**УДК 004.054**

Д.В. ФЕДАСЮК, В.С. ЯКОВИНА, П.В. СЕРДЮК, О.О. НИТРЕБИЧ

Національний університет «Львівська політехніка», м.Львів

# МЕТОД ПОБУДОВИ СЦЕНАРІЇВ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ АНАЛІЗУ ЙОГО ЗМІННИХ

**Анотація.** Підвищення надійності програмного продукту є надзвичайно важливою та актуальною науковою задачею. Основним засобом вирішення даної проблеми є тестування програмного забезпечення (ПЗ), що вимагає значних затрат ресурсів, адже у більшості випадків проводиться вручну та повинно охоплювати усі сценарії використання і потоки управління та даних програмного продукту. У цій роботі описано метод автоматизованої побудови сценаріїв тестування на основі розробленої моделі поведінки ПЗ з урахуванням його змінних, що дозволяє скоротити фінансові, часові та людські ресурси та забезпечує рівномірне покриття коду.

**Ключові слова: тестування на основі моделі, сценарій тестування, програмна відмова, модель поведінки ПЗ.**

**Аннотация.** Повышение надежности программного продукта является чрезвычайно важной и актуальной научной задачей. Основным средством решения данной проблемы является тестирование программного обеспечения (ПО), которое требует значительных затрат ресурсов, ведь в большинстве случаев производится вручную и должно охватывать все сценарии использования и потоки управления и данных программного продукта. В этой работе реализован метод автоматизированного построения сценариев тестирования на основе разработанной модели поведения ПО с учетом его переменных, что позволяет сократить финансовые, временные и человеческие ресурсы и обеспечивает равномерное покрытие кода.

**Ключевые слова: тестирования на основе модели, сценарий тестирования, программный отказ, модель поведения ПО.**

**Abstract.** Improving the software reliability is very important and actual scientific task. The primary tool for solving this problem is software testing, which requires significant resources because it's mostly manual and should cover all usage scenarious and control and data flows of software. In this paper the new method for automated test cases construction is implemented. It is based on developed software usage model with consideration of its variables, therefore allowing to reduce financial, temporal and human resources and provide a uniform code coverage.

**Key words: model-based testing, test case, software failure, software usage model.**

## Вступ

На сьогодні тестування є одним з найбільш важливих засобів перевірки надійності програмного забезпечення. Однак, зазвичай це ручне тестування, що є трудомістким процесом без ефективної автоматизації, яке потребує багато часу та фінансових ресурсів, але не здатне забезпечити виявлення усіх відмов ПЗ. Отже, важливим завданням у сфері програмної інженерії є автоматизація процесу тестування, а саме ефективна побудова сценаріїв тестування, що дозволить скоротити часові та економічні затратити.

Однією з найновіших технологій для автоматизованої побудови сценаріїв тестування програмного забезпечення є тестування на основі моделі програмного продукту [1-3] – тестування ПЗ, в якому тестові сценарії частково або повністю будуються з моделі, яка описує деякі аспекти (частіше функціональні) тестованої системи. Моделі програмної системи можуть відображати її поведінку або використовуватися для створення тестових стратегій чи середовища тестування.

Існує ряд моделей програмного забезпечення, що використовуються засобами автоматизованої побудови тестових сценаріїв. У загальному вони поділяються на чотири великі групи, які у свою чергу під час моделювання відповідно використовують [4]: скінченні автомати, граф станів, UML діаграми та Марковські ланцюги. Окрім того існують моделі на основі дерева прийняття рішень, таблиць рішень та граматик, але вони є новими і не достатньо дослідженими [4].

Граф станів, який ще іноді називають "потік тестування" використовується для моделювання програмних систем [5, 6] та побудови тестових сценаріїв та відображає реальне використання таких систем, і, таким чином, є більш точним, ніж використання скінченних автоматів. Перевагами моделей даного класу є можливість їхнього застосування для багатопотокових програм, але все-таки такі моделі тяжко спроектувати та реалізувати для складних програмних систем.

Широкого застосування набуло тестування програмної системи на основі UML діаграм [7-10]. Зазвичай автоматизоване генерування тестових сценаріїв відбувається на основі діаграми станів [8], прецедентів [9], послідовності [10]. Основними перевагами такого підходу є використання для складних систем, а можливість моделювання паралельного програмного забезпечення.

Моделі на основі скінченних автоматів є відомими динамічними моделями для автоматизованої побудови тестів [11, 12]. Формально скінченні автомати, що зображують програмне забезпечення, є сукупністю п'яти параметрів (*I, S, T, F, L*), де *I* – множина вхідних параметрів ПЗ, *S* – множина всіх станів системи, *T* – це функція, яка визначає, чи відбувається перехід, коли системи з деякого стану, *F* – множина станів системи, які є кінцевими (з них система завершує свою роботу) та *L* – стани, з яких система починає свою роботу. Основними недоліками використання даних моделей є те, що їх легко спроектувати й підтримувати лише для систем із малим рівнем складності, неможливість використання даного класу моделей для паралельних ПЗ та систем з нескінченною кількістю станів.

