УДК 004.415.53

МОДЕЛЬ ОЦІНКИ ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ МЕТОДУ АВТОМАТИЗАЦІЇ

Шоботенко Н.М.
Національний технічний університет України «Київський політехнічний інститут»

Досліджено наступні методи автоматизації тестування: статичний аналіз, динамічний аналіз, інтелектуальна обробка даних, використання UML. Було проведено їх порівняльний аналіз. Запропоновано модель для оцінки ефективності впровадження методу автоматизації.
Ключові слова: керування якістю, забезпечення якості, контроль якості, автоматизація тестування.

Вступ. Недоліки програмного забезпечення є звичайною справою і приводять час від часу до економічних втрат. Зараз організації, що займаються розробкою програм, виділяють більше часу та ресурсів для аналізу та тестування програми як одного цілого, а не незалежних сутностей. Розробники відмічають, що на написання тестового коду витрачається стільки ж часу і коштів, що і на написання самого продукту. Для забезпечення якості програмного забезпечення організації ставлять перед розробниками та тестерами досягнення наступних цілей:

· Визначення джерела помилок швидше та з більшою точністю;

· Виявлення недоліків на ранніх рівнях життєвого циклу розробки програмного забезпечення;

· Знешкодження чим більшої кількості помилок до випуску продукту.

Тестування є необхідною умовою для успішної реалізації програмного продукту, але технології тестування, що зараз використовуються, часто розцінюються як складні, виснажливі, часозатратні і недостатні [1].

Крім того, як окремий випадок програмного продукту, потрібно виділити – веб-додатки, кількість яких зростає кожного дня. В зв'язку з цим виникає проблема автоматизації тестування веб-додатків для забезпечення необхідної якості продукту. В статті представлено чотири підходи, що лише нещодавно почали використовувати при проектуванні та створенні інструментів автоматизації тестування десктопних додатків – це статичний аналіз, динамічний аналіз, інтелектуальна обробка даних та використання UML, – з метою обрання методу, що найбільше підходить для автоматизації тестування саме веб-додатків.

Аналіз програм. В загальному, запропоновано автоматизувати генерацію тестів, використовуючи сучасні досягнення в програмному аналізі, автоматичному управління обмеженнями та зростаючу обчислювальну силу комп'ютерів.

Ключовим технічним моментом є автоматична, керована кодом генерація тестів: маючи програму з вхідними та вихідними параметрами, автоматично генеруємо множину вхідних даних, що під час виконання, пройде стільки станів програми, скільки можливо.

Методи автоматизації генерації тестів, керованої кодом, можна розділити на дві групи : статичний та динамічний [2].

Статичний аналіз програм

Загальна характеристика. Статичний аналіз є популярним засобом пошуку у вихідному або двійковому коді програм певних шаблонів або ситуацій (помилок стилю кодування, порушень проектних угод щодо використання певних бібліотек або властивостей мови програмування, критичних помилок, вразливостей, закладок).[3]

Переваги та можливості. Головна перевага статичного аналізу програм в можливості суттєвого зниження вартості усунення дефекту в програмі тому, що ці інструменти дозволяють виявити значну кількість помилок ще на етапі конструювання. Деякі з них можна застосувати відразу після компіляції програми.

Наступні переваги статичного аналізу коду:

1. Повне покриття коду. Статичні аналізатори перевіряють навіть ті частини коду, що досить рідко отримують управління, та їх, як правило, не вдається протестувати іншими методами. Це дозволяє знаходити дефекти в обробниках помилок чи в системі логування.

2. Статистичний аналіз не залежить від компілятора, який використовується, та середовища, в якому буде виконуватися скомпільована програма. Це дозволяє знаходити приховані помилки, що проявляють себе при зміні версії компілятора чи використанні інших ключів для оптимізації коду.

3. Надає можливість легко і швидко знаходити друкарські помилки та наслідки використання Copy-Paste. Як правило, знаходження цих помилок іншими способами є досить неефективною витратою часу та зусиль.

Недоліки статичного аналізу програм:

1. Статичний аналіз, як правило, слабкий в діагностиці втрати пам'яті та паралельних помилок. Для вияву подібних помилок фактично потрібно віртуально виконати частину програми.

2. Статичний аналіз попереджує про підозрілі місця. Це означає, що насправді код може бути коректним. Зрозуміти, вказує це на помилку чи ні, може лише програміст. [4]

Підходи до реалізації. Статична генерація тестів складається з аналізу програми П шляхом читання програмного коду та використання технік символьного виконання для моделювання виконання абстрактної програми, щоб спробувати вирахувати вхідні дані для проходу П через усі специфічні шляхи виконання чи гілки, навіть без самого виконання програми. Основна ідея посимвольно дослідити дерево усіх обчислень, що присутні в програмі, з усіма можливими заданими значеннями в якості вхідних параметрів.

