Рижевский. – М. : "Энергия", 1975. – 250 с.

2. Астафьев Г.П. Радионавигационные устройства и системы/ Г.П. Астафьев, В.С. Шебшаевич, Ю.А. Юрков Изд-во. «Советское радио», 1958.

3. Чмых М.К. Цифровая фазометрия / М.К. Чмых. – М.: Радио и Связь, 1993. – 184с.

4. Горященко, К. Л. Вимірювач кута фазових зсувів за методом коінциденції / К. Л. Горященко, І. В. Гула // Вісник Національного технічного університету України «Київський політехнічний інститут». Серія: Радіотехніка. Радіоапаратобудування. – 2013. – Вип. 53. – С. 74–81

5. Гула І.В. Дослідження оптимальної структури автоматизованого вимірювача фази сигналів на основі принципу коінциденції / І.В. Гула // Вісник ХНУ. – Технічні науки. – 2013. - №4. – С. 230-232.

References

1. Bogorodyczk`j A.A Nonyusnыe analogovo-cyfrovыe preobrazovately / A.A Bogorodyczkyj, A.G. Ryzhevskyj. – M. : "Эnergyya", 1975. – 250 s.

2. Astafev G.P. Radyonavygacyonnыe ustrojstva y systemы/ G.P. Astafev, V.S. Shebshaevych, Yu.A. Yurkov Yzd-vo. «Sovetskoe rad`o», 1958.

3. Chmukh M.K. Tsyfrovaia fazometryia. – M.: Radyo y Sviaz, 1993. – 184s.

4. Horiashchenko, K. L. Hula I. V. Vymiriuvach kuta fazovykh zsuviv za metodom kointsydentsii. Visnyk Natsionalnoho tekhnichnoho universytetu Ukrainy «Kyivskyi politekhnichnyi instytut». Seriia: Radiotekhnika. Radioaparatobuduvannia. – 2013. – Vyp. 53. – S. 74–81

5. Hula I.V. Doslidzhennia optymalnoi struktury avtomatyzovanoho vymiriuvacha fazy syhnaliv na osnovi pryntsypu kointsydentsii. Visnyk KhNU. – Tekhnichni nauky. – 2013. - #4. – S. 230-232.

Рецензія/Peer review : 26.09.2013 р. Надрукована/Printed :6.12.2013 р.
Рецензент: Троцишин І.В., д.т.н., проф. кафедри ТЕЗ
Одеської національної академії зв'язку імені А.С. Попова.

УДК 004.415.538

Ю.В. ПОРЕМСЬКИЙ, О.В. ВАСЬКОВСЬКИЙ, А.С. СОТНІКОВА
Вінницький національний технічний університет

ОПТИМІЗАЦІЯ ПРОЦЕСУ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З ВЕЛИКОЮ КІЛЬКІСТЮ ВХІДНИХ ПАРАМЕТРІВ

Запропоновано новий метод тестування програмного забезпечення, що дозволяє зменшити кількість тест кейсів, які повинні бути створені та виконані. Це в рівній мірі може бути застосовано для unit, інтеграційного та системного тестування, а також тестування прийому.

Ключові слова: програмне забезпечення(ПЗ), тестування, специфікація, тест-план, тест кейс, попарне тестування, баг.

Y.V. POREMSKYY, A.V. VASKOVSKY, A.S. SOTNIKOVA
Vinnytsia national technical university

OPTIMIZATION OF TESTING PROCESS FOR SOFTWARE WITH BIG NUMBER OF INPUT PARAMETERS

Proposed a new method of software testing which allows to reduce number of test cases, that should be created and passed. This can be used for unit, integration, system and acceptance testing.

Key words: software (SW), testing, specification, test-plan, test case, pairwise testing, bug.

Постановка задачі

Незважаючи на безліч існуючих визначень, тестування в своїй основі є процесом зіставлення “того, що є” з “тим, що має бути” [1]. Більш формальне визначення дається стандартом IEEE Standard 610.12-1990, “IEEE Standard Glossary of Software Engineering Terminology” [2], який визначає “тестування” як:

“процес оперування системою або компонентом у визначених умовах, спостереження або запису результатів, та проведення аналізу деякого аспекту системи чи компоненту”.

В свою чергу «тестування» – це процес керованого експериментування з програмним продуктом за допомогою тестів, з метою виявлення в ньому помилок, тобто виявлення неточностей допущених розробниками програмного забезпечення. «Тест» – контрольна задача для перевірки коректності функціонування системи та/або програмного забезпечення [2].

