УДК 004.4

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ АВТОМАТИЧНОГО ГЕНЕРУВАННЯ ТЕСТОВИХ ДАНИХ ДЛЯ ДИНАМІЧНО ТИПІЗОВАНИХ МОВ ПРОГРАМУВАННЯ

Шерстюк Ю.Ю.[1)], Порплиця Н.П.[2)]
Західноукраїнський національний університет
[1)] магістрант; [2)]к.т.н., доцент

I. Постановка проблеми

Розробка програмних продуктів є конкурентною діяльністю, а час, доступний для виведення продукту на ринок, часто обмежений. Крім того, складність та розмір програмних систем зростають у останні роки. Щоб уникнути невдачі на ринку, важливо також досягти високої якості. Разом ці тенденції ставлять великі вимоги до розробників ПЗ: вони повинні розробляти більші системи та швидше. Це особливо викликає складнощі заходів спрямованих на підвищення якості, наприклад, тестування програмного забезпечення [1-3].

Пошукове тестування програмного забезпечення (Search-basedsoftwaretesting- SBST) може зменшити час і зусилля, автоматично генеруючи відповідні та достатні тестові випадки. SBST було успішно застосовано в структурному тестуванні. Хоча попередні дослідження показали застосовність SBST для генерації тестових даних для структурного тестування, вони часто обмежувалися простими типами вхідних даних, такими як числові значення [4-6]. Числа є дуже поширеними вхідними даними, але є багато інших типів вхідних даних, які часто використовуються, особливо в об'єктно-орієнтованих програмах. Часто параметри є об'єктами, які самі зберігають внутрішній стан, або складні та складені структури даних, які вимагають належної ініціалізації даних. У таких ситуаціях генерація тестових даних для досягнення певної мети стає набагато складнішою, ніж для простих числових значень [7].

Іншим аспектом є генерація тестових даних для динамічних мов програмування, які стали популярними в останні роки. Динамічна мова - це мова програмування високого рівня, яка дозволяє програмі змінювати свою поведінку динамічно під час виконання. Часто вони не обмежуються на типі об'єктів, які передаються в якості аргументів або створюються під час викликів методів. Таким чином, можна змінювати частини програми, поки система все ще працює, і легко адаптувати програмні системи [8, 9]. Крім того, динамічні мови часто розглядаються як дуже гнучкі та продуктивні. У попередніх дослідженнях були розроблені інструменти SBST, переважно для статично типізованих мов, таких як C/C++ та Java. На нашу думку, SBST ще мало застосовувався до динамічних мов програмування, і актуальною є задача, як SBST може бути адаптований для цього [9, 10].

II. Мета роботи

У статті запропоновано RTG (Ruby Testcase Generator), інструмент, написаний мовою Ruby, який може створювати тестові випадки для вихідного коду Ruby. Метою роботи, відповідно, є підвищення ефективності автоматичного генерування тестових даних для досягнення повного покриття операторів. У статті приділено увагу двом аспектам:

- застосування SBST для динамічних мов програмування для генерації тестових даних;
- засоби генерувати різноманітних тестових вхідних даних, таких як об'єкти та складні структури даних.

III. Проектування програмної системи

У цьому розділі представлено інструмент RTG, автоматичний генератор тестових даних для Ruby. На рисунку 1 показано основну структуру та основні компоненти інструменту. Робота системи починається з отримання вихідного коду (Sourcecode) та обраного класу для тестування (CUT). Маючи ці передумови, аналізатор (Analyser) динамічно завантажує CUT та шукає корисну інформацію, необхідну для генератора тестових даних (Testcase generator). Генератор тестових даних формує набір тестових індивідів, формуючи послідовність створення об'єктів та викликів методів. Вхідні дані для параметрів генеруються за допомогою генератора даних(Data input generator). Після повної ініціалізації окремих тестових даних вони виконуються на програмі, яку тестують (Test caseexecutor). З цього генератор тестових даних отримує зворотний зв'язок, який впливає на подальші кроки пошуку. Після досягнення цілі пошуку інструмент постачає набір сценаріїв тестових даних (Testscenario).