Ще однією великою групою моделей для автоматизованої побудови тестових сценаріїв є моделі на основі дискретного Марковського ланцюга [13, 14]. За структурою Марковські ланцюги подібні до скінченних автоматів, але очевидною перевагою використання такого підходу є не лише генерація тестів,

але й аналіз відмов для оцінки та прогнозування надійності програмного продукту. Сценарії тестування отримуються через послідовні переходи між компонентами у відповідності до ймовірностей передачі контролю. Генерування тестових сценаріїв може бути автоматизоване за допомогою хорошого генератора випадкових чисел на будь-якій мові програмування високого рівня.

Таким чином для адекватного аналізу надійності ПЗ важливими задачами є удосконалення та розробка нових моделей на основі Марковського ланцюга, за допомогою яких можна здійснювати автоматизоване генерування тестів з урахуванням взаємозалежності переходів від однієї компоненти до іншої.

У даній роботі на основі розробленої авторами моделі поведінки ПЗ, що базується на Марковських ланцюгах [15], представлено метод автоматизованої побудови набору сценаріїв тестування, який враховує залежність виконання компонент та забезпечує рівномірне покриття коду, що дозволяє підвищити ефективність процесу тестування та зменшити його затрати. Окрім того даний метод можна застосувати для моделювання процесу тестування, що дасть змогу визначити вплив характеристик програмної системи на її надійність.

**Актуальність**

В останні роки складність комп’ютерних програм, що виконують важливі та відповідальні функції, швидко зростає, тому проблема підвищення надійності програмного забезпечення стає все більш актуальною науковою задачею. Тестування програмного продукту є основним засобом аналізу надійності програмних систем, яке ще до тепер часто є ручним, тому важливо розробляти нові методи автоматизованої побудови сценаріїв тестування, що значно підвищить ефективність процесу тестування. Окрім того, під час побудови тестових сценаріїв важливим є аналіз змінних програмного продукту, адже багато моделей оцінки надійності ПЗ використовують його метрики, які в свою базуються на змінних ПЗ.

**Мета**

Метою даної роботи є розроблення методу побудови сценаріїв тестування ПЗ стратегіями «чорної» та «білої» скриньки на основі моделі поведінки програмного продукту, що базується на аналізі його змінних.

**Модель поведінки програмного продукту на основі його змінних**

Модель поведінки ПЗ, що є вхідним параметром для автоматизованої побудови тестів, зображується у вигляді орієнтованого графу $G = \{S, P\}$, де $S = \{ S^0, S^1, ..., S^n \}$ – множина методів програми $S^i$, $P$ – множина переходів між відповідними методами [15].

Якщо позначити:

- $V$ – множина усіх змінних програмного забезпечення;
- $V^i$ – множина $\{ v, E \}$, де $v$ – змінна програмного продукту, а $E = \{ E_{cor}, E_{incor} \}$ – множина відповідних правильних $E_{cor}$ та неправильних $E_{incor}$ класів еквівалентності такої змінної [16];
- $V^i_{used} = \{ V^i, i = \overline{1,m} \}$ – множинна змінних, які використовуються у методі $S^i$; Вважаємо, що змінні та їхні значення є основним джерелом помило програмного забезпечення;
- $V^i_{change} = V^i_{change\_user} \cup V^i_{change\_program}$ – список змінних, що змінюються у методі $S^i$. Ця множина є об’єднанням множини змінних, що можуть змінюватись користувачем $V^i_{change\_user} = \{ V^i, i = \overline{1,l} \}$ (змінні, які можна протестувати за допомогою стратегії “чорної” скриньки), а також множина змінних, що можуть бути змінені лише внутрішньою логікою програми $V^i_{change\_program} = \{ V^i, i = \overline{l+1,k} \}$ (тестуються лише “білою” скринькою, $k$ – кількість змінних, що змінюються у методі $S^i$);
- $V^i_{error} = \{ V, E_{incor} \}$ – список змінних та відповідних неправильних класів еквівалентності, що можуть викликати помилку в програмній системі.

Тоді кожен вузол графу $S^i$ є набором відповідних $\{ V^i_{used}, V^i_{change}, V^i_{error} \}$.

Як відомо сьогодні існує дві найбільш важливі техніки, на основі яких здійснюють побудову тестових сценаріїв, а також проводять аналіз надійності та якості програмного продукту [17]:

1. Стратегія «чорної скриньки», яка розглядає системні характеристики програм, ігноруючи їхню внутрішню логічну структуру.

2. Стратегія «білої скриньки», яка передбачає детальне дослідження внутрішньої логіки і структури коду, для якого повинна бути відома повна інформація про вихідний код ПЗ.

