройдені. Зауважимо, що вершина може бути повністю пройдена, але ми про це ще не знаємо і вона непомічена.

Крок алгоритму складається з наступних пунктів:

1. Виконуємо операцію nextcall, запам'ятовуючи пройдені дуги (стимул дуги повертається nextcall, а її кінець визначається за допомогою операції status), до тих пір, поки nextcall не поверне порожній символ  $\epsilon$ . В останньому випадку помічаємо поточну вершину.

2. Якщо всі пройдені вершини позначені (і, тим самим, повністю пройдені), алгоритм зупиняється.

3. В іншому випадку, в пройденому графі шукаємо шлях з поточної вершини в якусь непомічену вершину. Якщо такий шлях  $\epsilon$ , проходимо його за допомогою операцій call. Крок закінчується.

4. Якщо такого шляху немає, алгоритм зупиняється.

Зупинка алгоритму. Кожен з цих пунктів виконується за кінцевий час, оскільки в графі кінцеве число дуг і кінцеве число шляхів. За один крок з пунктів 1-3 алгоритм проходить хоча б одну непройдену дугу і / або позначає хоча б одну непомічену вершину. Тому алгоритм A1 зупиняється через кінцевий час на будь-якому графі.

Довжина маршруту. Пройдений маршрут можна уявити як конкатенацію первинних входжень дуг і з'єднуючих їх шляхів, тому довжина маршруту виходить  $O(nk)$ .

Гарантований обхід. У графі 2-го роду кожен неостанній компонент складається з однієї вершини, з якої виходить одна дуга, яка веде в наступний компонент, тому алгоритм, починаючи працювати в вершині першого компонента, очевидно, після виявиться в останньому компоненті і, отже,

зупиниться в деякій вершині  $b$  останнього компонента. Нехай в момент зупинки є непройдена дуга  $(c, d)$ , тоді її початок  $c$  непомічено і, очевидно, належить останньому компоненту. Тоді існує маршрут з  $b$  в  $c$ . Видаляючи з маршруту цикли, отримуємо шлях з поточної вершини  $b$  в непомічену вершину  $c$ . Якщо цей шлях містить тільки пройдені дуги, то алгоритм не повинен був зупинитися. Якщо ж на цьому шляху є непройдені дуги, то розглянемо початковий відрізок цього шляху до першої непройденної дуги  $(c', d')$  - вершина  $c'$  непомічена і в пройденому графі є шлях в неї з поточної вершини, отже алгоритм не повинен був зупинитися. Ми прийшли до протиріччя і, отже, в момент зупинки всі дуги пройдені і здійснений обхід. Різні алгоритми, що використовують стратегію алгоритму A1, розрізняються вибором непоміченої вершини  $v'$  з поточної поміченої вершини  $v$ . Алгоритми, засновані на обході остова графа, вибирають найдальшу від кореня ( «пошук в глибину») або саму близьку до кореня ( «пошук в ширину») вершину  $v'$ , досягну з  $v$ . «Жадний» алгоритм вибирає  $v'$  найближчу до  $v$  по довжині  $[v, v']$  -шляху.

Тепер досліджуємо питання: які достовірні вердикти може виносити вільний алгоритм обходу?

Про відсутність непройденної дуги, що виходить з даної вершини, вільний алгоритм може дізнатися, тільки отримавши порожній стимул у відповідь на операцію `nextcall`, коли ця вершина була поточною. Таку вершину назовемо поміченою, як це робиться в алгоритмі A1. Якщо в момент зупинки алгоритму відсутні непомічені вершини, алгоритм може винести достовірний вердикт «здійснений обхід» (нагадаємо, що ми розглядаємо тільки досяжні графи). Це може статися тільки в сильно-зв'язковому графі, оскільки прохід по сполучній дузі унеможливорює повернення в її початкову вершину  $v_0$  і, отже, ця вершина залишиться непоміченою. Тому, якщо всі пройдені вершини позначені, достовірним буде навіть сильніший вердикт «здійснений гарантований обхід». Алгоритм A1 може робити це. Якщо ж в

момент зупинки залишилися непомічені вершини, алгоритм, аналізуючи пройдений граф, може винести один з двох достовірних вердиктів: якщо пройдений граф 2-го роду - «невідомо, здійснений чи обхід, але якщо здійснений, то обхід гарантований »; в іншому випадку - «невідомо, здійснений чи обхід, але якщо здійснений, то обхід негарантований».

#### **Предикат сполучних дуг.**

Алгоритм може гарантовано обходити графи 1-го роду, якщо йому якимось зовнішнім чином поставляється інформація про сполучні дуги. Нехай в кожній вершині графа заданий предикат від стимулу  $p(x)$ , який будемо називати предикатом сполучних дуг. Предикат достовірний, якщо він правдивий на і тільки на стимулах сполучних дуг. Алгоритм, з зовнішніми операціями `next`, `call` і `p`, звичайно, не є ненадлишковим, але в певному сенсі «мінімально надмірним» алгоритмом.

**Теорема:** Існує алгоритм А3 з предикатом сполучних дуг  $p$ , який зупиняється на будь-якому графі, проходячи маршрут довжиною  $O(nk)$ , і гарантовано обходить графи 2-го роду і графи 1-го роду з достовірним предикатом.

Алгоритм А3 відрізняється від алгоритму А2 тим, що спочатку проходяться тільки незв'язні дуги, що виходять з пройдених вершин, а стимули сполучних дуг запам'ятовуються в їх початкових вершинах. Коли незв'язних непройдених дуг більше не залишається, алгоритм шукає в пройденому графі шлях в початок однієї з запам'ятованої непройдених сполучних дуг і, якщо знаходить такий шлях, проходить його і сполучну дугу, після чого знову шукає непройдені незв'язні дуги. Є дуга сполучною чи ні, визначається по предикату  $p$ . Доказ тверджень теореми про зупинку алгоритму, довжині маршруту і обході очевидно. Легко показати, що алгоритм А3 після зупинки може проаналізувати пройдений граф і видати один з наступних достовірних вердиктів:

1. «Здійснено обхід», якщо з пройдених вершин не виходять непройдені дуги.

