https://doi.org/10.15407/csc.2024.03.033
УДК 004.05+004.942+004.416.2+004.415.532.3

О.В. КОЛЧИН, кандидат фіз.-мат. наук, с.н.с.,
Інститут кібернетики ім. В.М. Глушкова НАН України,
просп. Академіка Глушкова, 40, м. Київ, Україна, 03187,
ORCID: https://0000-0001-7809-536X,
kolchin_av@yahoo.com

С.В. ПОТІЄНКО, кандидат фіз.-мат. наук, с.н.с.,
Інститут кібернетики ім. В.М. Глушкова НАН України,
просп. Академіка Глушкова, 40, м. Київ, Україна, 03187,
ORCID: https://0000-0001-9462-599X,
stepan.potiyenko@gmail.com

АВТОМАТИЗОВАНИЙ МЕТОД ПЕРЕВІРКИ ТА НАЛАГОДЖЕННЯ ТЕСТОВИХ СЦЕНАРІЇВ НА ОСНОВІ ФОРМАЛЬНОЇ МОДЕЛІ

Автоматичне створення тестів на основі моделі є популярною стратегією тестування. Однак багато звітів показують недоліки таких тестів, пов'язані з низькою якістю та сумнівною ефективністю. Щоб усунути ці проблеми та підвищити довіру до згенерованих тестових сценаріїв, ми пропонуємо інтерактивний метод їхнього аналізу та редагування, заснований на візуалізації шляху сценарію вздовж графа потоку керування моделі з додатковою інформацією про фактичну історію обчислень усіх змінних та можливі альтернативні варіанти поведінки. Метод сприяє безпечному внесенню змін до тестових сценаріїв, дозволяючи усувати виявлені недоліки та залишаючи цілі покриття неушкодженими. Для консистентної підстановки у параметри сигналів певних значень, які б визначали артефакти тестового середовища (такі як, наприклад, файли, бази даних тощо) та перевіряли граничні випадки (в предикатах умов, індексації масивів тощо), розроблено метод інтерактивної конкретизації символьних трас. Запропонований інструментарій суттєво зменшує час на дослідження результатів тестової генерації та редагування отриманих тестів, а його застосування позитивно впливає на якість та ефективність тестового набору.

Ключові слова: *модельне тестування, генерація тестів, налагодження, валідація тестів.*

Вступ

Формальні моделі активно використовуються для прототипування, а також можуть розглядатися як абстрактні реалізації вимог у розробці систем, критичних до безпеки. Розвиток формальних методів перевірки моделі дає змогу автоматизувати не лише перевірку властивостей, а ще й створення за методом білої скриньки тестових сценаріїв, які можуть бути покладені в основу для перевірки майбутньої реалізації.

Cite: Колчин О.В., Потієнко С.В. Автоматизований метод перевірки та налагодження тестових сценаріїв на основі формальної моделі. *Control Systems and Computers*, 2024, 3, 33-44. https://doi.org/10.15407/csc.2024.03.033

Для великих систем зі складною логікою поведінки автоматизація побудови тестів здатна суттєво зменшити трудомісткість і терміни виконання проєкту та підвищити тестове покриття. Але така автоматизація буде бажаною за умови достатньо високої якості отриманих тестів. Проблема полягає в суттєвих недоліках результатів: алгоритми генерації орієнтуються переважно на досягнення заданого покриття певного структурного критерію, а послідовність подій тесту зумовлена порядком перебору варіантів, виконаним алгоритмом обходу поведінки моделі, а не їх причинно-наслідковим взаємозв'язком. Тому виконання дій тесту часто не має смислового змісту; тести обриваються до логічного завершення (виконане обчислення не дало спостережуваного ефекту, який можна було би перевірити), і, як наслідок, виявляються неефективними під кутом зору спроможності знаходження помилок. Автоматичні тести потребують ручного супроводу для визначення умов успішного проходження, а також для подальшого оновлення у зв'язку зі змінами коду продукту. У багатьох роботах наголошується, що погано складені тести мають негативний ефект на їхній супровід.

Рішення щодо зарахування того чи іншого тесту до проєктного тестового набору має бути прийняте інженером. Отже, автоматизація валідації результатів тестової генерації є актуальною задачею. Часто виявлені проблеми тестів можна подолати завдяки незначним змінам, але при цьому виникає потреба у консистентних обчисленнях відповідно до оновлень. І такі обчислення можуть бути складними для ручної роботи, але їх легко автоматизувати. Така автоматизація і є метою цієї роботи.

У статті ми покажемо типові проблеми згенерованих тестів, з якими нам довелося зіткнутися на практиці [1, 2] та опишемо розроблені методи налагодження тестових сценаріїв, оцінки їхньої значущості, а також редагування. Також було розроблено прототип інтерактивного аналізу поведінки моделі, яка випробовується тестом. Його використання допомогло значно покращити тестовий набір та скоротити трудомісткість ручної роботи.

Проблеми створення набору тестів і мотиваційні приклади

Множина всіх поведінок програмної системи може бути надмірно великою, а за наявності циклів — навіть нескінченною. Тому на практиці до тестового набору висуваються вимоги щодо забезпечення певного покриття [3]. Критерії покриття допомагають визначити, чи охоплює набір тестів достатньо велику підмножину простору станів, щоб можна було довіряти програмній системі. Часто повнота тестового покриття оцінюється за кількістю виконаних операторів коду, а також гілок, шляхів, досягненню граничних значень функцій, виконання умов у виразах тощо [4]. Багато досліджень було присвячено порівнянню ефективності та стандартизації різних типів покриття. Наприклад, критерій Модифікованого покриття умов/рішень (*MC/DC*) включається до сертифікації *DO*-178*B* та *DO*-178*C* безпеки авіаційних систем, а також є частиною вимог до стандарту автомобільної безпеки *ISO* 26262 *Road Vehicles Functional Safety*.