На жаль, даний підхід не працює, коли програма містить в собі оператори, що викликають обмеження поза межею доступу перевірника обмежень. У цьому випадку статична генерація тестів не може генерувати тестові вхідні дані для проходження програми через будь-яку гілку його умовного оператора. Іншими словами статична генерація тестів показує погані результати, коли не має можливості провести ідеальне символьне виконання. На жаль, на практиці часто використовуються комплексні програмні оператори (маніпуляції з вказівниками, арифметичні операції і так далі) та виклики до операційної системи та бібліотечних функцій, які важко або неможливо виконати символьно з достатньою точністю.

В інструменті автоматизованого тестування Yogi, що комбінує в собі тестування та статичний аналіз, реалізована одна унікальна функція. В один і той же час він проводить подвійний пошук: тесту, що встановить порушення програмою властивості, та абстракції, що доводить протилежне. Якщо абстракція має шлях, що приводить до порушення властивості, Yogi фокусується на генерації тестів по ньому Якщо такий тест не може бути генерований, Yogi використовує інформацію щодо незадоволених обмежень генератора тестів для вдосконалення абстракції. Тобто обмеження тестів та абстракції працюють спільно, використовуючи сліди помилок в абстракції для управління генерацією тестів та обмежень із провалених тестів для управління вдосконаленням абстракцій. [2]

Динамічний аналіз програм

Загальна характеристика. Динамічний аналіз коду – це спосіб аналізу програм безпосередньо при її виконанні. Процес динамічного аналізу можна розбити на кілька етапів – підготовка початкових даних, проведення тестового запуску програми і збір необхідних параметрів і аналіз отриманих даних.

Динамічне тестування найбільш важливе в тих областях, де головним критерієм є надійність програми, час відклику чи споживані ресурси. Це може бути, наприклад, система реального часу, що управляє відповідальною ділянкою виробництва, або сервер бази даних. В таких областях будь-яка допущена помилка може виявитися критичною.

Переваги та можливості:

1. В більшості реалізацій поява помилкових спрацьовувань виключена так, як знаходження помилки відбувається в момент її виникнення в програмі; таким чином, знайдена помилка не є передбаченням, зробленим на основі аналізу моделі програми, а констатацією факту її виникнення;

2. Зазвичай немає необхідності в вихідному коді; це дозволяє протестувати програм із закритим кодом.

З допомогою динамічного тестування можуть бути отримані наступні метрики:

· споживані ресурси – час виконання програми в цілому чи її окремих модулів, кількість зовнішніх запитів, об'єм використаної оперативної пам'яті та інших ресурсів;

· програмні помилки – ділення на нуль, витік пам'яті, «стан гонки».

· наявні в програмі вразливості.

Недоліки динамічного аналізу програм:

1. Динамічний аналіз знаходить дефекти лише на ділянці, визначеній конкретними вхідними даними – дефекти, що знаходяться в інших частинах програми не будуть знайдені;

2. Не може перевірити реалізацію потрібного функціоналу, чи виконує програма усе, передбачене специфікаціями .

3. Потребує значних обчислювальних ресурсів при проведенні тестування;

4. Тільки один шлях виконання може бути перевірений в конкретний момент часу, що потребує великої кількості тестових запусків для більшої повноти тестування;

5. При тестуванні на реальному процесорі виконання некоректного коду може привести до певних проблем. Наприклад, зациклення при виконанні безкінечного циклу. [5]

Підходи до реалізації. *Динамічна генерація тестів* складається з*:*

· виконання програми П, маючи задані чи випадкові вхідні дані;

· збір символьних обмежень входів на умовні оператори протягом виконання;

· використання перевірника обмежень, щоб вивести варіації попередніх вхідних даних для управління наступним виконанням програми по іншій альтернативній гілці.

Цей процес повторюється, поки не досягне певного оператору програми.

Направлене автоматизоване випадкове тестування (НАВТ) є частковим випадком динамічної генерації тестів, що комбінує її з технологіями перевірки моделей для систематичного проходження усіх можливих шляхів, з перевіркою усіх виконань за допомогою інструментів відслідкування різних типів помилок. В НАВТ направлений пошук – кожен новий вектор вхідних даних намагається провести виконання програми по іншому новому шляху. При повторі цього процесу такий пошук намагається провести програму через усі ймовірні шляхи виконання, так само як систематичне тестування та динамічна перевірка програмних моделей.

На практиці направлений пошук великих програм стикається з проблемою недостачі часу для дослідження всіх можливих шляхів. Але навіть так, ми досягаємо кращого рівня покриття, ніж при випадковому тестуванні. А, отже, більше можливостей виявлення помилок.

В реальних програмах неточність при символьному виконанні типово зростає в деяких місцях, і динамічна генерація тестів може відновлювати її. Динамічна більш загальна та потужна в порівнянні з статичною та розширяє її за рахунок додаткової інформації, зібраної під час виконання.