а) «Обхід гарантований і предикат достовірний», якщо предикат в пройденому графі 1-го роду правдивий на і тільки на його сполучних дугах.

б) «Обхід гарантований, але предикат недостовірний», якщо предикат в пройденому графі 1-го роду правдивий на деяких дугах останнього компонента і / або хибна на деяких сполучних дугах, з початку яких не виходять інші дуги.

с) «Обхід негарантований і предикат недостовірний» - у всіх інших випадках.

2. «Обхід не досконалий», якщо хоча б з однієї пройденої вершини виходить не проторена дуга.

а) «Або предикат достовірний і тоді вихідний граф НЕ 1-го роду, або предикат недостовірний і тоді невідомо, чи є вихідний граф графом 1-го роду», якщо предикат правдивий на і тільки на сполучних дугах і непройдених дугах.

б) «Предикат НЕ достовірний і невідомо, чи є вихідний граф графом 1-го роду» - у всіх інших випадках.

Предикат  $r$  формально визначений на трійках (граф, вершина, стимул). Будемо називати предикат ненадлишковим, якщо він не залежить від графа. Більш точно, залежність предиката від графа зводиться до залежності від безлічі стимулів, допустимих в вершині, тобто, формально предикат визначено на трійках (вершина, безліч допустимих в вершині стимулів, стимул). Якщо ненадлишковий предикат розглядати не як зовнішню, а як внутрішню операцію алгоритму, то відповідним чином модифікований алгоритм  $A_3$  (позначимо його  $A_4$ ), по-перше, буде ненадлишковим, і по-друге, гарантовано обходить всі графи 2-го роду і ті графи 1-го роду, на яких предикат достовірний.

Разом з тим, ненадлишковий предикат, очевидно, не може бути достовірним на всіх графах з виділеними початковими вершинами  $v_0$ , ізоморфних з точністю до розмальовки дуг стимулами, якщо це не графи 2-го роду.

**Теорема:** Не існує ненадлишкового алгоритму, який обходив би деякий граф  $G$  1-го (але не 2-го) роду і всі графи, що відрізняються від  $G$  тільки забарвленням дуг стимулами. Якщо граф  $G$  не 2-го роду, то в ньому є єднальна дуга  $(a, x, b)$ , з початку якої виходить ще одна дуга  $(a, x', b')$ . Розглянемо момент часу, коли алгоритм здійснює первинний прохід по другій дузі - в вершині  $a$  викликається операція  $\text{call}(x')$ . Якщо алгоритм обходить граф, то в цей момент перша дуга  $(a, x, b)$  ще не пройдена. Якщо алгоритм надлишковий, то інформація про графа, якою він володіє в цей момент, однакова як для графа  $G$ , так і для графа  $G'$ , що відрізняється від  $G$  зміною місцями стимулів розмальовки цих двох дуг  $(a, x', b)$  і  $(a, x, b')$ . Тому в графі  $G'$  алгоритм піде по дузі  $(a, x', b)$  в той момент, коли дуга  $(a, x, b')$  ще не пройдена. Однак, в цьому графі сполучною є дуга  $(a, x', b)$ , тобто, обхід здійснений не буде.

#### Пошук в ширину.

Починаючи зі стартового вузла, цей алгоритм спочатку визначає все безпосередньо сусідні вузли, потім все вузли в двох кроках, потім в трьох, і так далі, поки мета не досягнута. Типовим є те, що для кожного вузла його неперевірені сусіди поміщаються в список Open, який зазвичай є FIFO чергою. Можна помітити, що він знаходить шлях навколо перешкод, і цей шлях є найкоротшим, тобто одним з декількох найкоротших в довжину шляхів, якщо всі кроки мають однакову вартість. Тут є безліч простих проблем. Одна полягає в тому, що пошук йде рівномірно у всіх напрямках, замість того, щоб бути спрямованим в бік цілі. Інша проблема в тому, що не всі кроки рівні, принаймні, кроки по діагоналі повинні бути довгими ортогональних. Це покращує простий пошук в ширину тим, що запускаються

36

два одночасних пошуку в ширину з стартового і кінцевого вузлів і зупиняється, коли вузол з одного фронту пошуку знаходить сусідній вузол з іншого фронту. Це може поліпшити простий пошук в ширину (зазвичай в 2 рази), але все ще є дуже неефективним. Хитрощі, на зразок цієї, добре запам'ятати, так як вони можуть стати в нагоді в подальшому.

### **Алгоритм Дійкстра.**

Є.Дійкстра розробив класичний алгоритм для проходу по графам, грані яких мають різну вагу. На кожному кроці, він **шукає необроблені вузли близькі до стартового, потім переглядає сусідів знайденого вузла, і встановлює або оновлює їх відповідні відстані від старту.** Цей алгоритм має дві переваги в порівнянні з пошуком в ширину: він бере до уваги вартість або довжину шляху і оновлює вузли, якщо до них знайдений кращий шлях. Для реалізації, список Open з чергою FIFO замінюється пріоритетною чергою, де витягнутий вузол має краще значення - тут, це найменша вартість шляху від старту. Показана хороша адаптація алгоритму до вартості місцевості. Однак, він має слабкість пошуку в ширину, ігноруючи напрямок до мети.