Основна ідея тестування – виконати запуск ПЗ і спостерігати за його роботою (властивостями поведінки) та її результатами. У випадку, коли відбувся збій в роботі програми, виконується аналіз звіту з метою виявлення місцезнаходження помилки, яка його викликала. Зрозуміло, що вдалим тестом є той, при якому виконання програми закінчилось помилкою і навпаки. Тестування виконує дві основні задачі:

1) знаходження і усунення помилок в ПЗ;

2) демонстрація якості функціонування ПЗ.

3) відповідність ПЗ своїй специфікації.

В зв’язку з тим, що процес тестування ітераційний, то після виявлення і виправлення кожної

помилки обов'язково слідує повторення виконання тестів, що має на меті перевірку працездатності ПЗ. Такий процес отримав назву регресивного тестування[3]. Більше того, для ідентифікації причини виявленої проблеми може бути потрібно проведення спеціального додаткового тестування. При цьому завжди потрібно пам'ятати фундаментальний висновок, зроблений професором Едсжером Дейкстрою в 1972 г: "Тестування програм може служити доказом наявності помилок, але ніколи не доведе їх відсутність!" [4].

Помилковим є думка, що можна розробити ПЗ повністю протестувавши його, знайшовши всі помилки, і підсумувавши, що програма ідеальна. Нажаль, це неможливо через наступні чотири ключові причини [5]:

1. Кількість можливих вхідних та вихідних даних дуже велика для повного перебору.
2. Кількість можливих послідовностей виконання коду програми дуже велика, щоб її можна було перевірити повністю.
3. Кількість можливих комбінацій дій користувача з інтерфейсом та його переміщення по програмі занадто велика для повної перевірки.
4. Специфікація ПЗ суб'єктивна. Можна сказати, що помилка – це зовсім не помилка, а так і було задумано.

Під час створення програмного забезпечення існує ймовірність того, що програміст зробив помилку, яка впливає саме на конкретну програмну ситуацію. Якщо не протестувати її, користувач рано чи пізно з нею зіткнеться. Ця помилка може коштувати дуже «дорого», адже вона буде знайдена, коли ПЗ вже знаходиться в експлуатації, а тому на вилучення та заміну версії, що містить баг, мають бути витрачені значні зусилля.

Таким чином, маємо протиріччя – неможливе проведення повного тестування, проте якщо всю функціональність не протестувати, то виникає ймовірність пропущення помилок. В такому випадку перед розробником стоїть суперечлива ціль - продукт повинен бути реалізований, отже, необхідно закінчити процес тестування, але якщо приділити йому занадто мало часу, то залишаться непротестовані частини.

В свою чергу інженеру з якості ПЗ необхідно навчитися скорочувати величезну область всіх можливих тестів до керованого набору та приймати, враховуючий ризик, розумні рішення: що є важливим для тестування, а що ні.

На рис. 1 проілюстровано залежність між обсягом тестування та якістю проекту у сенсі кількості знайдених помилок. Якщо спробувати протестувати абсолютно все, ціна різко зросте, а кількість пропущених помилок спаде до нуля [2].