Зокрема, був створений інструмент SAGE, який реалізує розширене тестування білого ящика з використанням динамічної генерації тестів, що не залежить від мови реалізації, бо механізм SAGE працює з машинним кодом, що дозволяє проводити масштабоване, автоматизоване, направлене виконання.

В інструменті PEX динамічна генерація використовується для автоматизованого проектування нових модульних тестів шляхом вирахування множини вхідних параметрів, що перевіряють усі стани та допущення аналізованої програми. Даний підхід отримав назву – параметричне модульне тестування (ПМТ). [2]

Підхід інтелектуальної обробки даних

Загальна характеристика. Розглядається підхід до автоматизації тестування з використанням інформаційної нечіткої мережі (НІМ).

Дана мережа має деревоподібну структуру, де один і той же вхідний атрибут використовується в усіх вузлах даного шару (рівня). Компоненти мережі включають кореневий вузол, змінну кількість прихованих шарів (один шар для кожного обраного вхідного), та цільовий рівень(вихідний), який показує можливі вихідні значення. Кожен ключовий вузол асоціюється зі значенням (класом) в області цільового атрибута. Цільовий рівень не має еквівалента в деревах рішень, але є схожа концепція вузлів в графах рішень. Якщо ж модель передбачає прогнозування значень неперервного цільового атрибуту, то цільові вузли визначені інтервалами, які не перетинаються, в діапазоні атрибуту. Немає обмежень на максимальну кількість вихідних вузлів в даній мережі.

Якщо запустити дану мережу на даних виконання програмної системи, кожна взаємодія між кінцевим та ключовим вузлом дасть ймовірнісний вихід тесту.

Реалізація подібної мережі може мати наступну архітектуру середовища:

· Наслідувана система (НС). Цей модуль представляє програму, компонент або систему, яка буде протестована в наступних версіях програми. Важливо, що даний модуль – керований даними. Він повинен мати добре визначений інтерфейс в термінах заданих вхідних даних та отриманих виходів.

· Специфікація виходів та входів додатку (СВВД). Базова інформація щодо кожної вхідної та вихідної змінної інтерфейсу НС включає назву змінної, тип та діапазон можливих значень. Ця інформація зазвичай доступна зі інструментів управління вимогами та тестами.

· Генератор випадкових тестів (ГВТ). Цей модуль генерує випадкові комбінації значень з діапазону кожної вхідної змінної. Кількість навчальних тестів для генерації визначається користувачем. Генеровані тести в подальшому застосовуються в Тестових каналах та модулі НІМ.

· Тестовий канал (ТК). Інколи цей модуль ще називають «тестова збруя», передає тренувальні випадки від ГВТ до НС. Виконання тренувальних випадків може бути проведене комерційними інструментами автоматизації тестів. Цей же модуль одержує від НС вихідні дані після проходження кожного випадку та повертає до модулю НІМ.

· Нечітка інформаційна мережа(НІМ). Вхідні дані НІМ включають тренувальні тести, генеровані ГВТ, та вихідні, отримані НС після кожного тесту. Крім того, НІМ також використовує описи змінних із СВВД. Алгоритм НІМ повторно виконується для знаходження підмножини релевантних вхідних змінних до кожного виходу і відповідної множини ненадлишкових тестів. Актуальні тести генеруються за автоматично визначеними класами еквівалентності з використанням існуючої тестової політики (один тест з кожного боку еквівалентного класу).

Кожна вихідна змінна представлена відокремленою нечіткою мережею.

Переваги та можливості:

ГВТ одержує лист виходів та входів системи разом із іхніми типами із специфікації системи. Не потребується жодна інформація щодо функціональних вимог, бо алгоритм автоматично визначає відносини входу-виходу зі випадково генерованих тестових випадків.

Алгоритм навчається на вхідних даних, представлених ГВТ, та вихідних, одержаних від НС за допомогою модуля ТК. Як було визначено вище, відокремлена модель будується для кожної вихідної змінної. Наступна інформація може бути отримана із кожної моделі:

1. Множина вхідних атрибутів релевантних до відповідних вихідних.

2. Логічні (якщо ... то) правила, що показують відносини між обраним вхідним атрибутом та відповідним вихідним. Множина правил формується на кожного кінцевому вузлі, відображаючи розподіл вихідних значень. Ці правила можуть бути використані для визначення прогнозу (найбільш ймовірнісного) значення виходу відповідного тестового випадку.

3. Інтервали дискретизації кожної неперервної вхідної змінною включені в мережу. В термінах тестування кожен інтервал представляє собою «клас еквівалентності».