### **Пошук в глибину.**

Цей пошук протилежний пошуку в ширину. Замість відвідин спочатку всіх сусідів, а потім їх спадкоємців, він спочатку відвідує всіх спадкоємців, а тільки потім перемикається на сусідів. Для впевненості в тому, що пошук закінчиться, необхідно передбачити зупинку на певній глибині. Можна використовувати той же самий код, що і в пошуку в ширину, якщо додати параметр для відстеження глибини кожного вузла і замінити Open з черги FIFO на стек LIFO (last-in-first-out). Насправді можна повністю позбутися від списку Open і замість цього зробити пошук рекурсивною підпрограмою, що зменшить витрату пам'яті використаної під Open. Необхідно, щоб кожна клітинка маркувалася як "відвідана" при просуванні в глиб і ця позначка знімалася на зворотному ходу, щоб уникнути генерації шляхів, які відвідують

двічі одну й ту ж комірку. Можна помітити, що цього недостатньо, алгоритм все одно може плутатися навколо себе і витратити час на безглузді шляхи. Для геометричного пошуку шляху можна зробити два доповнення. Перше полягатиме в додаванні мітки на кожному клітинку з довжиною знайденого до неї найкоротшого шляху; алгоритм більше не відвідає цей осередок поки не буде мати до неї шлях з меншою вартістю. Інша полягає у виборі спочатку сусідів, які ближче до мети. З цими двома доповненнями, можна помітити, що пошук в глибину швидко знайде шлях. Можуть оброблятися навіть зважені шляхи, якщо зробити зупинку по загальній накопиченій вартості замість загальної відстані.

Насправді в алгоритмі пошуку в глибину існує ще одна проблема - вибір правильної глибини зупинки. Якщо вона буде дуже маленькою, то шлях не буде знайдений; якщо занадто великий, то потенційно можна витратити багато часу даремно, досліджуючи занадто глибоко, або знайти шлях з дуже високою вартістю. Ці проблеми вирішуються ітеративним поглибленням - техніка, при якій виконується пошук в глибину з збільшується глибиною до тих пір, поки шлях не буде знайдений. При пошуку шляху ми можемо не починати з глибини що дорівнює одиниці, а відразу почати з глибини рівній відстані по прямій від старту до мети. Цей пошук є асимптотично оптимальним серед усіх переборних алгоритмів по часу і пам'яті.

### Алгоритм "кращий-перший".

Це перший розглянутий евристичний пошук, який бере до уваги знання про простір пошуку для направлення своїх зусиль. Він схожий на алгоритм Дійкстра, за винятком того, що вузли в списку Open оцінюються за приблизним залишком відстані до цілі. Ця оцінка також не вимагає наявності оновлень, на відміну від алгоритму Дійкстра. Показує роботу алгоритму. Це найшвидший з усіх плануючих алгоритмів розглянутих раніше, який прямує по прямій до мети. Він так само має і свої слабкості. Він не бере до уваги накопичену вартість шляху, прямуючи по прямій через зону з високою

38

вартістю, а не обходячи її. Можна побачити, що знайдений шлях навколо перешкоди не прямий, а згинається навколо перешкоди на манер алгоритму трасування, розглянутого раніше.

### **Тестування на основі ненадлишкових алгоритмів обходу.**

Запропоновані ненадлишкових алгоритми обходу детермінованих орієнтованих графів можуть служити основою тестування з відкритим станом.

Ми припускаємо виконаною таку гіпотезу про допустимість для тестування з відкритим станом частково певних автоматів: для кожного стану, досяжного за моделлю з початкового стану, всі стимули, допустимі в моделі, допустимі і в реалізації (зворотне не потрібно).

Тестування з прихованим станом - набагато складніше завдання. Перш за все, виникає проблема допустимості стимулів для частково певних автоматів. Якщо ми не знаємо, в якому стані знаходиться тестований автомат, то ми не знаємо, які стимули в ньому допустимі, а які ні. Якщо не робити ніяких припущень про реалізацію, то ми можемо подавати тільки такі стимули, які допустимі в усіх станах. Це одна з причин, за якої часто розглядаються тільки повністю певні автомати. Специфікація автомата, взагалі кажучи, може визначати кілька специфікаційних переходів з даного престану з даного стимулу з отриманням однієї і тієї ж реакції; такі переходи розрізняються тільки своїм постстаном. Іншими словами, рівняння постумови  $POST(v, x, y, v') = true$  може мати більше одного рішення щодо постстану  $v'$ . Якщо для будь-яких можливих  $v, x, y$  таких рішень не більше одного, то будемо говорити, що специфікація і описуваний нею модельний автомат слабо-детерміновані. Це називається спостережуваним недетермінізмом.

Слід зазначити, що слабкий детермінізм не применшує описову потужність специфікацій з точністю до еквівалентності специфікованих



автоматів. Справа в тому, що класу еквівалентних кінцевих автоматів відповідає регулярна множина послідовностей в алфавіті стимулів і реакцій (в декартовому творі їх алфавітів), яке, по відомій теоремі про регулярні множини, може бути породжене детермінованим кінцевим графом. Детермінізм тут розуміється як відсутність двох однаково розфарбованих (стимулом і реакцією) дуг, що виходять з однієї вершини, що еквівалентно слабкому детермінізму в нашому визначенні. Інша справа, що специфікації без обмеження слабо-детермінованості часто писати простіше, а визначення еквівалентних слабо-детермінованих специфікацій може бути важкою справою.

Для тестування з прихованим станом частково певних автоматів зі слабо-детермінованої специфікацією, приймемо таку гіпотезу про допустимість: для даних престану, допустимого в ньому стимулу і реакції будь-який стимул, допустимий в постстані специфікаційного переходу, припустимо і в постстані реалізаційної переходу. Якщо через  $x(v)$  позначити безліч стимулів, допустимих в стані  $v$ , то наша гіпотеза означає вкладеність  $x(v') \subseteq x(vR)$ , де  $vR$  - постстан реалізаційної переходу  $(v, x, y, vR)$ , а  $v'$  - рішення рівняння постумови  $POST(v, x, y, v') = \text{true}$ . Якщо рівняння не має рішень, то це означає неправильність реакції  $y$ , і на цьому тестування припиняється.