Але питання про здатність таких тестів виявляти помилки залишається актуальним [1, 3, 5]. Традиційні підходи до створення набору тестів зосереджені переважно на досягненні вищого рівня покриття як основної мети. Ця стратегія часто призводить до того, що тести досягають своїх елементів покриття заплутаними шляхами. Багато емпіричних досліджень [3, 5—7] показують, що рівень структурного покриття сам по собі не є достовірним показником ефективності автоматично створених тестів і не повинен використовуватися як основний показник якості. Можна зустріти звіти (наприклад, [6, 7]), які свідчать про те, що забезпечення структурного покриття як мета для генерації тестів взагалі є помилковою стратегією: такі метрики розроблено для вимірювання неповноти покриття наявним тестовим набором та для локалізації непокритих ділянок, і не обов'язково ведуть до задовільних результатів, коли використовуються у зворотній ролі — як специфікації для генерації тестів.

```
 1.  in(x);
 2.  if( x == 1 )
 3.     res := compute();
 4.  else
 5.     res := -1;
 6.  try{ save_file(res);}
 7.  catch (exception ex){
 8.      print "failure";
 9.      return -1;
10.  }
11.  print "success";
12.  return 1;
```

Рис. 1. Приклад втраченого обчислення

```
 1.  in(x);
 2.  if( !access granted ) {
 3.    while(!access granted)
 4.      wait(5 min);
 5.     res := create();
 6.  } else
 7.    res := read();
 8.  if(x == 1)
 9.    stmt1();
10.  if(x == 2)
11.    stmt2();
 ...
```

Рис. 2. Приклад зайвої витрати ресурсів

Далі наведено типові випадки, з якими ми зіткнулися [1] під час використання методів генерації тестів.

Приклад 1. *Проблема*: передчасне завершення (рис. 1). Тестовий сценарій, який проходить шлях {1, 2, 3, 6, 7, 8, 9}, обчислює деяке значення (виклик функції *compute*, у рядку 3), але далі призводить до зупинення системи через не пов'язану аварійну ситуацію (у прагненні бути "коротким"). Тестовий набір уже може не містити іншого виклику функції *compute*, тому що вона "вже була покрита", і, як наслідок, значення *res* не стає частиною спостережуваного виходу. Таке дострокове завершення часто зумовлено стратегією пошуку в ширину (*breadth-first search*, *BFS*) та призводить до незадовільної спроможності виявлення несправностей. Наприклад, експерименти, проведені в [6], показують, що випадково згенеровані тести виявили більше мутацій, ніж тести з *MC/DC* і покриттям гілок, отриманих за допомогою *BFS*.

Приклад 2. *Проблема*: сценарій є занадто довгим, пролягає через непов'язану функціональність. Тести, створені за допомогою пошуку вглиб (*depth-first search, DFS*), зазвичай є заплутаними і містять випробовування рядків/функцій, які не пов'язані з основною метою тесту, та страждають від високого рівня перетинання тестових випадків і нерівномірного покриття. Це пов'язано з особливістю стратегії обходу *DFS* та призводить до двох основних проблем: критичне незбалансоване тестування поведінки [8] моделі та складне обслуговування тестів (зайве витрачання ресурсів; незначна зміна у вихідній моделі може вплинути на значну частину набору тестів).

У рис. 2 наведено код, де для покриття виклику функції *stmt*1() було згенеровано тестовий сценарій {1, 2, 3, 4, 5, 6, 8, 9, …}, який передбачав шлях через п'ятихвилинне очікування доступу (рядок 4), хоча такої витрати часових ресурсів можна було б уникнути у відповідний спосіб {1, 2, 6, 7, 8, 9, …}. Навіть більше, префікс {1, 2, 3, 4, 5} використовувався і для покриття подальших рядків коду, тобто виклику *stmt*2() і т.д., і лише один тест (саме для покриття рядка 7) уникав цього очікування.

Приклад 3. *Проблема*: відсутність спостережуваного ефекту (рис. 3, 4). Для покриття за слабким критерієм (покриття операторів) стратегія *DFS* побудувала лише один тест ($a = 0$; $b = 0$; $c = 0$), що відповідає шляху {1, 2, 3, 4, 5, 6, 7, 8, 9}, який випробовує всі рядки програми. За критерієм покриття *MC/DC* (для цього прикладу збігається з покриттям гілок) було отримано тести ($a = 0$; $b = 0$; $c = 0$), та ($a = 1$; $b = 1$; $c = 1$), другий відповідає шляху {1, 2, 3, 5, 7, 9}. Але ці тести не здатні виявити мутації (див. рис. 4) цієї програми, у якій видалено рядки 3–6, тобто, тестовий набір відбудеться так само успішно, як і для програми на рис. 3, тому що тестові сценарії не передбачають перевірки спостережуваного ефекту [9] від досягнутого покриття.

Подібні випадки трапляються доволі часто. Для запобігання цим недолікам розвивають методи генерації: інтегрують спеціальні евристики та застосовують більш вимогливі або створюють нові критерії покриття [1, 2, 9]. Хоча

```
1. in(a,b,c);
2. res := "no errors";
3. if( a == 0 )
4.     res := "error A";
5. if( b == 0 )
6.     res := "error B";
7. if( c == 0 )
8.     res := "error C";
9.  return res;
```

Рис. 3. Програма обробки помилок

```
1. in(a,b,c);
2. res := "no errors";
3. if( a == 0 )
4.     res := "error A";
5. if( b == 0 )
6.     res := "error B";
7. if( c == 0 )
8.     res := "error C";
9.  return res;
```

Рис. 4. Хибна програма обробки помилок

такі заходи і здатні вдосконалити результати (наприклад, покриття потоку даних сприяє подоланню проблем з відсутністю спостережуваного ефекту), тим не менш, автоматизацію бажано використовувати як "калькулятор" і "радника", а не для цілковитої заміни людини [10], особливо на етапі валідації та інтеграції тестів до проєктного тестового набору.

