Науковий вісник НЛТУ України
Scientific Bulletin of UNFU
https://nv.nltu.edu.ua
https://doi.org/10.36930/40320412
Article received 26.07.2022 p.
Article accepted 31.08.2022 p.
UDC 004.41
ISSN 1994-7836 (print)
ISSN 2519-2477 (online)
@ Correspondence author
A. V. Franko
andrii.v.franko@lpnu.ua

Р. П. Базилевич, А. В. Франко
Національний університет "Львівська політехніка", м. Львів, Україна

ДЕКОМПОЗИЦІЯ ВХІДНИХ ДАНИХ У ЗАДАЧАХ ГЕНЕРУВАННЯ МОДУЛЬНИХ ТЕСТІВ

Для дослідження можливостей декомпозиції вхідних даних у задачах генерування модульних тестів застосовано методи ізолювання, структурне та функціональне моделювання. Проаналізовано наявні методи та виділено основні стратегії, які застосовуються системами символьного виконання. Описано їх переваги та недоліки. Запропоновано нові методи кластеризації вхідних даних для генерування модульних тестів. Визначено основні кроки для створення моделі кластеризації вхідних даних з використанням засобів символьного виконання. Обґрунтовано застосування коду як основного джерела даних для кластеризації. Охарактеризовано об'єкти, що містяться у вхідних даних або пов'язані з ними та є потенційними сутностями для кластеризації. Визначено функцію як множину операторів мови програмування, одиницею декомпозиції коду програми для генерування модульних тестів. Охарактеризовано її властивості за мірою придатності кількісних і якісних характеристик для створення функції подібності. Обґрунтовано вибір зв'язків з іншими функціями та звертань до даних за межами локальної області видимості функції як основних параметрів кластеризації. Визначено користувацькі типи даних, які використовуються для визначення параметрів і значення, що повертається, як можливі другорядні параметри кластеризації. Сформульовано задачу кластеризації на підставі обраних характеристик для вхідних даних, що використовуються для генерування тестів. Запропоновано методи обчислення міри близькості між окремими функціями, а також між функцією та кластером. Описано практичні методи для обчислення характеристик та сутностей з вхідних даних. Запропоновано подальші дослідження з метою визначення оптимальних значень коефіцієнтів для запропонованої моделі кластеризації та функції розпізнавання для перевірки близькості кластерів до відомих шаблонів, що створить додаткові можливості для генерування тестів.

Ключові слова: модульне тестування; символьне виконання; кластеризація програмного забезпечення; генерування тестів; функція подібності.

Вступ / Introduction

Основною метою генерування модульних тестів є підвищення якості програмного забезпечення шляхом виявлення дефектів і верифікації функціональних характеристик. Сучасні системи символьного виконання здатні згенерувати множину тестів з високим рівнем покриття коду програми. Проте практичне використання отриманих результатів вимагає вирішення таких задач: великої обчислювальної складності та тривалого часу виконання, неможливості регресійного тестування через відсутність у тестів чіткої мети. Для встановлення цілей тесту необхідно мати шляхи отримання додаткової інформації про програмне забезпечення. Основним засобом для досягнення поставленої мети є моделювання системи, що підлягає тестуванню. Для створення таких моделей наявні засоби потребують адаптації до предметної області та навчання на прикладах, що є тривалим процесом. Вирішення задачі на модульному рівні є простішим через поширене використання стандартизованих компонент та повторне використання наявних бібліотек. Це створює можливості для розпізнавання функціонального призначення окремих модулів на підставі їх властивостей. Для вирішення такої задачі необхідна її декомпозиція, що передбачає подання коду, яке буде зручне для аналізу, поділу і розроблення методів і критеріїв. Додатковим результатом стане підвищення ефективності обчислень за рахунок поділу на модулі для генерування тестів.

Об'єкт дослідження – декомпозиція програмного коду для автоматизованого генерування тестів.

Предмет дослідження – методи та засоби декомпозиції коду програмного забезпечення, які визначають об'єкти для автоматизованого генерування тестів.

Мета роботи – розробити метод кластеризації програмного коду для декомпозиції вхідних даних у задачах генерування модульних тестів, який дасть змогу здійснювати моделювання програмної системи, що підлягає тестуванню.