Гіпотеза про допустимість, що здається на перший погляд невмотивованою, на практиці виявляється цілком природною. Вона означає, що допустимість стимулів в кожен момент часу однозначно визначається історією (послідовністю подаваних стимулів і одержуваних реакцій), що цілком природно при роботі користувача з програмною системою, що моделюється автоматом. Природно, передбачається, що помилки, можливі у реалізації, можуть бути виявлені (по одержуваним реакціям) до того, як вони призведуть до порушення «гіпотези про допустимість».

Будемо вважати, що початковий стан  $v_0$  відомо. Подавши на нього стимул  $x_0$  і отримавши реакцію  $y_0$ , ми перевіряємо правильність реакції і, якщо вона правильна, обчислюємо постстан  $v_1$ . Тоді, згідно з «гіпотезі про допустимість», реалізація знаходиться в постстані і, в якому допустимі всі стимули, допустимі в  $v_1$ . Виходячи з цього, на наступному кроці вибираємо стимул, допустимий в  $v_1$ , і так далі. У процесі такого тестування будемо будувати гіпотетичний граф і маршрут в ньому, проводячи дуги, відповідні гіпотетичним переходам автомата. Якщо якась реакція відкидається, то тест фіксує помилку, і тестування закінчується.

Важливо підкреслити наступне: якщо реалізаційний автомат детермінований, то побудований гіпотетичний граф також детермінований. Почасти це дає можливість контролювати допущення про детермінізм реалізаційної графа. Більш важливим є те, що якщо пройдений маршрут є обходом побудованого графа, то ми можемо вважати побудований граф графом станів експлікований підавтомат  $M(R)$  модельного автомата  $M$ . реалізаційна автомату  $R$  (в якому не враховуються «зайві» стимули, допустимі в реалізації, але неприпустимі в специфікації) задовольняє специфікації тоді і тільки тоді, коли він еквівалентний підавтомату  $M(R)$ . Після цього можна перевіряти цю еквівалентність, використовуючи звичайні методи тестування відповідності з допомогою перевіряючих послідовностей, розглядаючи граф  $M(R)$  в якості модельного графа.

Таким чином, ми бачимо, що ненадлишкові алгоритми обходу можуть використовуватися в якості базових у першій фазі тестування з прихованим станом, яку можна назвати фазою вираженість моделі. Умовою є детермінізм реалізації, слабкий детермінізм специфікації і можливість визначення правильності реакції і обчислення постстану для правильної реакції, а також «гіпотеза про допустимість». Друга фаза тестування - це звичайне тестування відповідності автоматів з використанням експлікованого графа в якості моделі.

Недетермінованої (зокрема, слабо-детермінованої) специфікації, звичайно, може відповідати недетермінована реалізація. Часто при тестуванні вдаються до факторизації специфікації, вводячи еквівалентність переходів. Найближчою метою такої факторизації є зменшення необхідного числа тестових впливів як наслідок зменшення числа станів і переходів в факторизованому модельному автоматі. При такому тестуванні потрібно перевіряти тільки фактор-переходи, для чого може бути обраний будь-який перехід з відповідного класу еквівалентності. Зрозуміло, що якщо задану еквівалентність «роздібнити», то тестування з фактор-моделі з більш «дробовою» еквівалентністю дасть не найгірше тестове покриття. Існують методи такого подальшого «дроблення» заданої еквівалентності, які в деяких випадках призводять до детермінованості фактор-моделі, залишаючи її все ще досить «малою» у порівнянні з вихідною нефакторизованою моделлю. Таким чином, наші ненадлишкові алгоритми обходу детермінованих графів можуть застосовуватися також при факторизованому тестуванні недетермінованих реалізацій.

Як правило, при тестуванні критичним є не стільки складність за часом і пам'яті алгоритму обходу, скільки число тестових впливів, тобто, довжина обходу. Вільний алгоритм A5 (і його ненадлишкова версія A6) забезпечують лише мінімальний порядок  $n_k$  довжини обходу в найгіршому випадку. У той же час на багатьох графах з мінімальною довжиною обходу по порядку меншою  $n_k$ , вони можуть будувати обхід настільки ж довгий, як і в найгіршому випадку. Дослідження ненадлишкових алгоритмів, які прагнуть зробити обхід мінімальної довжини, останнім часом активно розвивається.

### 2.3. Висновки до розділу

Для розробки програми моделювання процесів комп'ютерної логіки були досліджені принципи двійкової математики зокрема в мові програмування Object Pascal. Так само проаналізована теорія графів. Весь цей матеріал є необхідним для розробки програми моделювання процесів

комп'ютерної логіки. З теоретичного матеріалу, який був проаналізований, були виділені основні принципи побудови логіки роботи програми. У додатку були задіяні алгоритми обходу графа в ширину і глибину, а також використані логічні функції мови Object Pascal.

### РОЗДІЛ 3

## РОЗРОБКА ПРОГРАМИ ДЛЯ МОДЕЛЮВАННЯ ПРОЦЕСІВ КОМП'ЮТЕРНОЇ ЛОГІКИ

### 3.1 Розробка і призначення елементів інтерфейсу додатку

З аналізу наведеного в таблиці 1.1 були визначені наступні вимоги до інтерфейсу програми:

- Програма повинна мати доступний інтерфейс, враховувати всі можливі критерії та умови, що виникають в процесі роботи.
- Програма повинна мати максимально простий інтерфейс для простоти використання.
- Інтерфейс програми повинен бути придатним для виконання лабораторних робіт.