4. Множина ненадлишкових тестів. Кінцеві вузли мережі перетворюють на тести, кожен відображає ненадлишковий зв'язок входу/класів еквівалентності та відповідного розподілу значень виходу. В будь-який час, коли поведінка додатку, який тестується, характеризується деяким стандартним випадком, очікується число тестів, основаних на НІМ, значно меншим від кількості випадкових тестів, що використовували для навчання мережі.

Компактність, властива цим моделям, може допомогти при відновленні найбільш домінантних вимог за даними реалізації і, отже, побудові компактного набору тестів. Іншою важливою властивістю даного алгоритму є стійкість по відношенню до тренувальних випадків, навчання проводиться лише на малих та випадково генерованих підмножинах прийнятних вхідних значень.

Запропонований метод направлений на наступні невирішені проблеми, пов'язані із регресійним тестуванням систем:

· Метод автоматично створює множину ненадлишкових тестів, що покривають найважливіші функціональні відносини, присутні в системі (включно зі відповідними класами еквівалентності)..

· Метод придатний для тестування комплексних програмних систем, бо не залежить від аналізу системного коду на відміну від методів білого ящика автоматизованого вибору тестів.

· Використання методу не потребує істотної підтримки з боку людини. Існуючі методи та технології проектування тестів потребують базового аналізу вимог чи коду.

· Пропущені, застарілі або неповні вимоги не є перепоною для запропонованої методології, бо вона вивчає функціональні взаємовідносини автоматично за даними реалізації. Існуючі методи генерації тестів потребують наявність деталізованих вимог, які можуть бути недоступними або неповними в багатьох системах.

Недоліки підходу IFN:

1. Досить висока точність прогнозу важлива в моделях, якщо планується використання їх як «автоматизовані істини» при регресійному тестуванні, але передбачення «ідеальних прогнозів» усіх вихідних значень в комплексних програмних системах – нереалістично.

2. Для кожної вихідної змінної будується відокремлена модель, створення комплексної може бути напрямком подальших досліджень.

3. Мінімальна відстань до границь впливає на точність прогнозу чисельних рішень. Тому прогноз останньої точки матиме найгіршу точність. За результатами проведених експериментів похибка для останній точці перевищує по іншим майже вдвічі.

4. Так, як розмір тренувальної вибірки визначає користувач, оптимальний розмір визначається шляхом спостерігання за поведінкою похибки. Для випадку 4 входи – 5 виходів оптимальний розмір складає 1000 тестових випадків.

5. Система не потребує знання вимог програми, тому не може перевірити програму на їх задоволення, чи реалізує в повній мірі програма необхідний функціонал, чи робить програма те, що має робити, і не робить те, що немає.

Напрямки розвитку. Подальші дослідження включають створення єдиної комплексної моделі та додатків на основі запропонованої методології для великомасштабних програмних систем, генерації множин тестів та їх скорочення, Наступні експерименти також включатимуть оцінку можливостей методу для знаходження різноманітних типів помилок, що містяться в коді додатку [6].

Використання UML для автоматичної генерації тестів

Загальна характеристика. Правильність функціонування системи визначається відповідністю реальної поведінки системи еталонній поведінці. Для того, щоб якісно визначати цю відповідність,

потрібно вміти формалізувати еталонну поведінку системи. Поширеним способом опису поведінки системи є опис за допомогою діаграм UML (Unified Modeling Language) [7].

UML (англ. Unified Modeling Language) – уніфікована мова моделювання, використовується у парадигмі об'єктно-орієнтованого програмування. Вона є мовою широкого профілю, це відкритий стандарт, що використовує графічні позначення для створення абстрактної моделі системи, називаної *UML-моделлю*. UML був створений для визначення, візуалізації, проектування й документування, в основному, програмних систем. UML не є мовою програмування, але в засобах виконання UML-моделей, як інтерпретованого коду, можлива кодогенерація [8].

Використання даного підходу для автоматизації тестування може мати наступну архітектуру середовища, як запропоновано в [9];

Першим компонентом є системна модель, написана на UML, – набір діаграм класів, станів та об'єктів. Діаграма класів визначає сутності системи, станів – одна для кожного класу – пояснює, як ці сутності можуть розвиватися, діаграма об'єктів специфікує початкову конфігурацію.

Друга компонента, також написана на UML, – тест директива, що містить конкретні діаграми станів та об'єктів; діаграма об'єктів використовується для вираження тестових обмежень та критеріїв покриття; діаграма станів специфікує цілі тестування. Системна модель та тестові директиви можуть бути побудовані з допомогою будь-якого стандартного набору інструментів.

Компілятор бере модель та створює набір станів взаємодії машин, написаних на мові проміжного формату. Представлення кожної машини диктується діаграмою станів моделі, їх взаємодія імітує механізм дія-подія UML.

Переваги та можливості підходу:

1. UML є доступною та простою в користуванні. Крім того, графічне відображення дозволяє краще зрозуміти та відслідковувати специфікації системи на наявність неповноти чи неточностей.