У цій статті ми не розглядатимемо способи пошуку шляхів покриття, а зосередимося на проблемі налагодження тестових сценаріїв, побудованих на основі моделі. Для отримання тестів, яким можна довіряти, потрібно докласти чимало роботи — доводиться "вручну" обстежувати поведінку згідно з вхідними даними та перевіряти, (1) що немає помилки або надмірної абстракції, (2) що тест має сенс, оскільки часто в гонитві за покриттям тести виявляються безглуздими, а також шукати можливість незначної зміни тесту для значного поліпшення його якісних характеристик [8, 9, 11, 12].

Об'єктивна оцінка тестового сценарію

Спочатку потрібно оцінити вихідний тест і зібрати про нього інформацію, яка є важливою для проєкту. Наприклад, для випробування чого він був побудований? Який елемент покриття був його головною метою? Перевірку яких вимог він виконує? Часто буває корисним створення певної матриці відповідності між вимогами до системи, її реалізацією та тестами, що перевіряють вимоги. Це дає змогу простежити покриття вимог та сприяє цілеспрямованості тестів. Також корисною є інформація щодо допоміжних якостей, таких як здатність виявляти помилки, інтенсивність використання ресурсів, зв'язність і узгодженість даних, рівномірність покриття [1, 8].

Оцінювання покриття. Спочатку для всього набору тестів створюємо таблицю відповідності щодо елементів покриття. Наприклад, для простого критерію покриття рядків коду кожному тесту ставитимемо у відповідність номери рядків, або, якщо цей тест побудовано для випробування виконання певної вимоги, яка була виражена за допомогою формули темпоральної логіки, то відповідну вимогу (або формулу). Це дасть змогу відстежувати рівномірність покриття та буде підказкою щодо можливих змін у тесті. Також відстежується додаткова інформація про покриття граничних випадків (у предикатах умов, індексації масивів тощо). Особлива увага приділятиметься елементам покриття, які покриваються тільки в одному тесті.

Оцінювання здатності виявляти дефекти. Тут використовується мутаційний підхід. Для кожного тесту буде поставлено у відповідність ті мутації моделі, які цей тест здатен виявити. При цьому не потрібно генерувати моделі з мутаціями, натомість, буде одноразово згенеровано глобальну множину всіх типових мутацій моделі, після чого для кожного окремого тесту можна визначити відповідну для нього підмножину за лінійний час. Наприклад, якщо в результаті присвоєння у тестовому сценарії, змінна набуває нового значення та досягає відповідного спостереження, то це означатиме, що виявлено відповідну мутацію "пропущене присвоєння". Аналогічно оброб-

ляються мутації в передумовах: якщо мутація робить певну умову в тестовому сценарії невиконуваною, то така мутація вважатиметься виявленою (на ній проходження тесту буде невдалим). Як наслідок, буде автоматично отримано кількісну та якісну оцінку щодо "здатності до виявлення дефектів" для кожного тесту.

Оцінка ресурсоємності. У реальних системах часто необхідно враховувати трудомісткість створення відповідного тестового оточення та ресурсів, необхідних для успішного проходження тесту. Наприклад, нераціонально тестувати роботу індикатора, вимагаючи при цьому багатохвилинного очікування даних для індикації або досягнення максимальної швидкості руху транспортного засобу. В результаті такого оцінювання мають бути визначені критично важливі ресурси та місця їх споживання.

Оцінка згуртованості даних. Серед найтиповіших недоліків автоматично згенерованих тестів вказується слабкий зв'язок даних усередині тестового сценарію [1, 2, 8, 9]. Певні значення обчислюється, але не використовується, що впливає на читання тесту, осмислення його поведінки, рівень надлишковості. У зв'язку з цим ми пропонуємо враховувати покриття потоку даних (як окремих пар, так і певних ланцюжків обчислень). При цьому більшій зв'язності відповідатиме більша кількість покритих *def-use* пар (або ланцюжків), а випадки, в яких є присвоєння, але немає його використання (тобто *def* без відповідного *use*) — навпаки, знижуватимуть цю оцінку. Оцінювання можна уточнювати, вибираючи особливі точки інтересу та розглядаючи зв'язність вибірково.

Оцінка перекриття тестів. Оцінка накладання насправді не може бути прямим свідченням вади тестового набору. Рівномірністю часто нехтують, наприклад, на користь зменшення споживання ресурсів, тобто певна ділянка часто зустрічається, тому що обходить ресурсозатратні шляхи нерелевантні щодо мети тесту. Проте рівномірність покриття є — бажаною властивістю [1, 8], яку можна оцінювати простим збором статистики покриття рядків по всьому набору.

Суб'єктивна перевірка тестового випадку та роз'яснення його обчислень

Валідація тестового сценарію. Суть цього етапу — візуальний контроль за шляхом тестового сценарію, тобто за послідовністю операторів (умов, правил, дій) моделі. Особливої уваги заслуговуватиме унікальне покриття, тобто ті елементи (вимоги або властивості) моделі, які серед усього тестового набору перевірятимуться тільки у цьому сценарії. Такий контроль є суб'єктивним і виконується "вручну" (визначається рівень зрозумілості, заплутаності, читабельності [13]), але перевірку на відповідність певним характеристикам можна автоматизувати, наприклад, чи стають ефекти покриття спостережуваними (досягають вихідних сигналів). З міркувань стислості у візуалізації сценарію корисним є режим "тільки спостережувані", в якому використовується гамачне згортання секцій, які не включають вхідних або вихідних сигналів. Аналіз впливу та залежності окремих операторів автоматизується методом побудови відповідних шарів (прямих або зворотних). Ознакою виконання зайвих дій в тестовому сценарії відносно покриття обраної точки інтересу може стати кількість операторів сценарію, які не належать ні зворотному шару, ні дереву предомінаторів. В результаті можна виставляти певну кількісну оцінку тесту та виокремити ділянки, які потребують редагування. Якщо оцінка є незадовільною або виокремлено багато ділянок для змін, то такий тест можна взагалі не додавати у проєктний тестовий набір, при цьому необхідно зберегти інформацію щодо цілей тесту та досягнутих у ньому елементів покриття задля подальшої переробки (або для повторного запуску генерації).