Для досягнення зазначеної мети визначено такі основні *завдання дослідження*:

- встановити зв'язок між задачею декомпозиції вхідних даних і вибором об'єкта тестування, визначити очікувані результати тесту;
- вибрати структурні одиниці для подання коду в задачі декомпозиції;
- ізолювати характеристики для визначення міри подібності

Інформація про авторів:
Базилевич Роман Петрович, д-р техн. наук, професор, кафедра програмного забезпечення. **Email:** rbaz@polynet.lviv.ua; https://orcid.org/0000-0002-7949-1353
Франко Андрій Вадимович, аспірант, кафедра програмного забезпечення. **Email:** andrii.v.franko@lpnu.ua; https://orcid.org/0000-0002-8359-7353

Цитування за ДСТУ: Базилевич Р. П., Франко А. В. Декомпозиція вхідних даних у задачах генерування модульних тестів. Науковий вісник НЛТУ України. 2022, т. 32, № 4. С. 77–83.
Citation APA: Bazylevych, R. P., & Franko, A. V. (2022). Hierarchical model of automated test generation system. *Scientific Bulletin of UNFU*, 32(4), 77–83. https://doi.org/10.36930/40320412

обраних структур;
- вибрати форму і подати код для імплементації декомпозиції з урахуванням ізоляції обраних властивостей;
- обгрунтувати переваги використання кластеризації для вирішення задачі декомпозиції коду програмного забезпечення як підзадачі генерування модульних тестів.

Аналіз останніх досліджень та публікацій. Дослідження в галузі символьного виконання програмного коду використовують декомпозицію. Вони засновуються на графі потоку керування, що відображає множину можливих шляхів виконання програми, та розділенні його на окремі дерева [7]. Цей поділ базований на метриці покриття тестами умов та операторів тестованого коду. Мінімізується кількість дерев для отримання максимального покриття [12]. Цей підхід не враховує структурні особливості програмного забезпечення, такі як функціональне призначення модулів, подібності між об'єктами тестування та інші [7, 12, 16, 21]. Основною метою наявних підходів є ефективне використання обчислювальних ресурсів. Це покращує метрики покриття коду автоматично генерованими тестами та збільшує кількість згенерованих тестів. Альтернативними методами є обмеження символьного виконання окремими модулями з відсіканням інших частин програми [10]. Основною перевагою таких підходів є обмеження розмірності задачі для зменшення навантаження на обчислювальні ресурси. Новітні методи символьного виконання використовують поділ бінарного коду програм на сегменти та генерують тести для кожного сегменту ізольовано [10, 13]. Сегментація є продовженням підходу обмеження розмірності задачі. Тести, згенеровані у такий спосіб, документують наявний стан коду.

Методології сегментації та кластеризації програмного забезпечення надають більше можливостей, ніж використовується засобами символьного виконання [6, 9, 17]. Основна перевага цього підходу – це формування кластерів на підставі подібності, що зменшує обчислювальний час за рахунок повторного використання готових рішень та створення моделі системи, яку можна аналізувати методами штучного інтелекту [9, 17].

Кластеризація програмного забезпечення є добре дослідженою темою [2]. Причиною великої кількості досліджень в області є відсутність універсальних методів кластеризації для кожної задачі [2]. Одним із сучасних методів є "K-Core", що широко використовується для аналізу мереж і програмних систем [5]. Він передбачає використання виродженості графу як основної характеристики кластеризації. Кластеризація використовується для виявлення повторюваного програмного коду [8], оцінки ймовірності наявності дефектів у модулях програмного забезпечення [3, 18], оцінки вибраних метрик програмної системи [2].

Вона також використовується як метод зворотного розроблення складних програмних продуктів. Інженери використовують такі засоби для швидкого ознайомлення з кодом системи, документація якої відсутня або втрачена. Для кластеризації використовують різні види подання коду, зокрема діаграму класів, граф викликів функцій, діаграму об'єктів й інші, що будуються на підставі вихідного коду [20].

Останні дослідження в галузі символьного виконання програмного коду демонструють необхідність створення моделі програми, яка підлягає тестуванню. Для досягнення мети використовують нейронні мережі та інші методи машинного навчання [1, 19]. Це необхідно через обмеження в інтерпретації частин програмного коду та для постановки цілей тестування. Використання кластеризації для моделювання вихідного коду програмного забезпечення може бути простішою альтернативою методам машинного навчання. Також можливе зменшення розмірності вхідних даних за рахунок використання кластерів з виділеними характеристиками замість програмного коду для його моделювання методами машинного навчання в задачах генерування тестів.

Для подолання наявних недоліків символьного виконання, забезпечення масштабування та підвищення ефективності обчислень доцільно розробити нові підходи до ієрархічної кластеризації, які дадуть змогу встановлювати цілі символьного виконання на підставі аналізу отриманих кластерів. Потрібно з урахуванням наявних практик, виділити необхідні сутності та їх характеристики. Отримані результати дадуть змогу розробити методи декомпозиції вхідних даних (програмного коду) для використання на першому рівні запропонованої моделі системи автоматизованого генерування тестів [5]. Отримані у процесі кластеризації дані покращать процес встановлення цілей тестування на другому рівні ієрархічної моделі системи автоматизованого генерування тестів [5].