Унаслідок врахування усіх змінних і відповідних класів еквівалентності, розроблена модель використання програмного продукту більш адекватно описує поведінку програмного забезпечення та дає можливість побудувати сценарії тестування ПЗ стратегіями як "чорної", так і "білої" скриньки.

Розглянемо побудову сценаріїв тестування для кожної з цих стратегій.

**Побудова тестових сценаріїв методом «чорної» скриньки**

Вважатимемо, що кожен сценарій тестування складається з декількох кроків $T^k = \{T_k^0, T_k^1, ..., T_k^n\}$, де кожен крок сценарію $T_j^k = (S_j^k, C_j^k)$ визначається парою: методом $S_j^k$ та множиною змінних $C_j^k$ з відповідними класами еквівалентності, що точно змінюються в цьому методі. Змінні набувають своїх значень на кожному кроці тесту внаслідок введення даних користувачем, зчитуванням з бази даних та ін., через що можуть виникнути помилки під час виконання даного сценарію тестування.

Варто зауважити, що у відповідності до реального процесу тестування, набори тестів будуються ітераційно, крок за кроком виконуючи їх на програмному забезпеченні та враховуючи виявлені помилки під час побудови наступного набору тестів.

Введемо деякі позначення: $Er^j(S^i)$ – число тестів, під час яких відбулась відмова, і які включають метод $S^i$ на *j*-тій ітерації. $M(S^i)$ – множина номерів усіх методів, що мають переходи із компоненти $S^i$; $P(S^i)$ – множина всіх методів, у які можна передати контроль з методу $S^i$; $N$ – число всіх вершин графу поведінки програмної системи; $TS$ – набір усіх сценаріїв тестування; coverage($S^i$) – кількість входжень в метод $S^i$.

Алгоритм автоматизованої побудови тестових сценаріїв складається із двох основних частин: початкова генерація тестів без інформації про відмови та решти *K* наборів тестів, що її включають.

Крок1. *Ініціалізація.* Вибір мінімального числа покриття всіх компонент ПЗ $\alpha$ та середньої довжини тестового сценарію $\beta$ з досвіду тестувальника. Рекомендованим діапазоном початкових значень цих параметрів може бути $\beta \in \overline{5,7}$, $a \in \overline{5,10}$. Установлення $TS$={}, номер ітерації тестування $m$=0, номер тесту $k$=0.

Крок2. Побудова сценарію тестування $T_k$ під час початкової генерації набору тестів.

1. $j$=0 – номер кроку такого тесту;
2. Випадковим чином обирається початковий крок з ймовірністю $p_k^j = 1/N$;
3. З ймовірністю $p_k^{j+1} = 1/\left\|P(S^i)\right\|$ вибирається наступний крок $T_k^{j+1}$ у *k*-ому сценарії; $j$++;
4. Якщо $j \le \beta$, тоді перехід на Крок2.3;
5. Додавання побудованого тесту до набору: $TS = TS \cup \{T_k\}$;

Крок3. Перевірка покриття усіх компонент. Якщо coverage$(S^i) < \alpha, i = \overline{1,N}$, тоді $k$++, перехід на Крок2.

Крок4. Виконання усіх побудованих тестів та обчислення $Er^m(S^i), i = \overline{1,N}$. Якщо $Er^m(S^i) = \varnothing, i = \overline{1,N}$, тоді Крок9, інакше $m$++;

Крок5. *Побудова $m$-го набору тестів ($m \ge 1$). $TS$={}, $k$=0.*

Крок6. Побудова $T_k$ сценарію тестування для $m$-го набору тестів .

1. $j$=0 – номер кроку такого тесту;

2. Випадковим чином обирається номер першого кроку, що буде відповідно першим кроком $T_k^j$ тесту, з ймовірністю $p_k^j = \max\limits_{i=1..N} \dfrac{Er^{m-1}(S^i)}{\sum\limits_{l=1}^{N} Er^{m-1}(S^l)}$;
3. З ймовірністю $p_k^{j+1} = \max\limits_{i \in M(S^j)} \dfrac{Er^{m-1}(S^i)}{\sum\limits_{l \in M(S^j)} Er^{m-1}(S^l)}$ обирається наступний крок у $T_k^{j+1}$ сценарії; $j$++;
4. Якщо $j \leq \beta$, тоді перехід на Крок6.3;
5. Додавання побудованого тесту до набору: $TS = TS \cup \{T_k\}$.

Крок7. Перевірка покриття усіх компонент. Якщо $\text{coverage}(S^i) < \alpha, i = \overline{1,N}$, тоді $k$++, перехід на Крок6.

Step8. Перехід на Крок4.

Step9. Кінець.

Побудовані послідовності методів виконуються при різних вхідних даних таким чином, щоб кожний вхідний параметр набував принаймні одного значення з кожного класу еквівалентності. Класи еквівалентності у методах чорної скриньки будуються не з аналізу коду, а відповідно до досвіду тестувальника. При цьому вхідні дані можуть і не набути усіх можливих значень і деякі відмови будуть пропущені, що є недоліком даної стратегії тестування.