Роз'яснення історій обчислювань у тестовому сценарії. Для дослідження історії обчислення певної змінної (з прив'язкою до потоку керування) попередньо будується *SSA*-форма (*Static Single Assignment*) моделі, на основі якої класичними методами [14] аналізуються залежності даних та потоку керування. Появ-

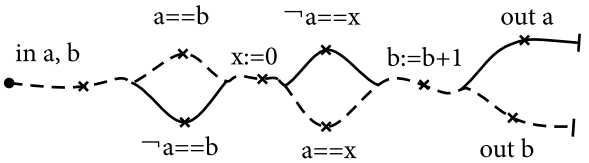

Рис. 5. Потік керування моделі з виділеним (пунктиром) шляхом тестового сценарію

нення також включає формулу причини невиконання (*unsat core*) [15] для кожної незадовільненої умови, суміжної зі шляхом обраного сценарію. Також треба зазначити, що семантика символьного виконання може уточнювати обмеження на змінну у передумові, а з огляду на класи еквівалентності (або нерівності) свого значення змінна може набути, навіть не входячи контекстно у передумову. Наприклад, після передумови ($a == b$), змінна b отримає конкретне значення 0 під час виконання подальшої умови ($a == 0$).

Таке дослідження схоже на розширений режим налагодження, в якому можна рухатися не лише вперед, а й аналізувати причиново-наслідкові зв'язки у зворотному напрямку, маючи на додачу підказки щодо місць фактичних та потенційно можливих змін значень змінних моделі. Розглянемо приклад моделі, зображеній на рис. 5. На ньому показано граф потоку керування моделі з виділеним шляхом, який відповідає обраному тестовому сценарію. Історія обчислень змінної b на ньому описує послідовність {*input* b; $a = b$; $b = 0$; $b := 1$; *output* b}.

Робота з циклами. Оскільки сценарій має певну обмежену довжину, то можна розрізняти всі виконані ітерації циклів як окремі ділянки. Отже, дозволяється "згортати" всі ітерації в одну (яка виглядатиме як відповідна частина графу потоку керування), або навіть "розгортати" певні окремі ітерації для їхнього детального дослідження [15], наприклад, можна відокремити останню ітерацію для детальнішого вивчення умови виходу із циклу.

Реконструкція тестового сценарію

Наведемо методи редагування тестового сценарію. Потрібність внесення змін може бути зумовлена недоліками, виявленими на попередніх етапах, оптимізацією або змінами у вихідній моделі на етапах підтримки та супроводження проєкту. При цілковито "ручному" редагуванні постають проблеми у синхронізації пов'язаних змін, які треба виявити та обчислити заново, та в оновленні інформації стосовно пов'язаних елементів покриття. Для подолання цих проблем розроблено інтерактивний конструктор трас [16], мета якого полягає саме в автоматичному обчисленні значень змінних моделі протягом всієї історії виконання сценарію та збору інформації щодо отриманого структурного покриття або покриття, зумовленого певною формулою темпоральної логіки. Розглянемо приклади його роботи.

Ситуація 1. Спочатку доцільно зафіксувати ділянки, які *не мають бути змінені* (які належать до унікального покриття чи з інших причин). До цих ділянок додаємо об'єднання їхніх зворотних шарів. Як наслідок, отримаємо ще більшу кількість "зафіксованих" операторів сценарію, які не можна буде змінювати. Отже, решту можна редагувати без впливу на важливе покриття. Передбачається, що ділянки, які потребують замін, уже ідентифіковано, й тоді потрібно оцінити здійсненність відповідних до них альтернатив, тобто, виконуваність інших умов на розгалуженнях. Виконуваність усіх умов, розташованих на розгалуженнях, які є суміжними до обраного сценарію, можна обчислити в процесі інтерактивної роботи, тому що їх множина є обмеженою довжиною обраного сценарію. Далі користувач редагуватиме сценарій, обираючи можливу альтернативу (умова якої позначена як "здійсненна"). При цьому шлях сценарію буде розділено на префіксну частину (до зміни) та суфіксну (саму зміну та після неї). Префіксна частина залишається без змін, а суфіксна буде автоматично оновлена згідно впливу зміненої умови. Звертаємо увагу на те, що ця зміна не може належати до "зафіксованих" частин.

Ситуація, в якій потрібно вносити зміни до ділянок, які належать до зворотних шарів, є складнішою та потребує більшого залучення користувача. У найскладнішому випадку ко-

ристувач супроводжуватиме вибір альтернатив на розгалуженнях на всьому шляху тестового сценарію.