2. UML володіє необхідними набором можливостей та засобів для комбінації достатності та абстрагування відповідної області, тобто є в змозі відобразити властивості, що тестуватимуться. Крім того, UML досить проста та не містить надлишкових конструкцій, ускладнень чи комплексності.

3. Наявність композиційності та масштабованості, що дозволяє комбінувати моделі модулів для створення моделі комплексної системи.

4. Наявність точок варіації в семантиці мови: визначення мови навмисно неповне. Подальша інтерпретація необхідна для моделі, написаної на UML, може бути використана для формального аналізу або автоматичної генерації тестів.

5. Системна модель представлена лише трьома видами діаграм: діаграмою класів, діаграмою об'єктів та діаграмою станів.

6. Дані види діаграм створюють на етапі проектування програмного забезпечення, тобто підхід не вимагає побудови чи створення додаткових чи більш спеціалізованих документів.

7. Так як тести основуються на документах проектування програмного забезпечення, що відображають в собі вимоги та специфікації системи, то вони перевіряють, чи виконує програма усе передбачене та необхідне, чи правильно реагує.

8. При внесенні змін в специфікації програми достатньо замінити застарілі діаграми новими, куди були попередньо внесені зміни щодо появи, зміни чи видалення вимог.

9. Тест директиви складаються із трьох окремих частин: цілей тестування (опис поведінки, що тестується), тестові обмеження та критерії покриття (вимоги щодо покриття множиною тестів, що генеруються). Тобто інструмент володіє усією необхідною інформацією для керування об'ємом множини генерованих тестів.

10. Відповідно до критеріїв покриття будується множина тестів або тестовий граф, що покриває усі досяжні задачі покриття. Для уникнення надлишковості при проведенні випробувань усі цілі тестування та критерії покриття можуть бути пройдені разом.

11. При комбінуванні тестових директив користувач має можливість налаштувати вибір тестів згідно бюджету, виділеного на тестову програму. Ієрархія множини тестів може бути сконструйована згідно принципу: чим більшим є набір тестів, тим більше покриття реалізації, що є корисним при регресійному тестуванні.

Недоліки підходу:

1. Придатність моделі основується на представленій програмі. Зрозуміло, що потрібно включити в модель усю інформацію, що пов'язана з поставленою ціллю, але, крім того, потрібно виключити усю зайву. Модель з надлишком інформації буде заскладною для охоплення та автоматизованої розробки програмного забезпечення.

2. Модель, що підходить для одного випадку, менше підходить для іншого: певна життєво необхідна інформація може бути відсутня. Якщо стоять кілька цілей, можливо, з'явиться потреба кількох різних моделей представлення однієї і тієї ж системи.

3. Автоматизовані тести базуються на документах проектування програмного забезпечення, але виникає проблема пропущених чи неповних вимог. Тобто спроектовані тести перевірятимуть лише те, що представлене діаграмами.

4. Потребує досягнення певного рівня деталізації вимог при їх представленні та відображенні в UML-діаграмах для коректної генерації тестів. [9]

Напрями розвитку. Ціллю представленого підходу є розробка методів та інструментів для автоматизації тестування програмного забезпечення з частковим акцентом на тестуванні розподілених компонентних систем. Як напрямок розробки можлива адаптація підходу для використання при тестуванні веб-додатків.

Порівняльний аналіз. В статті запропоновано провести аналіз чотирьох представлених вище підходів до автоматизації тестування за наступними критеріями:

- доступ до коду програми;
- верифікація чи валідація;
- вихідні дані, трудомісткість;
- види тестування;
- втручання користувача.

Пропонується використати наступну загальну формулу розрахунку рентабельності інвестицій (ROI) для оцінки доцільності використання відповідного методу автоматизації:

$$ROI = \frac{(Gain - Investment)}{Gain} * 100\%$$

де Gain – вартість проведення тестування при використанні лише ручного тестування;

де Investment – вартість створення та виконання бібліотеки скриптів із використанням обраного методу автоматизації.

Для оцінки доцільності використання відповідного методу автоматизації тестування варто врахувати наступне:

1. Витрати на розробку початкової бібліотеки автоматизованих тестів;

Таблиця 1

Порівняльний аналіз підходів

Підходи / Критерії	Статичний аналіз	Динамічний аналіз	Інтелектуальна обробка даних	Використання UML
Доступність коду програми	Відкритий код	Виконуваний файл	Виконуваний файл	Виконуваний файл
Верифікація чи валідація	Верифікація	Більше верифікація	Більше верифікація	Валідація
Вихідні дані та трудомісткість	Код, вимоги, специфікації програми, архітектура	Задані або випадкові вхідні дані	Специфікація виходів і входів додатку	Діаграми станів, об'єктів та класів
Види тестування	System testing, requirement based testing	Performance testing, functional testing, unit testing, system testing, load testing	Functional testing, regression testing, unit testing,	Requirement based testing, GUI testing, load testing, regression testing, system testing
Втручання користувача	Аналіз вимог, специфікацій на наявність неточностей, неповноти	В деяких випадках задання вхідних даних	Визначення розміру тренувальної вибірки	Заміна діаграм при зміні специфікацій

2. Витрати на підготовку і виконання 1го скрипта;
3. Витрати на аналіз результатів першого прогону набору автоматизованих скриптів;
4. Витрати на підтримку автоматизованих тестів у актуальному стані.
5. Кількість запланованих циклів тестування.