Наприклад, для того, щоб повністю покрити код ПЗ, необхідно для кожного метода $S^i$ здійснити $\|v_1\| \cdot \|v_2\| \cdots \|v_n\|$ перевірок, де $\|v_j\|$ – кількість класів еквівалентності змінної $v_j \in S^i$ $(j = \overline{1..n})$. Але через обмеження часових та людських ресурсів не можливо зробити таку велику кількість перевірок. Окрім того не всі перевірки відповідають реальній логіці програми (певних значень змінних не можливо досягнути під час використання програми), тому існує задача покриття коду тестами так, щоб повністю перебрати усі можливі набори змінних, допустимих внутрішньою логікою, або, якщо таких випадків дуже багато, то перебрати усі кортежі класів еквівалентності змінних довжиною до деякого степеня *m,* який визначається співвідношенням вимог до якості ПЗ та затратами на процес тестування. Стандартно ($m$=1) обмежуються перебором усіх класів еквівалентності для кожної змінної у методі $S^i$ – $\|v_1\| + \|v_2\| + \cdots + \|v_n\|$. Для $m$=2 необхідно перевірити всі пари наборів класів еквівалентності у методі $S^i$ – $\sum\limits_{k,l=1}^{n} \|v_k\| \cdot \|v_l\|$ і т.д. Тобто метод чорної скриньки може не повністю перебрати усі можливі випадки значень змінних, що є суттєвим недоліком даного підходу.

Отже, набори тестів, побудовані на кожній ітерації методом «чорної скриньки», можуть бути використані під час тестування, що у свою чергу дозволяє пришвидшити даний процес та зменшити фінансові та людські витрати. Окрім того, така генерація сценаріїв забезпечує рівномірне покриття коду, що покращує ефективність тестування, та відповідно підвищує якість ПЗ. Також завдяки рівномірному покриттю коду та використанню класів еквівалентності, ймовірність виявлення відмови зростає під час тестування програми, що в результаті збільшує надійність ПЗ.

**Побудова тестових сценаріїв методом «білої» скриньки**

Першою перевагою стратегії «білої» скриньки, яку неважко реалізувати, – це аналіз змінних, які використовуються та змінюються у методах. Відкинувши змінні, що не використовуються, можна зменшити перебір можливих $V_{used}^i$ - підмножини змінних множини $S^i$, яку може використати метод $S^i$ (змінні, що дійсно використовує цей метод). Також можна чітко визначити $V_{change}^i$ - множину, яку може змінювати відповідний метод завдяки аналізу коду. Очевидно, що множна змінних методу $S^i$ рівна $V_{used}^i \cup V_{change}^i$. Перетин цих змінних не обов’язково порожній.

Якщо розглянути тестовий сценарій, який складається лише з одного кроку $T=(S,C)$, то множина змінних, яка покривається даним сценарієм рівна $\alpha(T)=V_{used}\cap C$ . Значення інших змінних можна покрити тільки тестовими сценаріями, які матимуть два або більше кроків. Множина змінних методу $S^i$, яка покривається сценарієм тестування, що містить два кроки $T=\{T^0,T^1\}=\{(S^0,C^0),(S^1,C^1)\}$ та включає два послідовні методи $S^0$ та $S^1$ має вигляд: $\alpha(T)=V^1_{used}\cap\left(C^1\cup C^0\right)$. У загальному випадку для тестового сценарію $T=\{T^0,T^1,...,T^{\mathrm{N}}\}=\{(S^0,C^0),(S^1,C^1),...,(S^N,C^N)\}$ множина змінних, яку він покриває та впливає на процеси останнього методу $S^N$ дорівнює: $\alpha(T)=V^N_{used}\cap\left(C^N\cup C^{N-1}\cup...\cup C^1\cup C^0\right)$. Для того, щоб повністю покрити метод $S^N$ різними значеннями змінних, необхідно, щоб $\alpha(T)=V^N_{used}$ .

Зауважимо також, що сценарій тестування покриває множини змінних також і попередніх методів. Наприклад, для проміжного методу $S^m$ ($m<N$) цього сценарію $T$ множина покриття рівна:

$$\alpha\left(T,S^m\right)=V^m_{used}\cap\left(C^1\cup C^2\cup...\cup C^{m-1}\cup C^m\right). \quad (1)$$

Для останнього вузла: $\alpha\left(T,S^N\right)=\alpha(T)$.