Ситуація 2. У процесі еволюції проєкту зміни можуть потребувати відповідного оновлення тестового набору. У разі навіть невеликих змін постає питання про синхронізацію обчислень, на які ці зміни впливають у наявному тестовому наборі (який уже затверджено, прийнято та використовується). При простому підході процес генерації тестового набору треба повторити на оновленій моделі. Але при цьому є великий ризик отримати зовсім інші тестові сценарії, оскільки вони зумовлені порядком обходу графу поведінки моделі, а отже, зміна альтернативи на початку розгортання може суттєво вплинути на весь подальший пошук. Як наслідок, доведеться повторити весь етап валідації тестового набору та створення тестового оточення, а це є невиправдано трудомістким процесом. Тому актуальною є задача перевірки придатності попередніх тестів. Для цього ми перевірятимемо проходження попереднього тестового сценарію на новій моделі та вказуватимемо на черговий конфлікт, який зустрінеться, для його подальшого вирішення під керівництвом користувача.

Конкретизація тестового сценарію

Тестовий сценарій задається послідовністю переходів $t_1,\ldots, t_n$ і початковим станом E_0. Початковий стан описує обмеження на значення змінних програми у вигляді формули числення предикатів першого порядку. Формула може бути тотожно істинною, що означає відсутність будь-яких обмежень. Зазвичай послідовність переходів містить вхідні та вихідні сигнали, параметри яких являють інтерес для тестування.

Задача конкретизації [17] полягає в підборі набору конкретних значень початкового стану для всіх змінних, що зустрічаються в сценарії, а також підборі значень параметрів вхідних сигналів. Проблема полягає в тому, що, за побудовою з використанням екзистенційної абстракції, формула початкового стану не є визначальною для проходження тесту. Тому перед підбором конкретних значень треба уточнити цю формулу, тобто визначити всі необхідні обмеження (які стануть інструкціями для підготовки тестового середовища). Для цього ми пропонуємо наступний алгоритм.

Етап 1. Від початкового стану E_0 пройдемо тестовий сценарій за допомогою предикатного трансформера *pt* [18]. Для цього необхідно зробити n кроків, де крок i, $1 <= i <= n$, визначає стан системи E_i:

$$E_i = pt(E_{i-1} \wedge pre(t_i), post(t_i))$$

$pre(t_i)$ і $post(t_i)$ — перед- і пост-умови переходу t_i.

Етап 2. Від отриманого стану E_n пройдемо тестовий сценарій у зворотному напрямку за допомогою зворотного предикатного трансформера pt^{-1} [18]. Для цього також зробимо n кроків, де крок i, $1 <= i <= n$, визначає стан системи E'_{n-i}:

$$E'_{n-i} = pt^{-1}(E_{n-i+1,} pre(t_{n-i+1}), post(t_{n-i+1}))$$

Отримаємо нову формулу початкового стану E'_0. Згідно з властивостями зворотного предикатного трансформера [19], формула E'_0 є уточненням початкового стану E_0, (тобто виконуватиметься $E'_0 \rightarrow E_0$) та є необхідним і достатнім обмеженням на значення змінних для проходження тестового сценарію (тобто виконання послідовності переходів $t_1,\ldots,t_n$). Із цього випливає:

1) всі формули проміжних станів E'_i є необхідними та достатніми для виконання послідовності $t_{i+1},\ldots, t_n$;

2) будь-який набір конкретних значень, що задовольняє E'_0, призведе до успішного виконання тестового сценарію.

Далі опишемо методи підбору конкретних значень. Підбір здійснюється ітеративно. Обираємо значення однієї змінної, яке задовольняє формулі E'_0, підставляємо його, спрощуємо формулу (методи спрощення представляють широку область досліджень, що залишається за межами цієї публікації), обираємо значення наступної змінної, тощо. Послідовність змінних може бути довільною. Якщо тестовий сце-

нарій містить вхідні або вихідні сигнали, необхідно підібрати значення їхніх параметрів. Для цього використовуємо предикатний трансформер аналогічно до етапу 1 алгоритму уточнення формули. Беремо підібраний з E'_0 набір значень як початковий стан S_0, його можна розглядати як кон'юнкцію рівностей <змінна> = = <значення>. Далі зробимо n кроків, де крок i, $1 <= i <= n$, визначає конкретний стан системи S_i. Якщо перехід t_i не містить вхідних сигналів, то:

$$S_i = pt(\, S_{i-1} \wedge pre(t_i), post(t_i)\,).$$

Якщо при цьому в t_i є вихідні сигнали, то значення їхніх параметрів беруться з попереднього стану S_{i-1}.

Якщо ж перехід t_i містить вхідні сигнали, то спочатку треба побудувати проміжну формулу S'_i:

$$S'_i = pt(\, S_{i-1} \wedge pre(t_i), post(t_i)\,) \wedge E'_i.$$

Ця формула не є конкретним станом системи, оскільки предикатний трансформер видалить із неї значення змінних, які стоять у параметрах вхідних сигналів, вони лише набувають обмежень з E'_i. Конкретні значення підберемо так само, як у початковому стані, тобто ітеративно від формули S'_i прийдемо до конкретного стану S_i. В результаті виконання всіх n кроків отримаємо конкретизований тест з усіма необхідними значеннями, як у початковому стані, так і в параметрах вхідних і вихідних сигналів.

Постає питання, як краще підбирати значення і скільки таких наборів, отже, й конкретизованих тестів, треба побудувати для одного символьного тестового сценарію. Тут ми пропонуємо два підходи — автоматичний та інтерактивний за участю користувача. Досвід промислових проєктів показав, що частіше інтерес являють граничні значення. Для їх пошуку треба визначити змінні, які мають конкретні діапазони, тобто змінні, для яких у кон'юнкції формули стану зустрічаються порівняння з конкретними значеннями (формулу стану треба перетворити в ДНФ і кожен кон'юнкт розглядати окремо). Наприклад, формула $x > 0 \wedge y > x$ задає конкретний діапазон лише для змінної $x \in (0, +\infty)$. Тут ми розглядаємо три варіанти — ліва або права межа, або середнє значення. Лівою межею є 1, правою — максимальне значення для типу змінної x, середнє — будь-яке значення, зазвичай, це ліва межа + 1. Розглянемо вибір лівої межі, після підстановки отримаємо формулу $x = 1 \wedge y > 1$. Тепер з'явився конкретний діапазон для y, і за правилом лівої межі отримаємо конкретний стан $x = 1 \wedge y = 2$. В автоматичному режимі ми можемо побудувати 3^m конкретних станів для кожної формули, де m — це кількість змінних, присутніх у тестовому сценарії. Зазвичай, це призводить до надмірної кількості конкретизованих тестів, тому використовується одне правило для всіх змінних в одному тесті. Однак це може призвести до втрати важливих тестів.