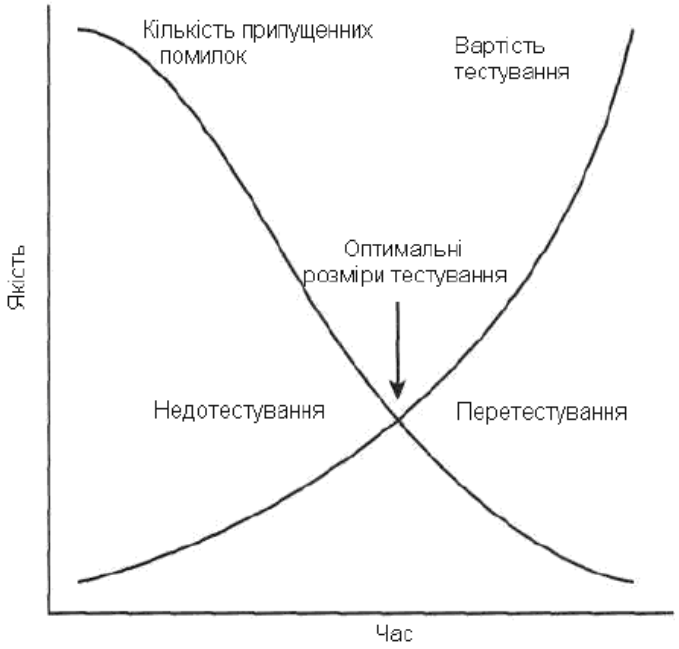

Рис 1. Якісно-часова залежність оптимальності розмірів тестування

Кожен проект має оптимальний обсяг тестування. Якщо сильно скоротити тестування або приймати хибні рішення відносно того, що саме тестувати, собівартість ПЗ зменшиться, але залишиться велика кількість помилок. В зв’язку з цим головною метою є знаходження оптимального обсягу тестових сценаріїв (тест кейсів).

Якщо програмний продукт має велику кількість параметрів та їх комбінацій, які необхідно протестувати, то у випадку, коли ми не протестуємо деякі з них, це може бути ризиковим для її подальшого використання. Ми повинні, деяким чином, обрати логічно-обґрунтовану підмножину тестів, обмежену нашими ресурсами. Але виникає наступне питання «Які є методи вибору такої підмножини?». Усі ці методи можна зобразити у вигляді списку, впорядкованого від найгіршого методу до найкращого [6]:

- Не тестувати взагалі через велику кількість вхідних комбінацій, а отже, величезну кількість тест кейсів.
- Тестувати усі комбінації, але відкласти випуск проекту на невизначений термін.
- Обрати один або два тести та сподіватись на краще.
- Обрати тести, які вже виконувались (можливо, як частина тестів, виконаних програмістом). Об’єднати їх у тест-план та виконати ще раз.
- Обрати тести, які легко створити та виконати, ігноруючи корисність, отриманої в результаті їх виконання, інформації.
- Створити список усіх можливих комбінацій параметрів та обрати декілька перших.
- Створити список усіх комбінацій та обрати з них довільну підмножину.
- Обрати спеціально відібрану достатньо малу підмножину, яка б знаходила дуже велику

кількість багів - більшу, ніж Ви очікували від даної підмножини.

Останній метод виглядає найкраще. Проте, виникає задача щодо отримання “спеціально відібраної” підмножини.

Для вирішення цієї задачі необхідно проаналізувати існуючі проблеми в тестуванні. Як це вже зазначалося раніше, найбільшу кількість тестів вимагає перевірка всієї різноманітності вхідних та вихідних даних та їх комбінацій. Отже, якщо ми зможемо оптимізувати процес вибірки множини тестованих параметрів, ми зможемо зменшити і кількість необхідних тест кейсів без значної втрати надійності процесу оцінки якості.

Розглянемо тестування усіх можливих комбінацій параметрів використовуючи декартів добуток. Декартів добуток - це сценарій, за яким кожен елемент групи співставляється з кожним елементом кожної групи так, що ми отримуємо усі комбінації елементів усіх груп.

Для прикладу, розглянемо просте програмне забезпечення: одно-екранний графічний інтерфейс користувача з двома випадаючими списками та кнопкою “OK”. Список_1 складається з трьох значень: “0”, “1”, “2”; Список_2 складається з двох значень: “10” та “20”. Користувач обирає одне значення з кожного списку і, після натиснення кнопки “OK”, на екрані з'являється добуток двох обраних значень. Змінні та їх значення виглядають наступним чином(Таблиця 1):

Таблиця 1

Приклад списків параметрів простого програмного забезпечення

Список_1	Список_2
0	10
1	20
2	

В даному випадку ми маємо 3*2=6 комбінацій параметрів. Результатом декартового добутку двох вхідних значень є 6 комбінацій, які мають такий вигляд(Таблиця 2):

Таблиця 2

Можливі комбінації параметрів простого програмного забезпечення

Список_1	Список_2
0	10
0	20
1	10
1	20
2	10
2	20

Як зрозуміло з вищенаведеної таблиці, кожне значення співставляються одне з одним. Такий тест легко виконати вручну менше, ніж за 2 хвилини. Але якщо програма складається з більшої кількості вхідних параметрів, виникає наступне питання «наскільки зросте кількість можливих комбінацій?» Для відповіді на це питання розглянемо наступний приклад.