Результати дослідження та їх обговорення / Research results and their discussion

Використання кластеризації для задач генерування модульних тестів. Вибір сутностей та їх характеристик для проведення кластеризації. Генерування тестів містить п'ять основних задач:

1. Поділ вхідних даних на частини – отримання декількох множин з множини вхідних даних на підставі критеріїв поділу та фільтрації другорядної інформації.
2. Вибір об'єкта тестування – вибір підмножини об'єктів з множини сутностей, вибраних при поділі вхідних даних на частини, для яких буде згенеровано тести.
3. Визначення мети тестування – тест має встановлювати істинність чи хибність набору тверджень внаслідок свого виконання.
4. Генерування вхідних даних тесту – з множини можливих вхідних даних для обраного об'єкта тестування обрати елементи, що будуть відповідати цілям тесту.
5. Генерування очікувань від тестування – вибір елемента з множини можливих вихідних даних, для яких очікуваний результат від дій обраного об'єкта тестування над наданими вхідними даними підтвердить істинність набору тверджень, які є метою тестування.

Кластеризація програмного забезпечення здатна виконати першу та другу задачі генерування тесту. Для її проведення необхідно визначити сутності, які їй підлягають, їх характеристики та функцію подібності. Проблема вибору сутностей полягає у визначенні гранулярності отриманих кластерів. Основним об'єктом тестування є код програмного забезпечення. Можливими одиницями його декомпозиції є команди бінарного коду; інструкції мови програмування; функції; файли, що містять множини функцій; директорії, що містять множин файлів. Декомпозиція на рівні інструкцій бінарного коду чи мови програмування є складною задачею, оскільки програми містять велику кількість таких елементів. Через обмежений синтаксис та часте повторення імплементація функції подібності має враховувати їх розміщення у структурних одиницях програмного забезпечення, тому необхідно використовувати елементи декомпозиції вищого порядку. Кількість функцій є мен-

шою на один-два порядки, що залежить від стилю програмування. Функція є унікальною комбінацією множини інструкцій (функції, що мають повністю ідентичний набір інструкцій є надлишковістю та поганим дизайном) та декларації (назва, вхідні та вихідні параметри). Вони можуть бути використаними як одиниці декомпозиції в задачі тестування. Файли, як одиниця декомпозиції, не можуть бути об'єктом тесту, оскільки файли не є одиницею функціоналу, функція є доцільною одиницею декомпозиції для задачі генерування модульних тестів. Для кластеризації необхідно визначити набір характеристик її об'єктів. Кожна функція належить до певного файлу, має набір вхідних і вихідних параметрів, набір інструкцій, може викликати інші функції або бути викликаною іншими функціями, модифікувати чи читати дані у глобальній області видимості. Вибір характеристик має відповідати таким критеріям: алгоритмічна складність є близькою до лінійної, достатній рівень інформації про систему для можливості генерування цілей тестування, давати можливість створити зручне графічне подання кластерів для людського сприйняття. Назва функції не може використовуватися як характеристика, оскільки для назви можливий лише семантичний аналіз. Такий аналіз може мати неоднозначні результати, вони можуть бути подібними, що не гарантує подібність двох сутностей. Назви функцій цілком залежать від людського фактору та мають високу варіативність. Їх використання як характеристики не є доцільним. Використання інформації про вхідні дані та значення, що повертає функція, вимагає опрацювання типів і назв вхідних параметрів. Стандартні типи даних є повторюваними, тому вони можуть бути лише другорядною характеристикою, а їх назви вимагають семантичного аналізу. Подібність вхідних і вихідних параметрів продемонстровано на рис. 1. Використання користувацьких типів даних є обмеженим та може бути використано як характеристика для кластеризації. Як показано на рис. 2, спільне використання користувацьких типів даних означає зв'язок між функціями та дає підстави вважати функції подібними або пов'язаними.

```
3  int calculate_crc(void * data, int size)
4  int transmit_data(void * data, int size)
5  int process_massage(void * data, int size)
6  int get_data_from_buffer(void * data, int size)
7  int write_data_to_file(void *data, int size)
```

Рис. 1. Повторюваність використання стандартних типів даних на прикладі мов C/C++ / Example of common usage of the standart data types for C/C++ language

```
typedef struct msg
{
    void * data;
    int    size;
} msg_s_t;

void process_msg(msg_s_t * msg);
void send_message(msg_s_t * msg);
```

Рис. 2. Використання користувацьких типів даних є ознакою зв'язку між функціями / User data types can be characteristic for clusterization as they indicate relation between functions