З іншого боку, до цього процесу можна залучити користувача. По-перше, користувач може частково задати порядок підбору змінних, по-друге — задати індивідуальні правила підбору значень для обраних змінних. Для інших змінних існує інтерактивний режим, у якому конкретизатор ітеративно запитує користувача, у який спосіб конкретизувати змінну. Для цього виводяться поточні обмеження на змінну й очікується варіант конкретизації — одне з трьох правил або конкретне значення. Однак незрідка буває, що користувача цікавлять лише деякі параметри вхідних сигналів. Нехай точкою інтересу є вхідний сигнал у переході t_i. Тоді почнемо конкретизацію з цього переходу. Спочатку в інтерактивному режимі за допомогою користувача підберемо значення змінних-параметрів вхідних сигналів в t_i з урахуванням обмежень E'_i і отримаємо уточнений стан S'_i. Далі застосуємо алгоритм автоматичної конкретизації у такий спосіб: від стану S'_i пройдемо тестовий сценарій у зворотному напрямку, починаючи з переходу t_i, і отримаємо уточнені стани $S'_0, \ldots, S'_{i-1}$; підберемо значення для S'_0 й отримаємо конкретний початковий стан S_0; від S_0 пройдемо тестовий сценарій у прямому напрямку й отримаємо конкретизований тест. Якщо користувач задав кілька точок інтересу, то слід починати конкретизацію з

Псевдокод :

```
01:  print("Enter account id")
02:  input(account_id)
03:  acc[i] := read_from_db(account_id)
04:  if (acc[i].not_found)
05:      print("Account not found")
06:      return
07:  if (acc[i].balance > 10000)
08:      rate := 0.5
09:  else if (acc[i].balance > 5000)
10:      rate := 1
11:  else if (acc[i].balance > 0)
12:      rate := 2
13:  else
14:      print("Rejected")
15:      return
16:  print("Enter requested amount")
17:  input(sum)
18:  if (acc[i].score >= 10 and
         and sum > 0 and sum < 5000)
19:      print("Accepted, rate is ", rate,
         " credit score is ", acc[i].score)
20:  else
21:      print("Rejected")
```

Тестовий сценарій (індекс – номер рядку):

t_{01}, t_{02}, t_{03}, t_{04}, t_{07}, t_{09}, t_{10}, t_{16}, t_{17}, t_{18}, t_{19}

```
01:  out("Enter account id")
02:  in(account_id)
03:  in(acc[i]) and acc[i].id == account_id
04:  !(acc[i].not_found)
05:
06:
07:  !(acc[i].balance > 10000)
08:
09:  acc[i].balance > 5000
10:  rate := 1
11:
12:
13:
14:
15:
16:  out("Enter requested amount")
17:  in(sum)
18:  acc[i].score >= 10
     and sum > 0 and sum < 5000
19:  out("Accepted, rate is ", rate,
     " credit score is ", acc[i].score)
20:
21:
```

Рис. 6. Приклад коду та тестового сценарію

Прямий прохід:

E_0: 1
E_{01}: 1
E_{02}: 1
E_{03}: acc[i].id = account_id
E_{04}: E_{03} ∧ acc[i].not_found = false
E_{07}: E_{04} ∧ acc[i].balance <= 10000
E_{09}: E_{07} ∧ acc[i].balance > 5000
E_{10}: E_{09} ∧ rate = 1
E_{16}: E_{10}
E_{17}: E_{10}
E_{18}: E_{10} ∧ acc[i].score >= 10
∧ sum > 0 ∧ sum < 5000
E_{19}: E_{18}

Зворотний прохід:

E'_{19}, E'_{18}, E'_{17}:
acc[i].id = account_id ∧
acc[i].not_found = false ∧
acc[i].balance <= 10000 ∧
acc[i].balance > 5000 ∧
rate = 1 ∧
acc[i].score >= 10 ∧
sum > 0 ∧ sum < 5000

E'_{16}, E'_{10}:
acc[i].id = account_id ∧
acc[i].not_found = false ∧
acc[i].balance <= 10000 ∧
acc[i].balance > 5000 ∧
rate = 1 ∧
acc[i].score >= 10

E'_{09}, E'_{07}, E'_{04}, E'_{03}:
acc[i].id = account_id ∧
acc[i].not_found = false ∧
acc[i].balance <= 10000 ∧
acc[i].balance > 5000 ∧
acc[i].score >= 10

E'_{02}, E'_{01}, E'_{0}: 1

Рис. 7. Пряме та зворотне уточнення формул проміжних станів

останньої в послідовності переходів і при русі в зворотному напрямку зупинятись на інших точках інтересу та конкретизувати їх в інтерактивному режимі. Після цього прямий напрямок руху здійснюється автоматично, як у попередньому випадку. За допомогою наведених методів користувач може спрямовувати конкретизацію та будувати цікаві для нього тести.

На рис. 6—8 наведено: приклад псевдокоду та тестового сценарію (рис. 6), уточнення формул проміжних станів (рис. 7) та конкретизація одного тестового сценарію (рис. 8).