Нехай Список_1 (List) складається з цілих чисел від 0 до 9, а Список_2 змінений на текстове поле (Textbox), в яке можливо вводити будь-яке ціле значення від 1 до 99. Також, в програмі існують 2 чек-бокси. Якщо чек-бокс_1 (Negative) відмічений (On), то значення добутку змінних List та Textbox множиться на “-1” (стає від'ємним). Якщо відмічений чек-бокс_2 (Square), то добуток підноситься до квадрату.

Тепер ми маємо 10*99*2*2=3960 можливих вхідних комбінацій та безліч недопустимих комбінацій (поле Textbox має можливість введення некоректних даних).

В основі попарного тестування полягає ідея того, що немає необхідності виконувати усі можливі тестові комбінації. Нехай в наведеному прикладі існує баг, при якому, при введенні значень List=0 та Negative=On, програма не буде працювати коректно. Як результат, ПЗ повертає помилку, коли виникає необхідність перетворити значення, яке рівне нулю на від'ємне число.

Очевидно, що 198 з розроблених тест кейсів зіткнуться з цією проблемою. Таким чином, виникає надлишковість операцій тестування, оскільки цей баг можна знайти за допомогою всього лише однієї тестової комбінації, яка включає значення List=0 та Negative=On.

Так як більшість багів відтворюються при конфлікті значень всього двох змінних, то немає необхідності тестувати усі комбінації значень, потрібно лише протестувати усі пари параметрів. Що дозволить значно скоротити час тестування ПЗ.

В прикладі, що наведено вище, користувач має більше можливості ввести некоректні дані в поле Textbox, ніж у випадаюче меню. До тих пір, поки введенні дані не є дискретними, їх можна об'єднувати в групи. Так, в нашому випадку, 99 можливих коректних та безліч некоректних значень можна скоротити до трьох груп:

- будь-яке коректне числове значення - Valid_int;
- будь-яке некоректне числове значення - Invalid_int;

- будь-які графічні символи (які будуть некоректними) - Alpha.

Крім того надається можливість скоротити число значень змінної List з 10 до двох: 0 та будь-яке інше значення - Other. Як результат, змінні та значення мають наступний вигляд, що наведений в таблиці 3.

Таблиця 3

Об'єднання параметрів у групи

List	Textbox	Negative	Square
0	Valid_int	On	On
Other	Invalid_int	Off	Off
	Alpha		

Відсортуємо тести таким чином, щоб ті, які мають найбільшу кількість значень, були першими, а ті, що мають найменшу кількість значень - останніми. В нашому випадку, першою змінною (стовпцем) таблиці буде Textbox, яка має 3 можливих значення.

Наступним кроком є процес заповнення таблиці, де кожен рядок буде окремим тестовим сценарієм. Процес заповнення виконується послідовно, тобто стовпець за стовпцем. Необхідно враховувати, скільки значень має друга колонка. В нашому прикладі – 2 параметри. Це означає, що кожне значення стовпця Textbox потрібно повторити двічі. Кожному значенню Textbox необхідно співставити кожне з двох значень стовпця List. В результаті ми отримаємо відповідну таблицю:

Таблиця 4

Створення пар параметрів з найбільшою кількістю значень

Textbox	List	Negative	Square
Valid_int	0		
Valid_int	Other		
Invalid_int	0		
Invalid_int	Other		
Alpha	0		
Alpha	Other		

Продовжимо заповнювати третій стовпець таблиці. Змінна Negative має два значення: On та Off. Проставимо ці значення в стовпець Negative:

Таблиця 5

Додавання параметру Negative до існуючої множини тестів

Textbox	List	Negative	Square
Valid_int	0	On	
Valid_int	Other	Off	
Invalid_int	0	On	
Invalid_int	Other	Off	
Alpha	0	On	
Alpha	Other	Off	

Перевіримо, чи отримано усі комбінації значень для стовпців List та Negative:

- пара значень 0/On, але без пари 0/Off;
- пара значень Other/Off, але без пари Other/On.

Як результат, змінимо місцями значення On та Off другої пари третього стовпця:

Таблиця 6

Адаптація параметру Negative відповідно до умови наявності всіх пар

Textbox	List	Negative	Square
Valid_int	0	On	
Valid_int	Other	Off	
Invalid_int	0	**Off**	
Invalid_int	Other	**On**	
Alpha	0	On	
Alpha	Other	Off	

Варто відмітити, що для останнього набору значень On та Off вже існують усі можливі пари і для нас не має значення порядок On/Off чи Off/On.