Використання множини інструкцій як характеристики вимагає складних алгоритмів порівняння. Вона містить арифметичні вирази та умови, для визначення подібності яких необхідний аналіз з алгоритмічною складністю, більшою за лінійну. Важливими для аналізу є окремі інструкції, що викликають інші функції або взаємодіють з даними за межами локальної області видимості об'єкта. Отримана інформація відображає структурні зв'язки та дає змогу побудувати графи, як графічну модель системи. Множину викликів функцій можна кластеризувати [11], проте подібний поділ не буде містити достатньо даних для тестування. Поділ на кластери має враховувати звертання до глобальних змінних, які описують внутрішній стан коду. Врахування подібних відношень дасть змогу генерувати тести, що будуть перевіряти зміну внутрішнього стану програми. На рис. 3 подано приклади зв'язку між даними та функціями, які необхідно врахувати при кластеризації. Операції з елементами структур даних можуть бути об'єднані в окремі кластери. Врахування зв'язку зі станами (змінними, що існують в глобальній області видимості відносно функції) дасть змогу об'єднати їх в один кластер. У такому разі відповідні тести для наповнення структур даних і читання їх вмісту можуть бути згенеровані.

Рис. 3. Зв'язок між функціями та даними, що може бути використаний для встановлення цілей тестування / Relationship between the functions and data, which can be used for setting test goals

На рис. 4 зображено властивості функції, що можуть бути використані для кластеризації з метою подальшого генерування модульних тестів. Зеленим позначено параметри, що мають перешочергове значення для кластеризації, жовтим – другорядні, червоним – характеристики, що не будуть використані у визначенні подібності для кластеризації. Виділення цих характеристик дасть змогу створити методи декопозиції програмного забезпечення, які є адаптованими до задачі автоматизованого модульного тестування.

Кластеризація програмного забезпечення на підставі обраних характеристик. Код – це множина I, якій належать інструкції від i_1 до i_n. Вони відповідають одному елементу множини F, що включає функції від f_1 до f_n. Кожній з них належить дві множини C_{calls} та $C_{callers}$, що містять відображення зв'язків між членами F. Звертання до даних відображаються двома множини D_{read} та D_{write}, що місять інформацію про взаємодію функцій зі змінними за межами локальної області видимості. Для кожної інструкції, що належить f_1, буде використана процедура оброблення, яка у разі виявлення виклику функції f_2 додає функцію f_1 у множину $C_{callers}$ (для f_2), а функцію f_2 – у множину C_{calls} (для f_1). Якщо виявлено звертання до даних за межами локальної видимості функції, то відомості про це додаються в множину D_{read}, якщо це читання, та в множину D_{write}, якщо це запис.

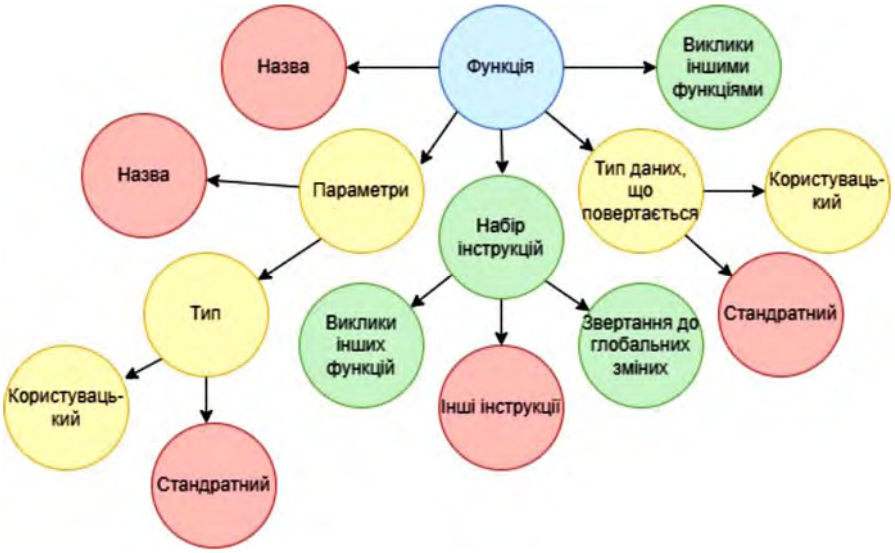

Рис. 4. Зображення функції (синім кольором), як основної сутності для кластеризації, та її основних властивостей (зеленим – характеристики, що будуть використані для кластеризації, жовтим – другорядні, що є менш важливими, червоним – що будуть відкинуті) / Visualization of function as a primary clusterization entity and its characteristics (green – the charecteristics that will be used for clusterization, yellow – secondary characteristics that are less significant, red – characteristics that are not important)