Автоматичний режим, правило лівої межи:

```
S0: 1
01: out("Enter account id")
02: in(account_id = 1)
03: in(acc[0]):
    acc[0].id = 1 ∧
    acc[0].not_found = false ∧
    acc[0].balance = 5001 ∧
    acc[0].score = 10
16: out("Enter requested amount")
17: in(sum = 1)
19: out("Accepted, rate is ", 1,
    " credit score is ", 10)
```

Інтерактивний режим, дві точки інтересу:
t_{03} і t_{17}

```
17: Введіть значення sum з обмеженнями
    sum > 0   sum < 5000:
>>> sum = 3000
03: Введіть значення acc[0] з обмеженнями
    acc[0].balance <= 10000 ∧
    acc[0].balance > 5000 ∧
    acc[0].score >= 10:
>>> acc[0].balance = 7500
    acc[0].score = middle
```

Конкретизований тест:

```
S0: 1
01: out("Enter account id")
02: in(account_id = 1)
03: in(acc[0]):
    acc[0].id = 1 ∧
    acc[0].not_found = false ∧
    acc[0].balance = 7500  ∧
    acc[0].score = 11
16: out("Enter requested amount")
17: in(sum = 3000)
19: out("Accepted, rate is ", 1,
    " credit score is ", 11)
```

Рис. 8. Приклади конкретизації

Для підготовки до тестування на цільовій платформі ми розробили процедуру трансляції конкретизованих тестів у виконувані. Вони є послідовностями інструкцій для ініціалізації середовища, імітації введення даних та перевірки параметрів вихідних сигналів. Так, у прикладі ми використали масив *acc*[ ] для симуляції роботи бази даних. З наведеного коду видно, що для успішного проходження тесту база даних має містити запис, що відповідає елементу *acc*[0]. Наша процедура генерує інструкції для відновлення такої бази даних перед виконанням тесту (наприклад, набір *SQL* запитів, таких як *CREATE TABLE* та *INSERT*).

Висновки

Робота присвячена автоматизації створення тестів на основі формальної моделі та підвищенню їх ефективності. Запропоновано метод інтерактивного налагодження тестових сценаріїв та їх валідації. Роль користувача при прийнятті рішення щодо додавання тесту у проєктний тестовий набір та внесення до нього змін залишається вирішальною, але задля зменшення трудомісткості автоматизовано такі процеси: оцінювання тестових сценаріїв за певними об'єктивними характеристиками (рівень покриття, здатність до виявлення дефектів, зв'язність даних та ін.); пошук можливих альтернатив для внесення корекцій; консистентне оновлення обчислень щодо відповідних корекцій. Окрема увага приділяється методу автоматизації підстановки конкретних значень для випробування граничних випадків для символьних шляхів сценаріїв.

На відміну від існуючих методів симуляції [20], запропонований метод інформує не тільки значення змінних, але й досліджує історію їх обчислень та додатково надає інформацію щодо можливих альтернатив.

На основі запропонованих методів створено дослідний прототип. Емпіричні результати продемонстрували позитивний вплив на загальну ефективність (спроможність виявляти дефекти та зменшення споживання ресурсів) та якість (змістовність, зручність читання, обслуговування, корисність налагодження тощо [12]) створених наборів тестів, при цьому досягнувши суттєвого скорочення трудовитрат на процес валідації. В окремих випадках вдалося запобігти втраті результатів генерації тестового набору (на виході тес-

тового генератора була проблема з незадовільним, або взагалі відсутнім спостереженням результатів обчислень у більшості тестових сценаріїв) завдяки зручності ідентифікації цілей покриття та ефективності редагування сценаріїв.

Як розвиток, планується вдосконалити процес локалізації причин невдалого проходження тесту для прискорення пошуку дефектів, а також розширити метод для використання у тестуванні розподілених систем з високим рівнем паралельних обчислень.