Заповнимо четвертий стовпець. Змінна Square також має два значення: On та Off. Для того, щоб не плутати значення стовпців 3 та 4, замінимо назви значень On та Off для стовпця Square на Checked та Unchecked. Заповнимо даний стовпець таким чином, щоб у нас утворилися усі можливі пари значень (Табл. 7):

Таблиця 7

Додавання параметру Square та його адаптація відповідно до умови наявності всіх пар

Textbox	List	Negative	Square
Valid_int	0	On	Checked
Valid_int	Other	Off	Unchecked
Invalid_int	0	Off	Checked
Invalid_int	Other	On	Unchecked
Alpha	0	On	Unchecked
Alpha	Other	Off	Checked

Ще раз перевіримо створені пари значень для колонок 2 та 4. У нас є пари 0/Checked та 0/Unchecked. У нас є пари Other/Checked та Other/Unchecked. Тепер перевіримо усі пари в стовпцях 3 та 4. У нас є пари On/Checked та Off/Unchecked, Off/Checked та Off/Unchecked. Всі можливі пари створені.

Ми отримали 6 тестових сценаріїв. Якщо ми будемо тестувати усі комбінації, то ми отримаємо 3*2*2*2=24. Нехай, наступна версія програми має ще 2 чек-бокси. Чек-бокс_3 (Factorial) бере факторіал від добутку значень змінних List та Textbox, а чек-бокс_4 (Hex) перетворює добуток в шістнадцятирічну систему числення і має два значення: Dec - десяткове представлення та Hex - шістнадцятирічне представлення. Додамо ще два стовпця до нашої таблиці та введемо до неї значення для стовпця Factorial(Таблиця 8):

Таблиця 8

Модифікація тестів після додавання нових параметрів в програмний продукт

Textbox	List	Negative	Square	Factorial	Hex
Valid_int	0	On	Checked	Yes	Dec
Valid_int	Other	Off	Unchecked	No	Hex
Invalid_int	0	Off	Checked	No	Hex
Invalid_int	Other	On	Unchecked	Yes	Dec
Alpha	0	On	Unchecked	No	Dec
Alpha	Other	Off	Checked	Yes	Hex

Отримано наступні результати:

- Стовпець 2 в порядку: у нас є всі можливі пари: 0/Dec, 0/Hex, Other/Dec та Other/Hex.
- Стовпець 3 містить проблему: у нас є пари On/Dec та Off/Hex, але немає пар On/Hex та Off/Dec.
- Стовпець 4 в порядку: у нас є всі можливі пари: Checked/Dec, Checked/Hex, Unchecked/Dec та Unchecked/Hex.
- Стовпець 5 в порядку: у нас є всі можливі пари: Yes/Dec, Yes/Hex, No/Dec та No/Hex.

В даній ситуації не можемо створити відсутні пари значень шляхом перестановки значень одного зі стовпців 2 або 6, оскільки це призведе до зникнення існуючих пар в інших стовпцях. В такому випадку, нам необхідно додати ще 2 тест кейси, які б реалізовували відсутні комбінації значень. Значення решти параметрів для новостворених сценаріїв не мають значень та можуть бути будь-якими, оскільки усі можливі пари вже створені. Заповнимо пусті поля таблиці(Таблиця 9):

Таблиця 9

Створення нових тестів для перевірки нових пар програмного продукту

Textbox	List	Negative	Square	Factorial	Hex
Valid_int	0	On	Checked	Yes	Dec
Valid_int	Other	Off	Unchecked	No	Hex
		On			**Hex**
Invalid_int	0	Off	Checked	No	Hex
Invalid_int	Other	On	Unchecked	Yes	Dec
		Off			**Dec**
Alpha	0	On	Unchecked	No	Dec
Alpha	Other	Off	Checked	Yes	Hex

Тепер ми маємо усі можливі 3*2*2*2*2*2=96 комбінацій значень в восьми тестових сценаріях, замість 99*10*2*2*2*2=15,840 тестів, які нам би довелось виконати, перевіряючи усі можливі тестові комбінації усіх значень усіх параметрів окремо.

На останньому прикладі ми переконались, що запропонований метод попарного тестування надає можливість гнучко адаптуватися до змін в вимогах до програмного продукту та розробляти нові тест кейси без необхідності нових розрахунків оптимізації процесу перевірки якості. Це має велике значення для проектів, які застосовують каскадний метод розробки ПЗ.