Очевидно, що чим більше кроків у сценарії, тим більше можливостей для задання різних значень змінних. З іншої сторони, зі збільшенням кількості кроків у сценаріях тестування зменшується ймовірність того, що тестовий сценарії буде пройдений до кінця, оскільки він може бути зупинений через виникнення відмови. Для зменшення трудоємності процесу тестування та підвищення його ефективності необхідно вибрати такий набір сценаріїв та кроків у ньому, які би повністю покривали значення усіх змінних методів та мали найменшу кількість кроків. Варто також зауважити, що у програмному забезпеченні буде багато змінних, які набувають своїх значень лише в одному методі (на одній формі, у конфігурації, і т.д), тому під час побудови тестових сценаріїв необхідно враховувати цей факт. Введемо позначення: $c(v_i)$ – кількість методів, у яких змінна може змінитись. Окрім того позначимо максимальну довжину тестового сценарію для даного ПЗ через $\beta_{max}$. Тоді міру покриття PVC (значення параметра покриття, яке вимагає, щоб змінні методу набували усіх своїх значень) визначатимемо як:

$$E(TS)=\sum_{i=1}^{N}\sum_{v_i\in V^i_{used}}\tilde{\delta}(v_i,\bigcup_{T\in TS}\alpha(T,S^i)), \quad (2)$$

де $\tilde{\delta}(v,S)=\begin{cases}0, v\notin S;\\ 1, v\in S.\end{cases}$

Також ведемо поняття, яке буде визначати складність виконання тестів. Оскільки з точки зору тестувальника складність виконання переходу до наступного кроку є значно меншою у більшості випадків, ніж задання даних, то вимірювати складність будемо як кількість даних, що вносить користувач під час побудови тесту.

$$CM(TS)=\sum_{i=1}^{\|TS\|}\sum_{j=1}^{\|T_i\|}C_i^j \quad (3)$$

Таким чином можна формалізувати задачу побудови оптимального набору сценаріїв тестування ПЗ, що має такий вигляд:

$$CM(TS)\to\min$$

$$\text{з обмеженнями } E(TS)=E_{\max}(TS)=\sum_{i=1}^{N}\left\|V^i_{used}\right\|. \quad (4)$$

Алгоритм знаходження оптимальних покриттів є алгоритмом перебору з відсіканням гілок, які є надлишковими згідно міри покриття $E(TS)$ та складності виконання $CM(TS)$.

Крок1. *Ініціалізація*. Побудова початкового набору тестових сценаріїв, що складаються лише з одного кроку $TS=\{(S^0,V^0_{change}),(S^1,V^1_{change}),...,(S^N,V^N_{change})\}$. На початковому кроці тестового

сценарію користувач буде змінювати усі можливі параметри, адже вони впливають на всі подальші кроки.

Крок 2. *Знаходження повного покриття PVC.* Множини тестових сценаріїв, побудованих на попередньому кроці, розширимо таким чином: для сценарію T=$\left((S^0,C^0),(S^1,C^1),...,(S^K,C^K)\right)$ з множини $TS$ знаходимо усі методи $S^{K+1}$, які суміжні методу $S^K$ і додаємо новий крок $(S^{K+1},C^{K+1})$ до сценарію $T$, де $C^{K+1}=V_{change}^{K+1}\setminus\bigcup_{i=1}^{K}C^i$. Додаємо новий сценарій до множини $TS$, якщо при цьому зростає $E(TS)$. Якщо $E(TS)=E_{\max}(TS)$, то переходимо на Крок3, інакше повторюємо Крок2.

Крок3. *Мінімізація складності виконання сценарію тестування.* Для кожного сценарію з множини $TS$ здійснюємо перевірку: якщо під час видалення його з множини $TS$ величина $E(TS)$ не змінюється, то видаляємо його, інакше перевіряємо кожен крок $(S^k,C^k)$, видаляючи з множини $C^k$ ті змінні, які не впливають на $E(TS)$. Це робиться для мінімізації $CM(TS)$.

Крок4. *Перебір класів еквівалентності змінних.* Побудовані тестові сценарії з множини $TS$ виконуються при різних вхідних даних таким чином, щоб кожний вхідний параемтр набував принаймні одного значення з кожного класу еквівалентності. Для підвищення ефективності тестування необхідно проаналізувати взаємозв'язані змінні та перебирати усі кортежі відповідних значень класів еквівалентності, що є значно складнішою задачею. Іншим підходом є більша дискретизація вузлів графу поведінки ПЗ до рівня програмних виразів та операторів управління, що дасть змогу глибоко проаналізувати покриття та взаємний вплив змінних.

Крок5. *Ітераційне виконання тестових сценаріїв.* Після побудови та виконання тестових сценаріїв знаходимо множину методів $S_{err}$, у яких виникли помилки та відповідний набір класів еквівалентності змінних $V_{err}$. Якщо $S_{err}=\varnothing$, то закінчуємо алгоритм, інакше, починаючи з Кроку1, будуємо нову множину тестових сценаріїв $TS$ з новим покриттям:

$$E(TS)=\sum_{S_i\in S_{err}}\sum_{v_i\in V_{used}^i\cap V_{err}^i}\tilde{\delta}(v_i,\bigcup_{T\in TS}\alpha(T,S^i)).$$

Отже, наведений алгоритм дозволяє будувати сценарії тестування ПЗ з максимальним покриттям коду при мінімальній складності виконання сценарію тестування, що підвищує ефективність процесу тестування та виправлення дефектів програмного продукту, що своєю чергою, дає змогу підвищити його показники надійності.