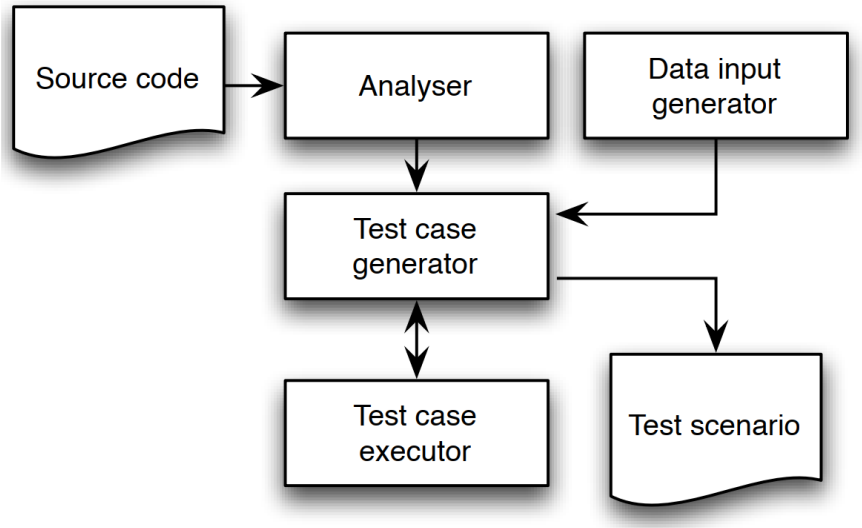

Рисунок 1 – Структура автоматичного генератора тестових даних

Аналізатор. Аналізатор видобуває інформацію, яка використовується пізніше в процесі генерації та адаптації тестових даних. Оскільки Ruby є рефлексивною та динамічною мовою, аналізатор зосереджується на виконанні своєї задачі під час виконання та виконує лише мінімальний аналіз статичного коду. Це означає, що CUT динамічно завантажується в систему та досліджується. Аналізатор надає об'єкт CUTInfo, який містить інформацію про конструктор та його аргументи. Крім того, він зберігає список методів, які оголошені в межах цього класу-цілі. Кожен метод стає методом під тест (MUT), і, таким чином, пов'язується з об'єктом MUTInfo. Цей об'єкт, з свого боку, містить інформацію про MUT, таку як його список аргументів та методи, викликані для кожного окремого аргумента. Крім того, він відстежує покриття, досягнуте під час процесу пошуку, крім адекватних або відкинутих генераторів даних.

Генератор даних. Генератор даних створює вхідні значення, які передаються як аргументи для викликів методів. Знаходження відповідних даних є дуже важливою, але важкою задачею. Існує великий набір можливих типів вхідних даних, і область значень вхідних даних для певного типу може бути досить великою. Оскільки одному генератору даних важко охопити всі різні можливості, основне рішення проекту полягає у наявності різних генераторів для конкретних проблем. Це надає користувачу можливість визначати нові генератори, які можуть створювати дані, що відповідають контексту. Таким чином, набір генераторів даних повинен бути змінюваним, додаванням або вилученням визначених користувачем генераторів. Однак це потребує загального спільного інтерфейсу, такого, що програма може самостійно працювати, не змінюючи свою поведінку для кожного генератора. Мати специфічні для проблеми генератори даних поліпшує також пошук відповідних значень тесту. Припустимо, що у нас є метод, який приймає рядок як вхід і перевіряє, чи є він дійсним ISBN-кодом чи ні. Якщо ми застосуємо простий генератор рядків, який випадковим чином створює будь-яку послідовність символів, то буде дуже важко, якщо не неможливо, знайти дійсний рядок ISBN. З іншого боку, якщо ми визначимо генератор рядків, який виробляє лише значення згідно з попередньо визначеним шаблоном, то шанс успіху у знаходженні дійсного вхідного значення збільшиться.

Виконавець тестових даних. Згенеровані тестові випадки виконуються для отримання інформації, такої як покриття коду. Виконавець тестових даних відстежує покриття, досягнуте попередніми тестовими сценаріями. Таким чином, можливо визначити, чи сприяє поточний тестовий випадок покриттю коду, і, отже, чи він стане частиною кінцевого набору тестових сценаріїв. Тестові випадки поділяються на три основні частини, а саме конструктор, послідовність викликів методів для зміни стану об'єкта та виклик поточного методу під тест. У випадку, якщо виконання тестового сценарію призводить до виникнення винятку, можна визначити відповідну частину. Це робиться для того, щоб уникнути помилкової оцінки в процесі пошуку.