На рис.3.1 представлено вікно програми моделювання логічних схем.

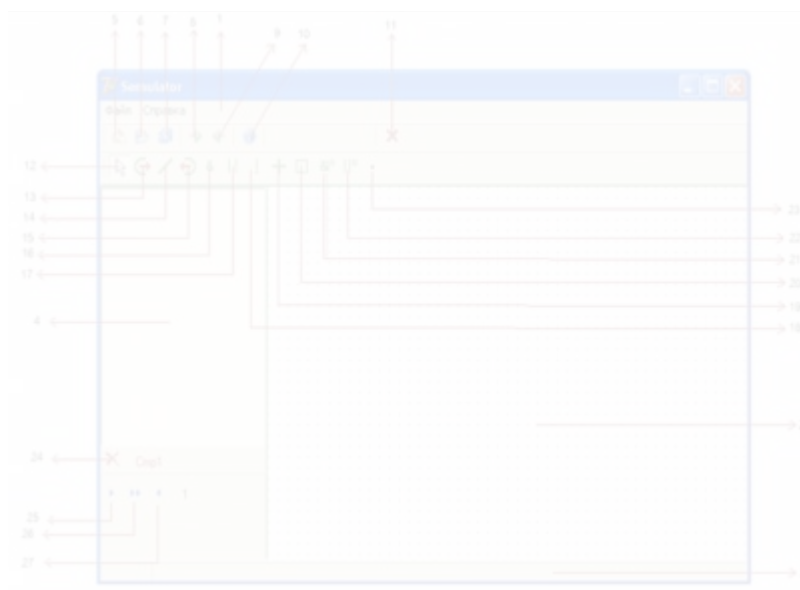


Рис.3.1 Опис і призначення елементів інтерфейсу додатку

1) *Головне меню*: реалізує основні дії роботи з документом

Також можуть бути доступними найбільш важливі пункти для роботи з документом схеми.

Пункт **Файл** містить наступні операції:

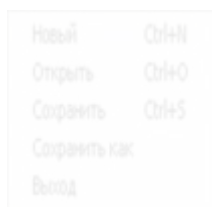


Рис. 3.2. Пункт меню **Файл**

а) Операція **Новий**: Операцію також можна викликати одночасним натисканням клавіш **CTRL+N**. Дана операція призначена для закриття поточної схеми і створення нової. При цьому створюється нова робоча область, яка може використовуватися для створення схеми. Якщо перед цим ви проробили якісь зміни поточної схеми, вам буде запропоновано зберегти поточну схему перед її закриттям. При запуску програми операція виконується автоматично.

б) Операція **Відкрити**: служить для відкриття створеного документа схеми. Операцію також можна викликати одночасним натисканням клавіш **CTRL+O**. Операція призначена для відкриття вже існуючого файлу схеми. Відображає стандартне діалогове вікно відкриття файлу, в якому необхідно вибрати диск і каталог, що містить файл схеми, який ви хочете відкрити.

с) Операція **Зберегти**: служить для збереження створеного документа. Операцію також можна викликати одночасним натисканням клавіш **CTRL+S**. Зберігає поточний файл схеми. Відображається стандартне діалогове вікно збереження файлу, в якому необхідно вибрати диск і каталог, де ви хочете зберегти схему і назву файлу.

д) Операція **Зберегти як**: команда аналогічна операції **Save**, але зберігає поточну схему з новим ім'ям файлу, залишаючи первісну схему незмінної.

Використовуйте цю команду, щоб безпечно експериментувати на копії схеми, без зміни оригіналу.

е) Операція **Вихід**: операцію також можна викликати одночасним натисканням клавіш ALT+F4. Операція призначена для завершення роботи програми. Якщо ви не зберегли зміни в схемі, то буде зроблений запит на збереження.

Пункт **Довідка** містить пункти меню для виклику файлу довідки.

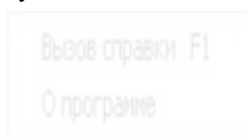


Рис. 3.3. Пункт меню **Довідка**

Операція *Виклик довідки*: Меню **Довідка** надає виклик файлу-довідки. Виклик довідки також можна здійснити натисканням клавіші F1

2) *Робоча область*: область на якій розміщуються всі необхідні компоненти (елементи логіки, а також допоміжні компоненти). Це основна частина програми для розміщення та аналізу компонентів, які утворюють схеми.

3) *Рядок стану*: показує дії в реальному часі роботи програми, відображає дії які виконує додаток.

4) *Список входів і виходів*: служить для відображення входів і виходів додатків, які знаходяться в робочій області програми.

5) *Новий документ*: служить для створення нового документа (аналогічно пункту меню).

6) **Відкрити**: служить для відкриття створеного документа схеми.

7) **Зберегти**: служить для збереження створеного документа схеми.

8) **Запуск**: кнопка, яка запускає програму на виконання (включає механізм обробки компонентів робочої області).

9) **Скидання**: кнопка зупинки програми (вимикає механізм обробки компонентів робочої області).

10) **Довідка**: кнопка для запуску довідки.

11) **Очистити**: кнопка призначена для очищення робочої області програми (даний механізм використовується при відкритті документа, а також при створенні нового).

12) **Стрілка**: кнопка для роботи з компонентами програми. Призначена для реалізації основних дій роботи з компонентами (виділення, переміщення).

13) **Вхід**: кнопка, що служить для розміщення на робочій області входу для схем.

14) **Лінія**: кнопка призначена для малювання з'єднувальних ліній.

15) **Вихід**: кнопка, що служить для розміщення на робочій області виходу для схем.

16) **I**: кнопка для розміщення в робочій області компонента "I".

Операція I має результат "істина" тільки в тому випадку, якщо обидва її операнда істинні.

17) **Або**: кнопка для розміщення в робочій області компонента "Або".