Формула (1) описує функцію подібності *s*, а формула (2) –міру подібності *d* між кластером *U* та функцією f_0.

$$s(f_1, f_2) = k_{calls} \cdot \left(|C^1_{calls} \bigcap C^2_{calls}| + |C^1_{callers} \bigcap C^2_{callers}| \right) + \\ + (k_{data} \cdot (|D^1_{read} \bigcap D^2_{read}| + |D^1_{write} \bigcap D^2_{write}|) \right); \quad (1)$$

$$d(U, f_0) = \sum_{i=1}^{|U|} \frac{s(f_i, f_0)}{|U|}. \quad (2)$$

Для розподілу на кластери необхідно визначити поріг подібності між функцією та кластером *T*, якщо подібність більша за поріг, то функція може бути додана до кластеру, для якого міра близькості найбільша. Якщо міра близькості для усіх наявних кластерів є меншою за визначений поріг, то потрібно виділити новий кластер. Коефіцієнти k_{calls} та k_{data} дають змогу балансувати між різними характеристиками та вносять можливість адаптації моделі до аналізу різних програмних систем.

Методи опрацювання коду для кластеризації програмного забезпечення. Кластеризація програмного забезпечення вимагає встановлення структурних зв'язків між окремими функціями. На підставі структурних зв'язків будується напрямлений граф, який є тотожним до графу викликів. Для побудови графу викликів для коду мовою C/C++ пропонується використовувати засіб cflow. Приклад результатів роботи утиліти подано на рис. 5.

Отримана інформація відображає статичну структуру коду. Її необхідно доповнювати методами динамічного аналізу коду для врахування поліморфізму у програмних системах. Побудований граф викликів функцій дає змогу використати методи ієрархічної кластеризації, базовані на послідовному обході графу та розподілу його вузлів до кластерів. Для встановлення зв'язку між функціями та даними достатньо використати лінійний синтаксичний аналіз. У разі якщо ідентифікатор неоголошений як параметр чи локальна зміна – це є доступом до даних за межами локальної області видимості. Візуалізовані результати оброблення даних для кластеризації за наведеними вище принципами та на підставі обраних характеристик подано на рис. 6, на якому графічно відображено структурні зв'язки між елементами, обраними для кластеризації та їх основними характеристиками на підставі вихідного коду гри в шахи. Використаний код отримано з відкритих джерел [15]. Синім кольором відображено функції, що підлягають кластеризації, фіолетовим – функції, що є зовнішніми інтерфейсами відносно коду, що підлягає кластеризації та червоним дані поза межами локальної області видимості функцій.

```
corovaner@DESKTOP-lTEDBN5:~/test2/json-parser$ cflow json.c examples/test_json.c
main() <int main (int argc, char **argv) at examples/test_json.c:121>: fprintf()
    stat() malloc()
    fopen() fclose()
    free() fread()
    printf()
    json_parse() <json_value *json_parse (const json_char *json, size_t length) at json.c:996>:
        json_parse_ex() <json_value *json_parse_ex (json_settings *settings, const json_char *json, size_t length,
            char *error_buf) at json.c:255>: memcpy()
            default_alloc() <void *default_alloc (size_t size, int zero, void *user_data) at json.c:105>:
                calloc()
                malloc()
            default_free() <void default_free (void *ptr, void *user_data) at json.c:lll>:
                free()
                sprintf()
                string_add()
            hex_value() <unsigned char hex_value (json_char c) at json.c:72>:
                isdigit()
            new_value() <int new_value (json_state *state, json_value **top, json_value **root, json_value **alloc,
```

```
            json_type type) at json.c:131>: json_alloc() <void *json_alloc (json_state *state, size_t size, int
            zero) at json.c:117>:
         isdigit()
         would_overflow() <int would_overflow (json_int_t value, json_char b) at json.c:88>:
         pow()
         strcpy()
         json_value_free_ex() <void json_value_free_ex (json_settings *settings, json_value *value) at
         json.c:1002>:
exit()
   process_value() <void process_value (json_value *value, int depth) at examples/test_json.c:85> (R):
   print_depth_shift() <void print_depth_shift (int depth) at examples/test_json.c:48>:
      printf()
   printf()
   process_object() <void process_object (json_value *value, int depth) at examples/test_json.c:58> (R):
      print_depth_shift() <void print_depth_shift (int depth) at examples/test_json.c:48>:
         printf()
      printf()
      process_value() <void process_value (json_value *value, int depth) at examples/test_json.c:85>
         (recursive: see 30)
      process_array() <void process_array (json_value *value, int depth) at examples/test_json.c:72> (R):
         printf()
   process_value() <void process_value (json_value *value, int depth) at examples/test_json.c:85> (recursive:
      see 30)
json_value_free() <void json_value_free (json_value *value) at json.c:1052>:
   default_free() <void default_free (void *ptr, void *user_data) at json.c:111>:
```