REFERENCES

1. Kolchin, A., Potiyenko, S., Weigert, T. (2019). "Challenges for automated, model-based test scenario generation". *In Comm. in Computer and Inf. Sci*., vol. 1078, pp. 182–194.
2. Kolchin, A., Potiyenko, S. (2022). "Extending Data Flow Coverage to Test Constraint Refinements". In: ter Beek, M.H., Monahan, R. (eds) Integrated Formal Methods. *Lecture Notes in Computer Science*, vol. 13274. https://doi.org/10.1007/978-3-031-07727-2_17.
3. Rushby, J. (2008). "Automated test generation and verified software". Verified Software: Theories, Tools, Experiments, pp. 161–172. https://doi.org/10.1007/978-3-540-69149-5_18.
4. Dssouli, R. et. al. (2017). "Testing the control-flow, data-flow, and time aspects of communication systems: a survey". *Advances in Computers*. vol. 107, pp. 95–155.
5. Inozemtseva, L., Holmes, R. (2014). "Coverage is not strongly correlated with test suite effectiveness". *In Proc. of ACM ICSE*'14, pp. 435–445.
6. Gay, G., Staats, M., Whalen, M., Heimdahl, M. (2015). "The risks of coverage-directed test case generation". *IEEE Transactions on Software Engineering*. vol.41, pp. 803–819.
7. Staats, M. et. al. (2012). "On the danger of coverage directed test case generation". *Lecture Notes in Computer Science,* vol. 7212, pp. 409–424. https://doi.org/10.1007/978-3-642-28872-2_28.
8. Palomba, F., et. al. (2016). "Automatic Test Case Generation: What if Test Code Quality Matters?" *In proc. of Int. Symp. on Software Testing and Analysis*, pp. 130–141.
9. Ying, M., Gay, G., Whalen, M. (2018). "Ensuring the observability of structural test obligations". *IEEE Transactions on Software Engineering*, 46 (7), pp. 748–772.
10. Perez, F., Font, J., Arcega, L., & Cetina, C. (2022). "Empowering the Human as the Fitness Function in Search-Based Model-Driven Engineering". *IEEE Transactions on Software Engineering*, 48 (11), pp. 4553–4568.
11. Ceccato, M., et.al. (2015). "Do automatically generated test cases make debugging easier? An experimental assessment of debugging effectiveness and efficiency". *ACM Transactions on Software Engineering and Methodology,* 25(1), pp. 1–38.
12. Lucredio, D., Vincenzi, A., Almeida, E., Ahmed, I. (2023). "Test case quality: an empirical study on belief and evidence". arXiv:2307.06410.
13. Winkler, D., Urbanke, P., Ramler, R. (2022). "What Do We Know About Readability of Test Code? - A Systematic Mapping Study". *In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering* (SANER), Honolulu, HI, USA, pp. 1167–1174.
14. Rastello, F., Tichadou, F. (2022). *SSA-based Compiler Design*. Springer Nature, 382 p.
15. Kolchin, A.V. (2010). "An automatic method for the dynamic construction of abstractions of states of a formal model". *Cybernetics and Systems Analysis,* 46 (4), pp. 583–601. https://doi.org/10.1007/s10559-010-9235-9.
16. Kolchin, A. (2018). "Interactive method for cumulative analysis of software formal models behavior". *Proc. of the* 11*th Int. Conf. on Programming UkrPROG*'2018, CEUR-WS vol. 2139, pp. 115–123.
17. Kolchin, A. et. al. (2013). "An approach to creating concretized test scenarios within test automation technology for industrial software projects". *Automatic control and computer science*, pp. 433–442. https://doi.org/10.3103/S0146411613070213.
18. Godlevskiy, A.B., Potiyenko, S.V. (2010). "Obratnaya transformatsiya formul v simvol'nom modelirovanii: ot rezul'tata k iskhodnoy formule". *Problemi programuvannya,* 2–3, pp. 363–368 [Годлевский А.Б., Потиенко С.В. Обратная трансформация формул в символьном моделировании: от результата к исходной формуле. Проблеми програмування. 2010. № 2–3. С. 363–368].
19. Potiyenko, S.V. (2008). "Metody pryamogo i obratnogo simvol'nogo modelirovaniya sistem, zadannykh bazovymi protokolami". Problemi programuvannya, 4, pp. 39–45 [Потиенко С.В. Методы прямого и обратного

символьного моделирования систем, заданных базовыми протоколами. Проблеми програмування. 2008. № 4. С. 39–45].
20. Vu, F., Leuschel, M. (2023). "Validation of Formal Models by Interactive Simulation". In: Glässer, U., Creissac Campos, J., Méry, D., Palanque, P. (eds) Rigorous State-Based Methods. ABZ 2023. *Lecture Notes in Computer Science*, vol 14010. Springer, Cham. https://doi.org/10.1007/978-3-031-33163-3_5.

Надійшла 14.04.2024

O.V. Kolchyn, PhD (Comp.Sci), Senior Researcher,
V.M. Glushkov Institute of cybernetics NAS of Ukraine,
Academican Glushkov ave, 40, Kyiv, Ukraine, 03187,
ORCID: https://orcid.org/0000-0001-7809-536X,
kolchin_av@yahoo.com

S.V. Potiyenko, PhD (Comp.Sci), Senior Researcher,
V.M. Glushkov Institute of cybernetics NAS of Ukraine,
Academican Glushkov ave, 40, Kyiv, Ukraine, 03187,
ORCID: https://orcid.org/0000-0001-9462-599X,
stepan.potiyenko@gmail.com

AN AUTOMATED METHOD FOR CHEKING
AND DEBUGGING TEST SCENARIOS BASED ON FORMAL MODELS

Introduction. Model-based test cases generation is a popular strategy for test automation. It helps to reduce time spent on the development of a test suite and can improve the level of coverage. However, many reports show a shortage of such test cases in poor quality and doubtable efficiency.

Purpose. The main goal of the proposed method is cost-effective validation, assessment, debugging and debugging of generated test cases. The method helps improve the quality and efficiency of the test cases and makes their scenario meaningful and goal-oriented. The method also develops debugging facilities and simplifies data dependency analysis and test scenario editing.

Methods. We propose an automated post-processing method which allows to evaluate path that is examined by the test case, and to make safe changes to the path which will eliminate the shortcomings while leaving the coverage targets of the test case unharmed. The method is based on visualization of the path along the control flow graph of the model with additional information about the factual evaluation history of all variables and possible alternative variants of behavior. For consistent substitution of certain values in the signal parameters, which would determine the artifacts of the test environment (such as files, databases, etc.) and check boundary cases (in predicates of conditions, indexing of arrays, etc.), a method of interactive specification of symbolic traces have been developed.

Results. The role of the user in deciding whether to add a test case to the project test suite and make changes to it remains crucial, but to reduce labor intensity, the following processes are automated: evaluation of test scenarios according to certain objective characteristics (level of coverage, ability to detect defects, data cohesion, etc.); highlighting of possible alternatives for making corrections; consistent updating of computations for the corresponding corrections. A prototype was developed based on the proposed methods. The empirical results demonstrated a positive impact on the overall efficiency (ability to detect defects and reduce resource consumption) and quality (meaningfulness, readability, maintenance, usefulness for debugging, etc) of the generated test suites. The method allows to make automatically generated test cases trustable and usable.

Conclusion. The proposed toolkit significantly reduces the time spent on researching the results of test generation and validation of the obtained tests and their editing. Unlike existing simulation methods, the proposed method not only informs about the values of variables but also explores the history of their computations and additionally provides information about admissible alternatives. Further, we plan to improve the process of localizing the causes of test failure at the execution phase to speed up the search for defects.

Keywords: *model-based testing, test case generation, debugging, test case validation.*