**Експериментальні результати**

На основі розробленого методу побудови сценаріїв тестування ПЗ було реалізовано програмний продукт, вхідним параметром якого є модель поведінки ПЗ. Було проведено чисельні експерименти, в яких задавались різні характеристики моделі поведінки ПЗ для умовного програмного продукту (кількість методів; кількість змінних з відповідними класами еквівалентності, що використовуються та змінюються методами; ймовірності переходів між методами та інформація про помилки). Для ілюстрації адекватності методу побудови сценаріїв тестування для такої моделі використання було побудовано набори тестових сценаріїв для двох стратегій тестування, які потім використовувались для дослідження залежності кількості відмов ПЗ від різних його характеристик. Результати моделювання поведінки кількості відмов в залежності від кількості тестових сценаріїв з урахуванням різних стратегій наведені на рис.1.

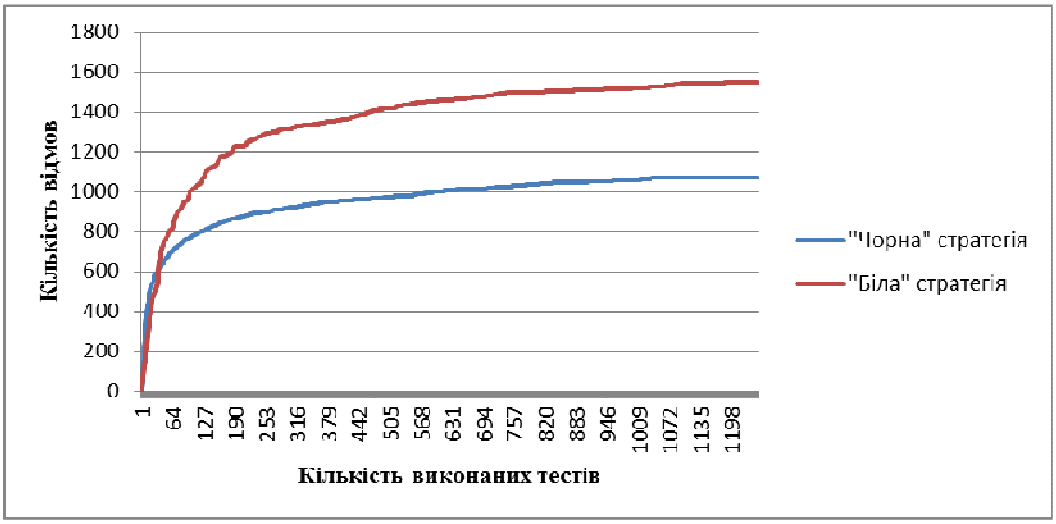


Рисунок 1 – Графік залежності кількості пройдених тестів та кількості зафіксованих відмов двома стратегіями тестування

Як видно з рис.1 під час тестування стратегією «білої» скриньки було знайдено більше відмов ПЗ, ніж під час тестування стратегією «чорної скриньки». Цей факт, а також якісний вигляд залежності відповідає відомим даним тестування реальних програмних продуктів [18, 19], адже тестування стратегією «білої» скриньки враховує особливості архітектури програмного продукту та особливості його помилок.

Графік залежності кількості відмов, що були зафіксовані під час процесу тестування, від кількості методів програмного забезпечення, (рис. 2) ілюструє лінійний зв’язок між ними для усіх стратегій тестування. Такий самий вигляд залежності кількості відмов від об'єму ПЗ був описаний в [19], що підтверджує адекватність запропонованого методу.