Рис. 5. Робота утиліти cflow для бібліотеки оброблення Json файлів / Usage of cflow to build a call graph for Json library

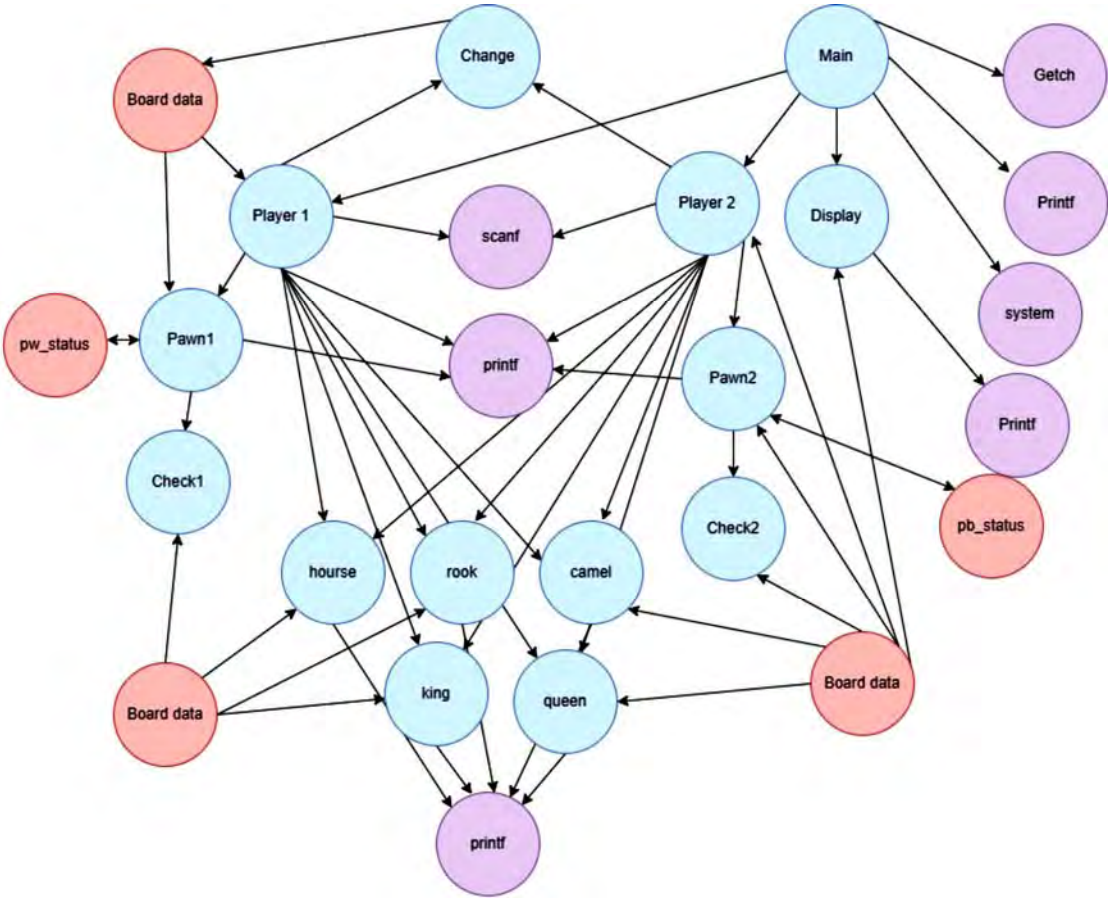

Рис. 6. Візуалізація графу для кластеризації програмного забезпечення, побудованого на підставі коду для гри в шахи / Graph visualization of a software for chess game prepared for clusterization [15]

Обговорення результатів дослідження. Запропоновано підхід до декомпозиції вхідних даних задачі генерування тестів на підставі наявних методологій кластеризації [6, 9, 17]. На відміну від наявних систем [7, 10, 12, 13, 16, 21], декомпозиція відбувається на підставі функції як структурної одиниці замість інструкцій мови програмування. Це дає змогу побудувати модель, розділену на кластери, замість поділу графу контролю потоку на дерева [7, 12, 16, 21] та зменшує розмірність задачі порівняно із [13]. Розроблені методи продовжують тенденцію до ширшого використання кластеризації [13] в задачах генерування тестів для подолання недоліків наявних методів і розпізнавання характеристик системи [17]. На підставі розроблених підходів доцільно створювати алгоритми кластеризації вихідного коду програмного забезпечення. Актуальним є визначення оптимальних значень для коефіцієнтів функції подібності в наступних дослідженнях. Її наявність дає змогу створити методи розпізнавання стандартних конструкцій для генерування додаткових тестів, що є темою для подальших досліджень.