Генератор тестових даних. Генератор тестових даних є ядром RTG і відповідає за створення тестових сценаріїв. Є дві основні задачі: знаходження відповідних вхідних значень та формування розумної послідовності викликів методів. Обидва ці завдання не можуть бути виконані одразу. Фактично, для пошуку можливих тестових даних використовується генетичний алгоритм (ГА). Таким

чином, під час процесу пошуку зберігається популяція тестових особин, яка еволюціонує в напрямку пошуку 'хороших' тестових даних.

Особина сама по собі не може бути виконана, але містить всю необхідну інформацію для створення повного тестового випадку. Цю інформацію можна розділити на три різні категорії, а саме конструктор, послідовність викликів методів та виклик методу під тест. Представлення особини можна побачити на рисунку 2.

Алгоритм пошуку починається з випадково ініціалізованої популяції особин. Кожна особина трансформується в виконуваний тестовий випадок та оцінюється. Оцінка базується на наступній функції придатності:

$$f_{fitness} = (\text{cov} \cdot p) + \frac{executed_{cs}}{total_{cs}} \cdot (1 - p) \quad (1)$$

де p – значення між 0 і 1, cov –показник покриття коду, досягнутого тестовим випадком, $executed_{cs}$ – кількість виконаних структур керування, $total_{cs}$ –загальна кількість існуючих структур керування. Отже, значення придатності є значенням між 0 і 1, де значення, близьке до 1, вказує на кращих особин, ніж значення, близьке до 0.

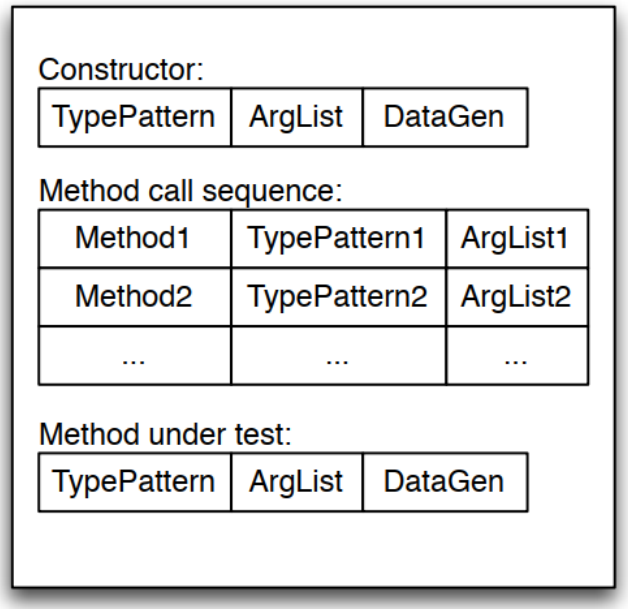

Рисунок 2 – Представлення особин

Особини вибираються з популяції, причому більш пристосовані особини мають більшу ймовірність вибору, ніж інші. Потім вони комбінуються і в кінцевому підсумку мутуються, щоб сформувати нове покоління популяції. Ці операції застосовуються по-різному на кожну частину особини.

Комбінація двох особин для конструктора та методу під тест стосується списку аргументів. У цьому випадку інформація про список аргументів обмінюється на випадково обраній позиції. З іншого боку, комбінація послідовності викликів методів відбувається шляхом вибору випадково двох позицій для кожної особи, на яких замінюється послідовність. Аналогічно випадок з мутацією, яка застосовується з попередньо визначеною ймовірністю до випадково обраних особин. Для конструктора та методу під тест існують дві можливості мутації, а саме генерація нового типового шаблону або виробництво нового вхідного значення для одного з існуючих аргументів. Генерація нового типового шаблону застосовується для охоплення комбінацій типів, які інакше не були б протестовані. Окрема таблиця ведеться для відстеження вже виконаних комбінацій типів. У разі, якщо всі можливі типові шаблони були протестовані принаймні один раз, тоді мутація стосується лише значення аргументу. Для мутації послідовності викликів методів випадково вибирається позиція, на якій метод додається або вилучається з поточної послідовності.