Крім того, важливо також врахувати, що у випадку використання інструменту автоматизації вперше, необхідно виділити кошти на власне ліцензію програмного забезпечення, навчання персоналу та купівлю додаткових апаратних засобів у випадку необхідності. [10]

Вартість тестування без автоматизації обчислюється за формулою:

$$Gain = \sum_{i=1}^{k}\sum_{j=1}^{m} t_j \cdot k_j \cdot c_j;$$

де t_j дорівнює 1, якщо j-й вид тестування проводиться, і 0 – у іншому випадку;

k_{ij} – кількість скриптів для проведення j-го виду тестування на i-ому циклі тестування;

c_j – вартість виконання вручну одного скрипта j-го виду тестування;

k – кількість запланованих циклів тестування;

m – кількість запланованих для проведення видів тестування.

Вартість впровадження конкретного методу автоматизації обчислюється за формулою:

$$Investment = \sum_{i=1}^{k}\sum_{j=1}^{m'} t'_j \cdot k'_{ij} \cdot c \ (T_e + T_a + T_m); + T_o \cdot \sum_{j=1}^{m'} t'_j \cdot k'_{ij} \cdot c + I_o;$$

де t_j' дорівнює 1, якщо j-й вид тестування проводиться, і 0 – у іншому випадку;

k_{ij}'– кількість скриптів, що піддаються автоматизації для проведення j-го виду тестування на і-ому циклі тестування;

c – вартість робочої години тестувальника;

k – кількість запланованих циклів тестування;

$\underline{m}$' – кількість видів тестування, що можуть бути автоматизовані з допомогою обраного методу;

I_o – сума витрат на ліцензію програмного забезпечення, навчання персоналу та купівлю додаткових апаратних засобів;

T_o – час розробки початкової бібліотеки тестів, що включає підготовку початкових даних, їх завантаження та власне генерацію автоматизованих скриптів, що потребують уваги тестувальника;

T_e – час підготовки та виконання тестових скриптів, що включає налагодження інструменту автоматизації та саме виконання тестів, що потребують уваги тестувальника;

T_a – час потрібний для аналізу результатів виконання тестів, що включає перегляд результатів та написання тестової документації;

T_m – час потрібний для підтримання бібліотеки в актуальному стані, що включає внесення змін у початкові дані, завантаження змінених даних, повторна генерація тестових скриптів.

Тобто, оцінити доцільність використання відповідного методу автоматизації можна за наступною кінцевою формулою:

Класична формула оцінки ефективності автоматизації:

$$ROI = \frac{\sum_{i=1}^{k}\sum_{j=1}^{m} t_j \cdot k_{ij} \cdot c_j - (\sum_{i=1}^{k}\sum_{j=1}^{m'} t'_j \cdot k'_{ij} \cdot c \ (T_e + T_a + T_m); + T_o \cdot \sum_{j=1}^{m'} t'_j \cdot k'_{ij} \cdot c + I_o)}{\sum_{i=1}^{k}\sum_{j=1}^{m} t_j \cdot k_{ij} \cdot c_j} * 100\%;$$

де T_m – час, витрачений на повністю ручне тестування;

N_r – кількість релізів;

N_c – кількість конфігурацій;

T_d – час, витрачений створення бібліотеки скриптів;

T_s – час, витрачений на підтримку бібліотеки.

Якщо провести порівняння ефективності використання методу автоматизації на прикладі наступного проекту – сайта-візитівки, з проведенням наступних видів тестування(smoke, GUI, functional, regression & usability testing) (табл. 2):

- час на ручне тестування – 28 годин;
- час на створення бібліотеки для всіх методів буде приблизно однаковим так, як генерація відбувається автоматично і потребує незначного втручання користувача;
- час на підтримку бібліотеки включає виконання тестових скриптів, що залежить від кількості скриптів, а не методу автоматизації.

Таблиця 2

Порівняння ефективності використання методів автоматизації за класичною формулою

Показники	Статичний аналіз	Динамічний аналіз	Інтелектуальна обробка даних	Використання UML
T_d	40 год	35 год	32 год	30 год
T_s	10 год	10 год	12 год	10 год
E_a	2.47	2.50	2.14	2.54

Отримані результати незначно відрізняються між собою та не надають змоги визначити ефектив-

ний метод автоматизації тестування відповідного проекту.