Отже, за результатами виконаної роботи можна сформулювати такі наукову новизну та практичну значущість результатів дослідження.

Наукова новизна отриманих результатів дослідження – розроблено метод, що поділяє програмний код на кластери та створює можливості його моделювання за характеристиками, важливими для задачі генерування тестів.

Практична значущість результатів дослідження – розроблений метод сприятиме підвищенню кількості виявлених дефектів у процесі тестування програмного забезпечення та економії обчислювальних ресурсів у задачах генерування модульних тестів.

Висновки / Conclusions

Розроблено метод кластеризації програмного коду для декомпозиції вхідних даних у задачах генерування модульних тестів, який дає змогу здійснювати моделювання програмної системи, що підлягає тестуванню.

1) Відзначено особливості наявних методів декомпозиції для забезпечення метрик покриття усіх інструкцій та умов тестами. Вказано на обмеження такого підходу та відсутність цілей згенерованих тестів, що не дає можливості проводити регресійне тестування після внесення змін у код.
2) Запропоновано на підставі наявних методів кластеризації програмного забезпечення розробити аналогічні для декомпозиції вхідних даних у задачі генерування модульних тестів.
3) Обґрунтовано необхідність використання функції як одиниці кластеризації для вхідних даних задачі генерування модульних тестів. Інструкції та бінарний код є часто повторюваними, залежать від попередніх інструкцій, а файли натомість є занадто великими та не можуть бути об'єктом тестування.
4) Виокремлено такі характеристики функції, що мають бути враховані під час кластеризації: структурні зв'язки з іншими функціями (виклики та їх спрямованість), структурні зв'язки з даними, що знаходяться за межами локальної області функції, користувацькі типи даних, що є параметрами функції. Ці характеристики забезпечують локальність кластерів завдяки структурними зв'язкам, лінійну обчислювальну складність аналізу, є простішими за інші можливі характеристики.
5) Сформульовано підхід до кластеризації на підставі порівняння міри подібності виокремлених характеристик.
6) Запропоновано проведення подальших досліджень для виявлення оптимальних параметрів кластеризації, для отримання максимально точного функціонального відображення системи та використання додаткової інформації із сформованої кластерної моделі для генерування тестів.