Оскільки Ruby підтримує поліморфізм, неможливо визначити з сигнатури методу, які типи параметрів можуть бути застосовані. Крім того, через комбінацію та мутацію особин може виникнути

нові комбінації типів. Така можлива ситуація показує наступний приклад. Припустимо, що у нас є метод, який приймає на вхід два параметри і застосовує оператор +. У цьому випадку можливі типові шаблони вхідних даних - (*Fixnum, Fixnum*) або (*String, String*), але будь-яка інша комбінація, така як (*Fixnum, String*) або (*String, Fixnum*), призводить до винятку.

Для зменшення кількості винятків система вивчає різницю між придатними та непридатними типовими шаблонами. Під час процесу пошуку операції можуть призводити до нових комбінацій типів. Така комбінація перевіряється на придатність, і в разі непридатності операція може повторюватися для знаходження кращого рішення. Система також вивчає різницю якості генераторів даних. Таким чином, якщо для конкретного типу потрібне нове вхідне значення, то вибір виконується на користь кращих генераторів. Оцінка генераторів даних ґрунтується на значенні придатності, що призначається тестовим випадкам. Генератори даних, які виробляли вхідні значення, які призвели до кращих кандидатів рішення, призведуть до більшої оцінки.

Висновок

У роботі наведено особливості реалізації RTG, інструменту для автоматичного створення тестових випадків для динамічної мови програмування Ruby. RTG може використовуватися для різних типів вхідних значень.

Встановлено різні види складності, які мають значний вплив на генерацію тестових випадків. Це самостійно визначені типи, складні та складені структури даних, вхідні дані з невеликим простором рішень та послідовності методів для зміни внутрішнього стану об'єкта. Складність послідовностей методів є чутливим фактором у автоматичному створенні тестових випадків. Чим складніше стає послідовність методів, тим складніше знаходити можливі рішення. Дуже складні та надійні послідовності методів можуть відбуватися нечасто, але в таких ситуаціях обидва генератори тестових випадків ймовірно не зможуть досягти високого покриття коду.

Метою RTG є знаходження тестових сценаріїв для охоплення якомога більшої кількості коду. Поточна версія RTG шукає придатні комбінації типів, а потім для кожного типу вибирає відповідний генератор даних. Цей проміжний крок не є обов'язковим. Більш ефективним рішенням було б безпосереднє використання набору наявних генераторів даних. У цьому випадку система шукала б комбінацію придатних генераторів замість типів даних, що може покращити якість, так і продуктивність створених тестових випадків.

Список використаних джерел

1. M. Alshraidehand L. Bottaci. Search-based software test data generation for string data using program-specific search operators: Research articles. Software Testing, Verification&Reliability,16(3):175–203, 2006.
2. A. Bouchachia. An immunegenetic algorithm for software test data generation. In HIS ’07: Proceedings of the 7th International Conference on Hybrid Intelligent Systems (HIS 2007), pp. 84–89, Washington, DC, USA, 2007. IEEE Computer Society.
3. M. Harmanand P. McMinn. A theoretical&empirical analysis of evolutionary testing and hill climbing for structural test data generation. InISSTA ’07: Proceedings of the 2007 International Symposium on Software Testing and Analysis, pp.73–83, NewYork, NY, USA, 2007,ACM.
4. S. Stinckwichand S. Ducasse. Introduction to the small talk special issue. Computer Languages, Systems&Structures, 32(2-3):85–86, 2006.
5. S. Wapplerand I. Schieferdecker. Improving evolutionary classtesting in the presence of non public methods. In ASE ’07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering, pp.381–384, NewYork, NY, USA, 2007. ACM.
6. S. Wapplerand J. Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In CEC’06: Proceedings of the 2006 IEEE Congress on Evolutionary Computation, pp. 851–858. IEEE, 2006.
7. S. Wapplerand J. Wegener. Evolutionary unit testing of object-oriented software using strongly typed genetic programming. In GECCO ’06: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, pp. 1925–1932,NewYork, NY, USA, 2006. ACM.
8. A. Watkinsand E. M. Hufnagel. Evolutionary test data generation: a comparison of fitness functions: Research articles. Software Practice and Experience, 36(1):95–116, 2006.
9. A. Windisch, S. Wappler, and J. Wegener. Applying particle swarm optimization to software testing. In GECCO ’07: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, pp.1121–1128, NewYork, NY, USA,2007. ACM.
10. R. Zhaoand Q. Li. Automatic test generation for dynamic data structures. In SERA ’07: Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management&Applications, pp. 545–549, Washington, DC,USA, 2007. IEEE Computer Society.