Операція АБО дає "істину", якщо значення "істина" має хоча б одні з операндів. Зрозуміло, у разі, коли справедливі обидва аргументи одночасно, результат як і раніше істинний. Нижче наведена таблиця істинності для компонентів I та АБО.

Таблиця 3.1.

**Таблиця істинності для компонентів I та АБО**

X	Y	X and Y	X or Y	X xor Y
0	0	0	0	0

47



0	1	01	1
1	0	0	1
1	1	1	0

18) *Чи не*: кнопка для розміщення в робочій області компонента "Не".

Найпростішою логічною операцією є операція НЕ, по-іншому її часто називають запереченням, доповненням або інверсією і позначають NOT\_X. Результат заперечення завжди протилежний значенню аргументу. Логічна операція НЕ є унарною, тобто має всього один операнд. На відміну від неї, операції І (AND) і АБО (OR) є бінарними, тому що являють собою результати дій над двома логічними величинами.

Таблиця 3.2.

#### Логічна операція НЕ

X	Not X
0	1
1	0

19) *Перетин*: кнопка служить для розміщення перетину на перпендикулярних з'єднувачах.

20) *Ластик*: кнопка включає режим активності гумки для стирання з'єднувачів.

21) *І-ні*: кнопка служить для розміщення компонента "І-ні".

22) *Чи-ні*: кнопка служить для розміщення компонента "Або-ні".

23) *Клема*: кнопка служить для розміщення компонента, який позначає приєднання одного з'єднувача до іншого.

24) **Видалити:** кнопка для видалення виділеного об'єкта робочої області.

25) **Лічильник:** кнопка включає лічильник, який призначений для автоматичної генерації всіх варіантів значень входів будь-якої схеми.

26) **Запуск:** кнопка альтернативного запуску програми (див 8).

27) **Скидання:** кнопка призводить лічильник в початковий стан.

### **3.2 Етапи розробки програми**

#### **Етап планування**

На цьому етапі було визначено завдання, які має вирішувати додаток. Додаток має мати набір стандартних елементів логіки ( «І», «АБО», «НЕ»), які попередньо були спроектовані.

#### **Етап проектування**

1. Дослідження предметної області.

2. Вибір мови програмування для реалізації проекту. Як така мова був обраний Object Pascal так як Object Pascal - це об'єктивно-орієнтована мова програмування, який і є основою Delphi. Він відноситься до мов високого рівня. Його родоначальник - мова Pascal. Delphi - це візуальне середовище програмування (Rapid Application Development - RAD), яка включає в себе крім компілятора ще й редактор, і засоби налагодження, і великі набори готових програм (так звані бібліотеки), наприклад набори математичних функцій. RAD-системи дозволяють також швидко будувати прототипи майбутніх програм: проектувати форму вікон, розміщувати в них всілякі елементи управління (кнопки, списки, перемикачі). За допомогою Delphi можна розробити додаток для Windows практично необмеженої складності. Така розробка дещо нагадує складання програм з кубиків: в Delphi є близько двох сотень готових компонентів, які розміщуються в майбутньому вікні одним клацанням миші. Це значно полегшує життя програмістам: не

потрібно витратити багато часу на створення інтерфейсу програми, а приділити увагу головному - функціонуванню. Але за таку зручність доводиться платити: мінімальні розмір програми створеної за допомогою RAD становить близько 286 КБ, а без використання візуального 17 КБ. На сьогоднішній день робота в Delphi є програмування - продуктивним методом створення додатків для Windows.

#### Етап кодування

На даному етапі були визначені:

1. Процедури і функції необхідні для побудови інтерфейсу додатку.
2. Процедури, які призводять додаток в режим обчислення значень вихідних значень, попередньо побудованих логічних схем.
3. Основний дії по роботі з файлами схем (збереження, відкриття, закриття).

#### Етап тестування і налагодження

В процесі розробки програми постійно змінювалася зовнішня структура програми, тобто не зраджуючи принципів роботи програми, вдосконалювався зовнішній зміст форми, для більш зручного користування програмою. Часто зустрічалися помилки в роботі програми, але вони були усунені.

### 3.3 Реалізація логічної схеми «Суматор» в розробленому середовищі проектування логічних схем

#### Теоретична частина

Побудова довірчих суматорів зазвичай починається з суматора по модулю 2. На рис.3.4 наведена таблиця істинності цього суматора.

X	Y	Out
0	0	0
0	1	1
1	0	1
1	1	0

50

Рис. 3.4. Таблиця істинності суматора по модулю 2

Відповідно до принципів побудови довільної таблиці істинності отримаємо схему суматора по модулю 2. Ця схема приведена на рис.3.5.



Рис. 3.5. Принципова схема, яка реалізує таблицю істинності суматора по модулю 2

Суматор за модулем 2 (схема виключає "АБО") зображується на схемах як показано на рис.3.6.

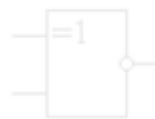


Рис. 3.6. Зображення схеми, яка виконує логічну функцію виключає "АБО"

Суматор за модулем 2 виконує підсумовування без урахування перенесення. У звичайному двійковому суматорі потрібно враховувати перенесення, тому потрібні схеми, що дозволяють формувати перенесення в наступний двійковий розряд. Таблиця істинності такої схеми, званої напівсуматор приведена на рис.3.7.

A	B	S	PO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Рис. 3.7. Таблиця істинності напівсуматора

Відповідно до принципів побудови довільної таблиці істинності отримаємо схему напівсуматора. Ця схема приведена на рис.3.8.

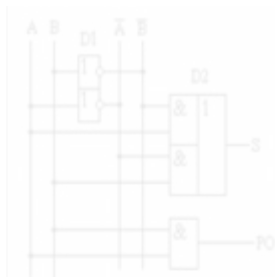


Рис. 3.8. Принципова схема, яка реалізує таблицю істинності напівсуматора

Напівсуматор зображується на схемах як показано на рис.3.9.



