DOI 10.31891/2307-5732-2023-321-3-273-279
УДК 519.681.2

КОСТИРКО ВАСИЛЬ
Львівський торговельно-економічний університет
ORCID ID: 0000-0002-6366-8695
e-mail: vkostyrko@lute.lviv.ua
АНІЛОВСЬКА ГАННА
Львівський торговельно-економічний університет
ORCID ID: 0000-0002-0154-1584
e-mail: hjaa5514@gmail.com
ПЛЕША ВАСИЛЬ
Львівський торговельно-економічний університет
ORCID ID: 0000-0001-5321-9602
e-mail: plesha_v@i.ua

ПРОЄКТУВАННЯ БІБЛІОТЕКИ ДЛЯ СПРОЩЕННЯ УМОВ ВЕРИФІКАЦІЇ ПРОГРАМ

Анотація програми інваріантами, заданими в певних точках програми, зводить проблему її верифікації до перевірки істинності ряду логічних виразів. Однак, проблема виводимості є алгоритмічно нерозв'язною навіть для достатньо простих предметних областей, таких як елементарна арифметика. Тому важлива розробка комп'ютерних засобів, які можуть спростити задачу верифікації програм. Особливо важлива наявність таких систем для навчання програмуванню. До них відносяться технологія символьного виконання програм та інструменти для еквівалентних перетворень та спрощення символьних виразів. Залишаються актуальними дослідження формальних методів верифікації програм. У статті описана побудована на базі бібліотеки ExprLib технологія автоматичного еквівалентного перетворення арифметичних та логічних виразів до найпростішого стандартизованого вигляду. Досліджено побудовані перетворення. Встановлено, що вони мають властивість ідемпотентності, а еквівалентні вирази мають єдине нормалізоване представлення. Бібліотека написана алгоритмічною мовою Python з використанням рекурсії на структурах даних типу дерево. Бібліотека реалізована класами та функціями мови Python, які представляють деревовидні структури списками, множинами та словниками мови Python. Побудовано класи для представлення дерев, одночленів та поліномів, відношень та їх кон'юнкцій, а також імплікацій. Для спрощення імплікацій застосовується правило редукції кон`юнкцій. Бібліотека застосовується в системі VerPro символьного виконання та верифікації програм на мові Python. Для перевірки тотожної істинності імплікацій, які представляють умови верифікації, використовуються солвери системи доведення теорем Z3. Система VerPro є експериментальною і зараз обмежується програмами цілочисельної арифметики. Наявність власної бібліотеки дозволяє розширювати область застосування системи, яка розвивається в напрямку генерації інваріантів та розширення областей охоплених нею програм.

Ключові слова: бібліотека функцій і класів; мова Python; еквівалентні перетворення виразів; умова коректності; дерево виразу; верифікація програм.

KOSTYRKO VASYL, ANILOVSKA HANNA, PLESHA VASYL
Lviv University of Trade and Economics

DESIGNING A LIBRARY TO SIMPLIFY PROGRAM VERIFICATION CONDITIONS

The ExpLib library described for the equivalent transformation of arithmetic and logical expressions in order to reduce them to the simplest form based on the classical relations of integer arithmetic and mathematical logic. Annotating the program with invariants specified at certain points of the program reduces the problem of program verification to checking the truth of a set of logical expressions. However, the problem of derivability is algorithmically unsolvable even for sufficiently simple subject area, such as elementary arithmetic. Therefore, it is important to develop computer tools that can simplify the task of program verification. It is especially important to have such systems for programming teaching. These include the technology of symbolic execution of programs and tools for equivalent transformation and simplification of symbolic expressions. Formal methods of program verification are reflected in many modern studies. The article describes the technology of automatic equivalent conversion the arithmetic and logical expressions to the simplest standardized form built on the basis of the ExprLib library. These transformations have the property of idempotency, and the equivalent expressions have a single normalized representation. The library is written in the Python algorithmic language using recursion on tree-type data structures. The library is implemented by classes and functions built in Python using lists, sets, and dictionaries. These classes represent trees, monomials and polynomials, relations and their conjunctions, as well as implications. To simplify the implications, the reduction rule of conjunctions is used. The ExpLib is used in the VerPro system for symbolic execution and Python programs verification. To verify the identical truth of the implications, which represent the verification conditions, solvers of the Z3 theorem proving system are used. The VerPro system is experimental and currently limited to integer arithmetic applications. Having the own library allows us to expand the scope of the system. The VerPro system is developing in the direction of invariant generating and expanding the program area.