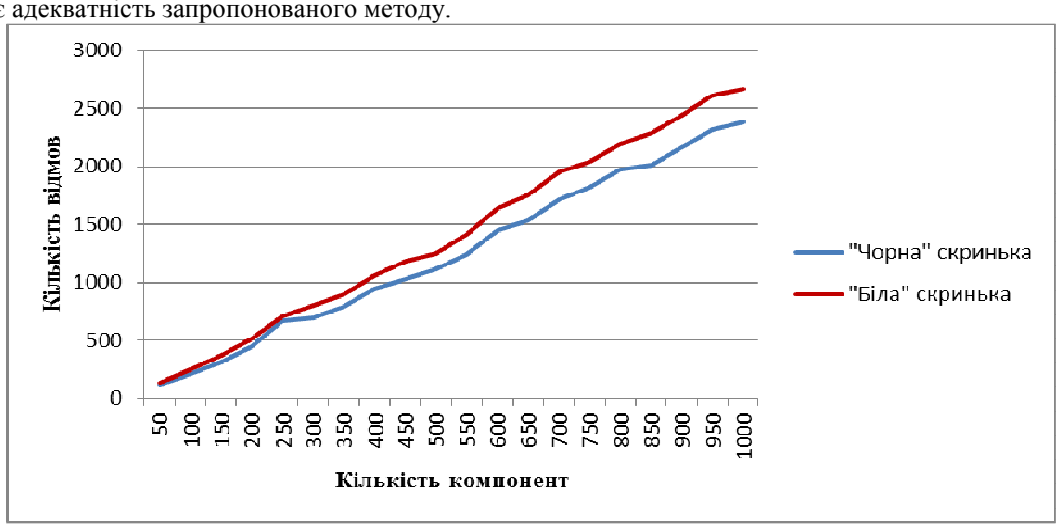


Рисунок 2 – Графік залежності кількості відмов, знайдених двома стратегіями тестування та кількості методів ПЗ

Окрім того проведено дослідження кількості виявлених відмов від середньої кількості змінних, що використовуються у методах програмного продукту (рис. 3). Кількість змінних, що використовуються методом ПЗ, вимірюється відсотком від загальної кількості змінних. Як видно з графіку зі збільшенням кількості таких змінних кількість відмов під час тестування стратегією «білої» скриньки дещо зростає, що обумовлене можливістю аналізувати усі змінні та їхні значення під час побудови тестів, а, отже, виявляти більше відмов. Під час тестування стратегією «чорної» скриньки із ростом кількості змінних, що використовуються методом ПЗ, кількість відмов зменшується. Це пояснюється тим, що стратегія «чорної» скриньки не дає можливості проаналізувати внутрішні змінні програмного продукту, які впливають на перебіг виконання програми, а лише ті, які може змінювати користувач (кількість таких змінних залишається незмінною), тому ефективність цієї стратегії погіршується з ростом середньої кількості змінних, що використовуються методом.

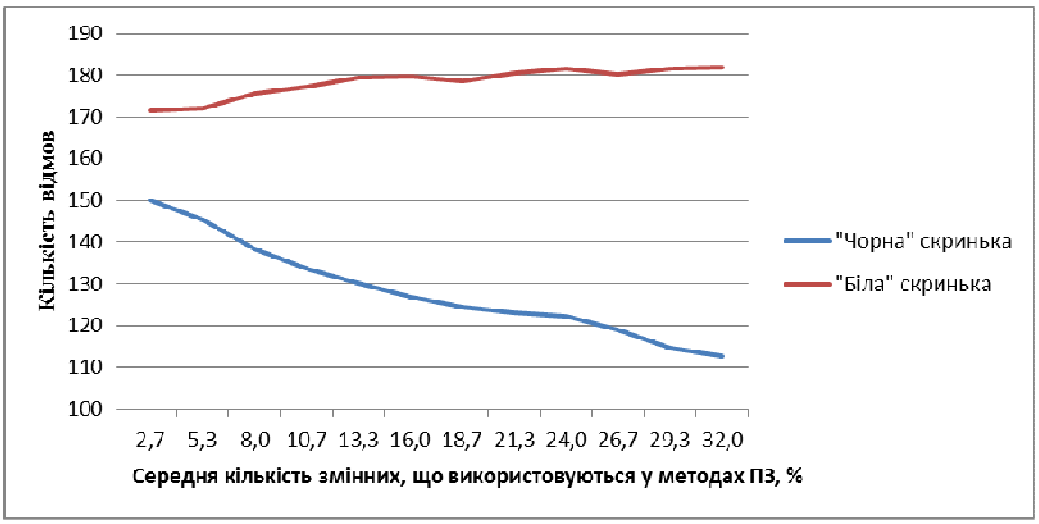


Рисунок 3 – Графік залежності кількості відмов від середньої кількості змінних, що використовуються у методах ПЗ, для двох стратегій тестування.

**Висновки**

У цій роботі розроблено метод автоматизованої побудови сценаріїв тестування програмного забезпечення, вхідним параметром якого є модель поведінки програмного забезпечення на основі його змінних [15], який може бути використаний як для стратегії «чорної», так і «білої» скриньки. Даний метод забезпечує рівномірне покриття коду та підвищує ефективність процесу тестування, зменшуючи часові, фінансові та людські ресурси. Метод побудови тестів реалізований у вигляді програмного продукту, який крім автоматизованої побудови тестових сценаріїв дозволяє змоделювати поведінку ПЗ та моделювати виконання побудованих тестових сценаріїв. З використанням реалізованого ПЗ проведено дослідження впливу характеристик процесу тестування (кількості виконаних тестів) і програмного продукту (кількість компонент ПЗ та середня кількість змінних, що використовуються у методах ПЗ) на кількість виявлених на етапі тестування відмов. Отримані графіки залежностей кількості відмов якісно співпадають з відомими літературними даними, що підтверджує адекватність розробленого методу побудови тестових сценаріїв.