References

1. Aichernig, B. K., Bloem, R., Ebrahimi, M., Tappler, M., & Winter, J. (2018). "Automata Learning for Symbolic Execution," 2018 Formal Methods in Computer Aided Design (FMCAD), 1–9. https://doi.org/10.23919/FMCAD.2018.8602991. https://doi.org/10.23919/fmcad.2018.8602991
2. Alsarhan, Q., Ahmed, B. S., Bures, M., & Zamli, K. Z. (2020). Software module clustering: An in-depth literature analysis. IEEE Transactions on Software Engineering. https://doi.org/10.1109/tse.2020.3042553
3. Arshad, A., Riaz, S., Jiao, L., & Murthy, A. (2018). Semi-supervised deep fuzzy c-mean clustering for software fault prediction. IEEE Access, 6, 25675–25685. https://doi.org/10.1109/access.2018.2835304
4. Bazylevych, R. P., & Franko, A. V. (2021). Hierarchical model of automated test generation system. *Scientific Bulletin of UNFU*, *31*(5), 96-101. https://doi.org/10.36930/40310515
5. Bazylevych, R. P., & Franko, A. V. (2021). Ierarkhichna model sistem avtomatizovanogo generuvannia modulnikh testiv. *Scientific Bulletin of UNFU*, *31*(5), 96–101. https://doi.org/10.36930/40310515
6. Bazylevych, R. P., Melnyk, R. A., & Rybak, O. G. (2000). Circuit partitioning for FPGAs by the optimal circuit reduction method. *VLSI design*, *11*(3), 237–248. https://doi.org/10.1155/2000/58485
7. Bucur, S., Ureche, V., Zamfir, C., & Candea, G. (2011). Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*, 183–198. https://doi.org/10.1145/1966445.1966463
8. Capiluppi, A., Di Ruscio, D., Di Rocco, J., Nguyen, P. T., & Ajienka, N. (2020). Detecting java software similarities by using different clustering techniques. Information and Software Technology, 122. https://doi.org/10.1016/j.infsof.2020.106279
9. Chen, M., & Mishra, P. (2010). Functional test generation using efficient property clustering and learning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *29*(3), 396–404. https://doi.org/10.1109/tcad.2010.2041846
10. Chipounov, V., Georgescu, V., Zamfir, C., & Candea, G. (2009). Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability* (*HotDep*) (No. CONF).
11. Gharibi, G., Alanazi, R., & Lee, Y. (2018). Automatic hierarchical clustering of static call graphs for program comprehension. In *2018 IEEE International conference on big data* (*Big Data*), 4016–4025. https://doi.org/10.1109/bigdata.2018.8622426
12. Li, Y., Su, Z., Wang, L., & Li, X. (2013). Steering symbolic execution to less traveled paths. *ACM SigPlan Notices*, *48*(10), 19–32. https://doi.org/10.1145/2544173.2509553
13. Ma, R., Gao, H., Dou, B., Wang, X., & Hu, C. (2019). Segmental Symbolic Execution Based on Clustering. In 2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), 1289–1296. https://doi.org/10.1109/smartworld-uic-atc-scalcom-iop-sci.2019.00239
14. Pan, W., Li, B., Liu, J., Ma, Y., & Hu, B. (2018). Analyzing the structure of Java software systems by weighted K-core decomposition. Future Generation Computer Systems, 83, 431–444. https://doi.org/10.1016/j.future.2017.09.039
15. Pedhadiya, Shreeji. "Chess-In-C." *GitHub*, 28 May 2022, github.com/shreejilucifer/Chess-In-C. Accessed 24 July 2022.
16. Rakadjiev, E., Shimosawa, T., Mine, H., & Oshima, S. (2015). Parallel SMT solving and concurrent symbolic execution. In *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 3, 17–26. https://doi.org/10.1109/trustcom.2015.608
17. Sebastio, S., Baranov, E., Biondi, F., Decourbe, O., Given-Wilson, T., Legay, A., & Quilbeuf, J. (2020). Optimizing symbolic execution for malware behavior classification. *Computers & Security*, *93*. https://doi.org/10.1016/j.cose.2020.101775
18. Shakhovska, N., Yakovyna, V., & Kryvinska, N. (2020). An improved software defect prediction algorithm using self-organizing maps combined with hierarchical clustering and data preprocessing. In International Conference on Database and Expert Systems Applications, 414–424. Springer, Cham. https://doi.org/10.1007/978-3-030-59003-1_27
19. Shen, S., Shinde, S., Ramesh, S., Roychoudhury, A., & Saxena, P. (2019). Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints. In NDSS. https://doi.org/10.14722/ndss.2019.23530
20. Shtern, M., & Tzerpos, V. (2012). Clustering Methodologies for Software Engineering. Advances in Software Engineering, 1–18. https://doi.org/10.1155/2012/792024
21. Siddiqui, J. H., & Khurshid, S. (2010). ParSym: Parallel symbolic execution. In *2010 2nd international conference on software technology and engineering*, vol. 1, V1-405. https://doi.org/10.1109/icste.2010.5608866

R. P. Bazylevych, A. V. Franko
Lviv Polytechnic National University, Lviv, Ukraine

HIERARCHICAL MODEL OF AUTOMATED TEST GENERATION SYSTEM

The research aims to enhance the decomposition methods for the input data in automatic test task by searching for the new decomposition principles. The research uses isolation, structural and functional modeling to propose an advanced clusterization model for the source code of software under testing. Some issues of current approaches to input data partitioning in symbolic execution software are discussed. Consequently, the strong and weak points are highlighted. Furthermore, code decomposition methods for recovering software design from source code are compared to ones that symbolic execution systems use. The formed hypothesis assumes that approaches and principles which are used to obtain the "knowledge" about software could be utilized to generate better input data partition for test generation systems. The paper describes the benefits of advanced decomposition in automated unit testing. According to the analysis, the source code is the primary object to obtain the data for clustering algorithms. The research highlights the possible entities which could be a base node for clusterization and provides arguments for using function as one. The paper determines the characteristics set to calculate the distance between the generated clusters and determine elements that should be combined into a cluster. The research proposes to use caller and callee relationship between functions and write/read between function and data outside of it local scope as primary characteristics for clusterization. If argument or return type of a function is user defined, then it may be used as a secondary parameter. The paper describes the model to calculate the distance between the functions or function and clusters. This model could be used to construct the clusterization algorithms for the input data of test generation systems. The benefits of proposed solution are a possibility to use the defined characteristic as a source of additional information to generate the test purposes and set goals for symbolic execution and potential optimization of computations due to high similarity of elements in clusters.

Keywords: unit testing; symbolic execution; software clusterization; test generation; similarity function.