Рис. 3.9. Зображення напівсуматора на схемах

Схема напівсуматора формує перенесення в наступний розряд але не може враховувати перенесення з попереднього розряду, тому вона і називається напівсуматор. Таблиця істинності повного двійкового одноразрядного суматора наведено на рис.3.10.

PI	A	B	S	PO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Рисунок 3.10. Таблиця істинності повного двійкового одноразрядного суматора

Відповідно до принципів побудови схеми по довільній таблиці істинності отримаємо схему повного двійкового одноразрядного суматора. Ця схема приведена на рис.3.11.

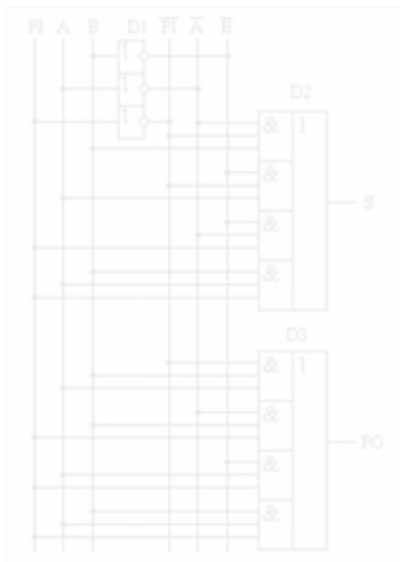


Рис. 3.11. Принципова схема, яка реалізує таблицю істинності повного двійкового одноразрядного суматора

Повний двійковий одноразрядний суматор зображується на схемах як показано на рис.3.12.



Рис. 3.12. Зображення повного двійкового одноразрядного суматора на схемах

Для того, щоб отримати багаторозрядний суматор, необхідно з'єднати входи і виходи переносів відповідних двійкових розрядів. Схема з'єднання наведена на рис.3.13.

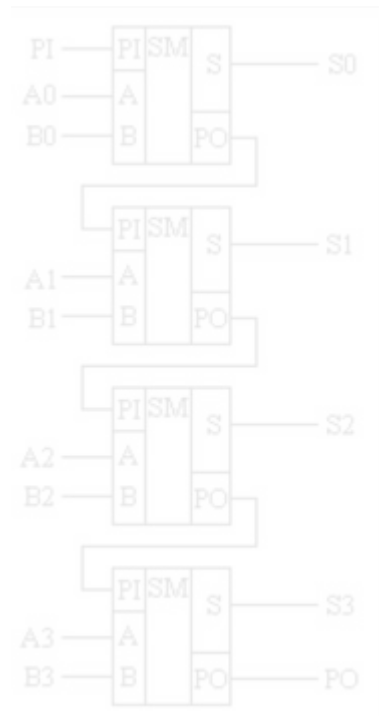


Рис. 3.13. Принципова схема многоразрядного довічного суматора

Повний двійковий багаторозрядний суматор зображується на схемах як показано на рис.3.14.

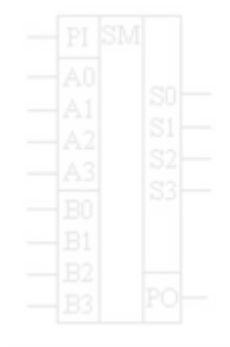


Рис. 3.14. Зображення повного двійкового многоразрядного суматора на схемах

### Практична частина

Побудова повного двійкового одноразрядного суматора в середовищі проектування логічних схем.

Для виконання даного завдання необхідно:

1. Ознайомитися з вище викладеною теоретичною частиною.
2. Визначити необхідні компоненти, які повинні бути розміщені в головному вікні програми.
3. З'єднати ці компоненти відповідно до правил побудови даної схеми.
4. Протестувати дану схему з метою посвідчення правильності її побудови.

На рисунку 3.15 представлені схема реалізації повного двійкового одноразрядного суматора.



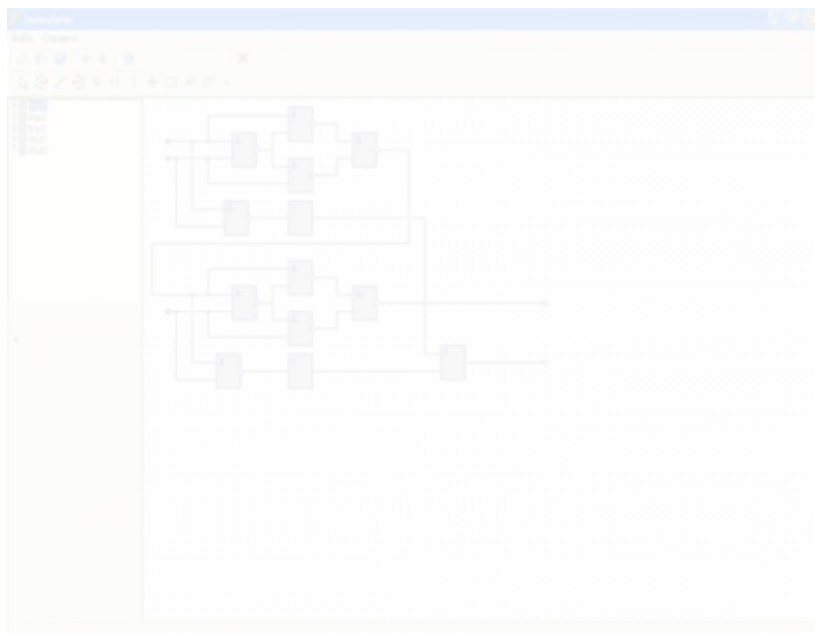


Рис. 3.15 Повний двійковий однорозрядний суматор

### 3.4. Висновки до розділу 3

В результаті проведеної роботи по розробці програми для моделювання процесів комп'ютерної логіки, програма набула доступний інтерфейс, враховує всі можливі критерії та умови, що виникають в процесі роботи. Програма має максимально простий інтерфейс для простоти використання, займає малий обсяг дискового простору і оперативної пам'яті. Додаток став придатним середовищем для виконання лабораторних робіт.