Використавши запропоновану формулу отримано результати (табл. 3):

Таблиця 3

Порівняння ефективності використання методу автоматизації за запропонованою формулою

Показники	Статичний аналіз	Динамічний аналіз	Інтелектуальна обробка даних	Використання UML
T_o	40 год	35 год	32 год	30 год
T_e	2 год	2 год	4 год	2 год
T_a	10 год	13 год	15 год	5 год
T_m	10 год	10 год	12 год	10 год
ROI	0,22	0,38	0,15	0,45

· кожен метод забезпечує автоматизацію різної множини видів тестування;

· час на підготовку вхідних даних та налаштування інструменту автоматизації відрізняється;

· приймається до уваги лише час, що потребує наявності тестера;

· вартість години ручного тестування та автоматизації різняться.

Тобто в залежності від специфікації проекту, видів тестування, які потрібно провести, дана формула дозволяє порівняти ефективність використання конкретного методу автоматизації

Висновки по роботі. Було запропоновано наступні критерії порівняння існуючих методів автоматизованого тестування : доступ до коду програми, верифікація чи валідація, вихідні дані, трудомісткість, види тестування, втручання користувача.

При порівнянні обраних підходів отримано, що використання UML для автоматизації володіє кращим потенціалом, бо:

· забезпечує автоматизацію більшої кількості видів тестування,

· потребує менших часових затрат на розробку бібліотеки тестів так, як вхідні дані є результатом етапу проектування і процес генерації може проходити без тестувальника,

· на етапі виконання генерована бібліотека не потребує налаштувань з боку тестувальника та може виконуватися не в робочий час,

· для підтримки актуальності бібліотеки достатньо внести зміни у вхідні дані та згенерувати бібліотеку знову.

Також в роботі вперше запропонована оцінка автоматизованого тестування від підходу до тестування.

Список літератури:

1. Li K., Wu M. Effective Software Test Automation: Developing an Automated Software Testing Tool [Електронний ресурс]. – Режим доступу: http://books.google.com.ua/books/about/Effective_Software_Test_Automation.html?id=noEyxwGQ6SkC&redir_esc=y
2. Godefroid P., de Halleux P., Nori A.V., Rajamani S. K., Schulte W., Tillmann N., Levin M.Y. Automating Software Testing Using Program Analysis [Електронний ресурс]. – Режим доступу: http://research.microsoft.com/pubs/74119/ieeesw2008.pdf
3. Аветисян А., Белеванцев А., Бородин А., Несов В. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ[Електронний ресурс]. – Режим доступу: http://cyberleninka.ru/article/n/ispolzovanie-staticheskogo-analiza-dlya-poiska-uyazvimostey-i-kriticheskih-oshibok-v-ishodnom-kode-programm
4. Статический анализ кода [Електронний ресурс]. – Режим доступу: http://www.viva64.com/ru/t/0046/
5. Динамический аналіз кода [Електронний ресурс]. – Режим доступу: http://www.viva64.com/ru/t/0070/
6. Last M., Friedman M., Kandel A. The data mining approach to automated software testing[Електронний ресурс]. – Режим доступу: http://dl.acm.org/citation.cfm?id=956795
7. Калинов А.Я., Косачёв А.С., Посыпкин М.А., Соколов А.А. Автоматическая генерация тестов для графического пользовательского интерфейса по UML-диаграммам действий[Електронний ресурс]. – Режим доступу: http://cyberleninka.ru/article/n/avtomaticheskaya-generatsiya-graficheskih-polzovatelskih-interfeysov-dostupa-k-integrirovannym-dannym-na-osnove-diagramm-klassov-uml
8. Вікіпедія: UML [Електронний ресурс]. – Режим доступу: http://ru.wikipedia.org/wiki/UML
9. Cavarra A., Davies J., Jeron T., Mounier L., Hartman A., Olvovsky S. Using UML for Automatic Test Generation [Електронний ресурс]. – Режим доступу: https://www.research.ibm.com/haifa/projects/verification/mdt/papers/uml4testgen.pdf
10. Гребенюк В.М. Оценка целесообразности внедрения автоматизированного тестирования[Електронний ресурс]. – Режим доступу: http://naukovedenie.ru/PDF/13tvn113.pdf

Шоботенко Н.Н.
Национальный технический университет «Киевский политехнический институт»

МОДЕЛЬ ОЦЕНКИ ЭФФЕКТИВНОСТИ ИСПОЛЬЗОВАНИЯ МЕТОДА АВТОМАТИЗАЦИИ

Аннотация
Исследованы следующие методы автоматизации тестирования: статический анализ, динамический анализ, интеллектуальная обработка данных, использование UML. Проведен их сравнительной анализ. Предложена модель для оценки эффективности использования метода автоматизации.

Ключевые слова: управление качеством, обеспечение качества, контроль качества, автоматизация тестирования.