Висновки

Запропоновано новий метод тестування програмного забезпечення, що дозволяє зменшити кількість тест кейсів, які повинні бути створені та виконані. Це метод може бути застосовано для unit, інтеграційного та системного тестування, а також тестування прийому.

Крім того розроблений підхід, який є особливо актуальний в задачах з великою різноманітністю вхідних параметрів. Основними його перевагами є

- ефективне підвищення швидкості процесу тестування;
- можливість розвитку розробленого тест-плану після внесення змін до проекту;
- можливість зменшення вартості програмного продукту, що розробляється, за рахунок зменшення витрат на його тестування.

Література

1. Дідковська М.В. Тестування: Основні визначення, аксіоми та принципи. Текст лекцій. Частина І / Дідковська М.В., Тимошенко Ю.О.

2. IEEE Standard 610.12-1990, “IEEE Standard Glossary of Software Engineering Terminology”.

3. Бейзер Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем / Б. Бейзер. — СбП: Питер, 2004. — 320 с.

4. Дейкстра Э. Дисциплина программирования / Э.Дейкстра ; пер. с англ. И. Х. Зусман ; ред. Э. З. Любимский. — М. : Мир, 1978. — 275 с.

5. Канер С. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений: Пер. с англ. / Сэм Канер, Джек Фолк, Енг Кек Нгуен. - К.: ДиаСофт, 2001. - 544 с.

6. Copeland L. A Practitioner’s Guide to Software Test Design / Lee Copeland // Artech House, 2004. - 276.

7. Berger B. Efficient Testing with All-Pairs / Bernie Berger // STAREast 2003 International Conference on Software Testing, 2003.

References

1. Didkovska M. Testing: Basic definitions, axioms and principles. The text of the lectures. Part I / Didkovska M., Timoshenko Yu.

2. IEEE Standard 610.12-1990, “IEEE Standard Glossary of Software Engineering Terminology”.

3. Beizer B. Black box testing. Technology functional testing of software and systems / B. Beiser. - St.-P.: Peter, 2004. - 320.

4. Dijkstra E. Discipline Program / E.Deykstra trans. from English. IH Zussman, ed. EZ Lyubimsky. - New York: Wiley, 1978. - 275.

5. Kaner C. Software Testing. Fundamental concepts of management of business applications: Per. from English. / Sam Kaner, Jack Falk, Yong Keck Nguyen. - K.: DiaSoft 2001. - 544.

6. Copeland L. A Practitioner’s Guide to Software Test Design / Lee Copeland // Artech House, 2004. - 276.

7. Berger B. Efficient Testing with All-Pairs / Bernie Berger // STAREast 2003 International Conference on Software Testing, 2003.

Рецензія/Peer review : 13.7.2013 р. Надрукована/Printed :21.12.2013 р.

УДК 621.396.662

О.І. ПОЛІКАРОВСЬКИХ
Хмельницький національний університет
І.В. ТРОЦИШИН
Одеська національна академія звязку ім. О.С.Попова

ПРИНЦИПИ ПОБУДОВИ СТРУКТУРНИХ ОДИНИЦЬ ПЕРСПЕКТИВНИХ ПРЯМИХ ЦИФРОВИХ СИНТЕЗАТОРІВ ЧАСТОТИ

Розглянуто принципи побудови прямих цифрових синтезаторів частоти із фазовим ядром на основі суматора у базисі Галуа. Розглянуто математичні моделі та структури спеціалізованих структурних одиниць прямих цифрових синтезаторів частоти на основі модулярної арифметики.

Ключові слова: : обчислювальний синтезатор частоти, модулярна арифметика, суматор, поле Галуа

O.I. POLIKAROVSKYKH
Khmelnitsky National University
I.V. TROCISHIN
Odessa National Academy of Telecommunications named after O.S.Popov

PRINCIPLES OF CONSTRUCTION OF STRUCTURAL UNITS FOR DIRECT DIGITAL FREQUENCY SYNTHESIZER

Principles of construction of the direct digital frequency synthesizer with phase core-based adder in the Galois basis was discussed. The mathematical model and the structure of specialized structural units of direct digital frequency synthesizers based on modular arithmetic was discussed.

Keywords: Galois field, modular arithmetic, direct frequency synthesizer (DDS).

Постановка задачі

Прямі цифрові синтезатори частоти відіграють важливу роль у сучасних радіоелектронних