Розроблений додаток відповідає всім поставленим вимогам, які були розроблені в результаті аналізу вимог до створення програмного забезпечення та особливостей побудови інтерфейсу.

### ВИСНОВКИ

У магістерській роботі було розглянуто розробка програмного забезпечення для моделювання процесів комп'ютерної логіки.

Одним з основних методів викладання є наочне моделювання на ЕОМ різних явищ і процесів, що мають математичні описи. Необхідність в такому моделюванні обумовлена тим, що для ряду експериментів важко або неможливо провести експеримент в природних умовах (великі фінансові, тимчасові та інші витрати і т.д.). У міру того як комп'ютер буде грати все більшу роль в розумінні і вивченні явищ, візуальне уявлення складних експериментальних результатів набуватиме все більшої важливості. Крім того, комп'ютерне моделювання дозволяє здійснювати експерименти з неіснуючими реально матеріалами, параметрами установки, конструкціями.

У першому розділі проаналізовані програми моделювання та аналізу цифрових і аналогових пристроїв на основі цього розроблено вимоги до створеного додатку, які були реалізовані:

- Програма має доступний інтерфейс, враховує всі можливі критерії та умови, що виникають в процесі роботи.
- Програма має максимально простий інтерфейс для простоти використання.
- Займає малий обсяг дискового простору і оперативної пам'яті.
- Програма придатна для виконання лабораторних робіт.

Всі вище перераховані вимоги реалізовані в розробленому додатку. Додаток здатний виявляти недоліки в змодельованих дослідах і програмно усувати їх. Вирішує завдання не тільки для розрахунків, але і для якісного моделювання досліджуваного явища і спостереження за ним.

У другому розділі проведений аналіз теоретичних основ побудови програми моделювання, який включає аналіз положень двійкової математики та аналіз положень теорії графів. На даному етапі розробки програми виділені принципи побудови логіки додатка.

У третьому розділі розглядаються етапи розробки програми, описані елементи інтерфейсу програми, також приведена реалізація логічної схеми «Суматор» в розробленому середовищі проектування логічних схем.

Результатом виконання магістерської роботи є розроблений додаток для моделювання процесів комп'ютерної логіки, який може бути придатним для виконання лабораторних робіт.

## Matches

Internet sources

73

1	<a href="http://delphikingdom.com/asp/viewitem.asp?catalogid=838">http://delphikingdom.com/asp/viewitem.asp?catalogid=838</a>	20 Sources	1.89%
2	<a href="https://skachatvs.com/2000005608/referat-elektronik-workbench-5-0-opis">https://skachatvs.com/2000005608/referat-elektronik-workbench-5-0-opis</a>	2 Sources	1.77%
3	<a href="http://www.delphikingdom.com/asp/articles_forum.asp?ArticleID=838">http://www.delphikingdom.com/asp/articles_forum.asp?ArticleID=838</a>		1.58%
4	<a href="https://jak.koshachek.com/articles/dvijkovi-sumatori.html">https://jak.koshachek.com/articles/dvijkovi-sumatori.html</a>		1.24%
5	<a href="https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf">https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf</a>		0.92%
6	<a href="https://lubbook.org/book_746_glava_2_Tema_2_ZHitt%D1%94vijj_%D1%81ikl_prog.html">https://lubbook.org/book_746_glava_2_Tema_2_ZHitt%D1%94vijj_%D1%81ikl_prog.html</a>		0.92%
7	<a href="https://k-dom.com.ua/shho-take-sumator-i-napivsumator">https://k-dom.com.ua/shho-take-sumator-i-napivsumator</a>		0.75%
8	<a href="https://www.yumpu.com/de/document/view/18851930/kompetenz-von-verhaltenstherapeuten-in-der-ausbildung">https://www.yumpu.com/de/document/view/18851930/kompetenz-von-verhaltenstherapeuten-in-der-ausbildung</a>	41 Sources	0.51%
9	<a href="http://um.co.ua/4/4-9/4-9581.html">http://um.co.ua/4/4-9/4-9581.html</a>		0.51%
10	<a href="https://www.ispras.ru/publications/neizbytochnye_algoritmy_obkhoda_grafov_determinirovanny_sluchay.pdf">https://www.ispras.ru/publications/neizbytochnye_algoritmy_obkhoda_grafov_determinirovanny_sluchay.pdf</a>		0.46%
11	<a href="https://old-zdia.znu.edu.ua/gazeta/konffacuv16_68.pdf">https://old-zdia.znu.edu.ua/gazeta/konffacuv16_68.pdf</a>		0.18%
12	<a href="https://revolution.allbest.ru/mathematics/00510853_0.html">https://revolution.allbest.ru/mathematics/00510853_0.html</a>		0.16%
13	<a href="http://tesis.ucsm.edu.pe/repositorio/bitstream/handle/UCSM/10007/70.2534.M.pdf?isAllowed=y&amp;sequence=1">http://tesis.ucsm.edu.pe/repositorio/bitstream/handle/UCSM/10007/70.2534.M.pdf?isAllowed=y&amp;sequence=1</a>		0.07%

## Quotes

Quotes

1

- <sup>1</sup> Delphi - це візуальне середовище програмування (Rapid Application Development - RAD), яка включає в себе крім компілятора ще й редактор, і засоби налагодження, і великі набори готових програм (так звані бібліотеки), наприклад набори математичних функцій.