**Список літератури**

1. Broy M. Model Based Testing of Reactive Systems. / M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner. – LNCS 3472, Springer. – 2005. – 659 p.

2. Blackburn M. Why Model-Based Test Automation is Different and What You Should Know to Get Started./ M. Blackburn, R. Busser, A. Nauman // In International Conference on Practical Software Quality. – 2004. – p. 87-90.

3. Legeard B. Controlling Test Case Explosion in Test Generation from B Formal Models. / B. Legeard, F. Peureux, M. Utting // The Journal of Software Testing, Verification and Reliability. – 2004. – no. 14(2). – pp. 81–103.

4. Ebrahim Shamsoddin-Motlagh. A Review of Automatic Test Cases Generation. / Ebrahim Shamsoddin-Motlagh // International Journal of Computer Applications. – 2012. – no. 57(13). – pp. 25-29.

5. Belli F. A Holistic Approach to Testing of Interactive Systems using Statecharts. / F. Belli, C.J. Budnik, A. Hollman // Proceedings of 2nd South-East European Workshop on Formal Methods (SEEFM 05), South-Eastern European Research Center SEERC. – 2005. – pp. 1–15.

6. Hyoung Seok Hong. Automatic Test Generation from Statecharts Using Model Checking / Insup Lee, Oleg Sokolsky, Sung Deok Cha // In Proceedings of FATES'01, Workshop on Formal Approaches to Testing of Software. – 2001.

7. Hu Y.T. Automatic Black-Box Method-Level Test Case Generation Based on Constraint Logic Programming. / Y.T. Hu, N.W. Lin // Computer Symposium (ICS). – 2010. – pp. 977-982.

8. Samuel P. (2008). Automatic test case generation using unified modeling language (UML) state diagrams. / P. Samuel, R. Mall, A. Bothra // The Institution of Engineering and Technology, IET Softw. – 2008. no. 2. – pp. 79–93.

9. Sarma M. Automatic Test Case Generation from UML Models. / M. Sarma, R. Mall // 10th International Conference on Information Technology. – 2007. – pp. 196-201.

10. Sarma M. (2007). Automatic Test Case Generation from UML Sequence Diagrams. / M. Sarma, D. Kundu, R. Mall // 15th International Conference on Advanced Computing and Communications. – 2007. – pp. 60-65.

11. Santiago V. An Environment for Automated Test Case Generation from Statechart-based and Finite State Machine-based Behavioral Models. / V. Santiago, N. Vijaykumar, D. Guimaraes //IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08). – 2008. – pp.63-72.

12. Susumu Fujiwara. Test Selection Based on Finite State Models. / Susumu Fujiwara, Gregor Bochmann, Ferhat Khendek, Mokhtar Amalou, Abderrazak Ghedamsi // IEEE Transactions on Software Engineering. – 1991. – no. 6(17). – pp. 591–603.

13. Whittaker J. A Markov Chain Model for Statistical Software Testing. / J. A. Whittaker, M. G. Thomason // Software Engineering, IEEE Transactions. – 1994. – pp. 812-824.

14. Winfried Dulz. Matelo - Statistical Usage Testing by Annotated Sequence Diagrams, Markov Chains and TTCN-3 / Winfried Dulz, Fenhua Zhen // Third International Conference On Quality Software. – 2003.

15. Fedasyuk D.V. Variables state-based software usage model. / D.V. Fedasyuk, V.S. Yakovyna, P.V. Serdyuk, O.O. Nytrebych // ECONTECHMOD. – 2014 (in publishing).

16. Степанченко И.В. Методы тестирования программного обеспечения: Учебное пособие. / И.В. Степанченко - Волгоград: ВолгГТУ. – 2006. - 74 с.

17. Mohd Ehmer Khan. Different Forms of Software Testing Techniques for Finding Errors. / Mohd Ehmer Khan // International Journal of Computer Science Issues. – 2010. V. 7. – pp. 11-16.

18. Shaik M. Software reliability Growth model with bass diffusion test- effort function and analysis of software release policy. / M. Shaik, R. Shaheda // International Journal of Computer Theory and Engineering. – 2011. – V.3. – pp. 671-680.

19. El Emam K. A methodology for validating software product metrics. / El K. Emam // Tech. rep. NCR/ERC-1076, National Research Council of Canada, Ottawa, Ontario, Canada. – 2000.

**Відомості про авторів**

**Федасюк Дмитро Васильович** – професор, доктор технічних наук, завідувач кафедри програмного забезпечення національного університету «Львівська політехніка».

**Яковина Віталій Степанович** – доцент, кандидат фізико-математичних наук, доцент кафедри програмного забезпечення національного університету «Львівська політехніка».

**Сердюк Павло Віталійович** – доцент, кандидат технічних наук, доцент кафедри програмного забезпечення національного університету «Львівська політехніка».

**Нитребич Оксана Олександрівна** – аспірант кафедри програмного забезпечення національного університету «Львівська політехніка».