Shobotenko N.M.
National Technical University «Kyiv Polytechnic Institute»

TEST AUTOMATION METHOD RATIONALE

Summary
Next test automation methods were investigated: static analysis, dynamic analysis, data mining approach, using UML. These comparative analysis were heid. Test automation method rationale was proposed.
Keywords: quality management, quality assurance, quality control, test automation.

УДК 691.3

ВПЛИВ МОДИФІКАТОРА АМКІРОЗ РМР-3(Р) НА ФІЗИКО-МЕХАНІЧНІ ВЛАСТИВОСТІ ВАЖКОГО БЕТОНУ

Яцинський А.Л., Ужегова О.А.
Луцький національний технічний університет

Проведено техніко-економічний вибір протиморозного модифікатора, яким за вимогами про доступність, якість та ціну придбання виявився Амкіроз РМР-3 (Р). Дослідження бетонів з добавкою даного модифікатора показали, що втрата міцності зразків після обробки заморожуванням/відтаванням значно знижується при збільшенні вмісту добавки до 2,28%. Подальше зростання відсоткового вмісту модифікатора у бетоні не приводить до відчутного покращення його якості.
Ключові слова: бетон, формування, модифікатор, границя міцності, руйнівне навантаження.

Постановка проблеми. Сучасне будівництво підкорює все більш, здавалось, непридатні території для використання. В таких умовах необхідно враховувати специфіку середовища, в якому проводиться будівництво. Промисловість надводного будівництва є новою, хоча різного роду гідромеліоративні споруди будуються давно. Отже, вимоги до якості бетону будуть схожі в обох випадках. В Україні переважна більшість гідромеліоративних споруд працюють вже декілька десятків років. Бетони цих споруд експлуатуються у важких умовах: піддаються зволоженню та висушуванню, зміні температур від −300 до +500 С, тиску води та льоду, і як наслідок, значна частина їх конструкцій потребує заміни, ремонту або відновлення. Через специфіку експлуатації серед найпоширеніших причин руйнування подібних конструкцій у вологому середовищі є дія морозу. Тому актуальною є задача створення бетонів з високими показниками водонепроникності, морозостійкості та, головне, міцності для будівництва гідротехнічних споруд та споруд громадського користування, які знаходяться на воді.

Аналіз останніх досліджень. Для цілеспрямованого підвищення морозостійкості бетонів у світовій та вітчизняній практиці в останні роки використовують спеціальні системи добавок, які, крім того, підвищують його міцність.

Морозостійкість будівельних матеріалів показує, наскільки той чи інший зразок здатний зберігати свої властивості після декількох послідовних циклів заморожування та відтавання. У випадку з бетоном основною причиною його руйнування під час цих процесів стає вода в твердому стані, яка чинить значний тиск на стінки мікротріщин і пор матеріалу [1].

У свою чергу, велика твердість бетону не дає воді вільно розширюватися при замерзанні, тому при тестуванні на морозостійкість у бетоні створюються високі напруги. Руйнування починається з виступаючих частин, а потім продовжується у верхніх шарах і, нарешті, проникає вглиб.

Фактором, який прискорює руйнування бетону, стає також різний коефіцієнт температурного розширення компонентів, з яких складається будівельний матеріал. Це створює додаткову напругу [2].

До невирішеної частини загальної проблеми слід віднести те, що умови виконання робіт та особливості застосування модифікуючих добавок в кожному випадку різні. У технічних описах виробник надає лише загальні вказівки щодо застосування [3]. Працівник, який використовує добавку, зобов'язаний перевірити придатність і можливість її застосування для передбачених цілей. Точну кількість протиморозної добавки та її вид, залежно від завдання, слід підбирати в лабораторії шляхом проведення пробних замісів і експериментів з готовими зразками.

Метою роботи є підвищення міцності і морозостійкості важкого бетону для будівництва гідротехнічних споруд та споруд громадського користування за рахунок оптимізації його складу.

Задачі дослідження:

1) провести техніко-економічний вибір протиморозного модифікатора;

2) експериментально дослідити вплив складу модифікованого важкого бетону на фізико-механічні властивості композиту.

Об'єкт дослідження: важкий бетон рівної високої рухливості, модифікований комплексною добавкою Амкіроз РМР-3 (Р), для зведення бетонних споруд на воді.

Предмет дослідження: закономірності впливу складу модифікованого важкого бетону на його фізико-механічні властивості та морозостійкість.

Методика проведення експерименту.

Ми вибрали протиморозну добавку, яка за своїм хімічним складом не допускає корозії арматури і відчутно не знижує рухливість бетону, а також має помірну роздрібну ціну. Такою модифікуючою добавкою є Амкіроз РМР-3(Р) українського виробництва.

Параметри якості та фізико-механічні характеристики модифікованого важкого бетону визначали