Keywords: library of functions and classes; Python language; equivalent transformations of expressions; correctness condition; expression tree; program verification.

Постановка проблеми

Стаття присвячена проблематиці формальної верифікації програм, анотованих передумовою, післяумовою та інваріантами, асоційованими з деякими точками програми – так, щоб у кожному циклі був принаймні один інваріант. Анотація програми інваріантами, заданими в певних точках програми, як встановив Флойд [2], зводить проблему її верифікації до перевірки істинності ряду логічних виразів. Однак, проблема виводу логічних формул є алгоритмічно нерозв'язною навіть для достатньо простих предметних

областей, в тому числі, для елементарної арифметики. Тому проблема верифікації програм залишається актуальною і все ще притягує дослідників. Об'єктом дослідження виступають анотовані програми, а предмет дослідження – бібліотека програм та програмний застосунок, які реалізують верифікацію.

Враховуючи це, в роботі було визначено такі основні проблеми дослідження: побудувати внутрішнє представлення арифметичних виразів – одночленів та поліномів, яке б гарантувало єдине і найпростіше їх зовнішнє представлення; побудувати внутрішнє представлення логічних виразів – відношень та їх кон'юнкцій з тими ж властивостями; побудувати аналогічне внутрішнє представлення умов верифікації у вигляді імплікації кон'юнкцій; реалізувати програмні функції для спрощення виразів з застосуванням співвідношень асоціативності, комутативності та дистрибутивності арифметичних та логічних операцій, а також операцій з арифметичними та логічними константами.

Аналіз останніх джерел

Еквівалентні перетворення арифметичних та логічних виразів реалізуються цілим рядом програмних пакетів: Mathematica [7], Maple, MATLAB, REDUCE. Мова Python також має бібліотеку символьних перетворень SymPy [6].

Однак, ці системи орієнтовані на застосування математиками в ролі специфічних формульних калькуляторів. Тому для реалізації системи VerPro нам прийшлося створити власний програмний інтерфейс (API), який базується на бібліотеці ExprLib.

Бібліотека ExprLib написана мовою Python і складається з функцій та класів, об'єднаних в такі три пакети:

treelib – пакет для представлення арифметичних та логічних виразів деревовидними структурами;

arilib – пакет для еквівалентного перетворення арифметичних виразів;

logilib – пакет для еквівалентного перетворення логічних виразів.

Ціллю перетворення арифметичних та логічних виразів в системі VerPro є приведення їх до більш простого (і бажано уніфікованого) вигляду за допомогою співвідношень асоціативності, комутативності, дистрибутивності, а також класичних співвідношень з області арифметики та математичної логіки.

При всій очевидності та тривіальності деяких фактів, співвідношень і властивостей арифметичних виразів під час реалізації бібліотеки їх прийшлось ретельно проаналізувати та формалізувати [7, 9]. Такі передумови появи бібліотеки ExprLib.

Багатократно більш складна та універсальна система верифікації програм Isabelle [13] вимагає значної роботи з освоєння системи та підготовки програм та їх анотацій для використання системи. Вона технологічно складна і тому використовується в унікальних науково–дослідних проєктах.

Мета даної роботи – описати принципи побудови бібліотек для автоматичного спрощення виразів з метою полегшення перевірки умов верифікації програм.

Виклад основного матеріалу

Перетворення арифметичних виразів. В арифметичних виразах будемо дозволяти використання простих змінних, знаків бінарних операцій + (додавання), – (віднімання) та * (множення), унарних операцій + (плюс) та – (мінус), дужок та цілих чисел.

Загальноприйняте текстове представлення арифметичного виразу є інфіксним і для визначення порядку виконання операцій містить дужки. Правило старшинства операцій дозволяє зменшити кількість дужок, що сприяє спрощенню виразів.

Однак, правила розстановки дужок в інфіксному представленні виразів є неоднозначними. Наприклад, $(((x))) = ((x)) = (x) = x$.

Асоціативність операцій додає такої неоднозначності, наприклад,

$$(x * y) * z = x * (y * z) = x * y * z .$$

Всі вирази, які з точністю до асоціативності еквівалентні заданому виразу, назвемо асоціативно–еквівалентними. Серед асоціативно–еквівалентних можна виділити два представлення заданого виразу з мінімальною кількістю дужок, які однозначно визначають порядок виконання операцій: з використанням старшинства операцій і без нього.

Наприклад, наступні вирази є асоціативно–еквівалентними і представляють два описані представлення:

$$((a * b) * c) + (c * a) = a * b * c + c * a .$$

Оскільки друге представлення містить меншу кількість символів, то воно людиною сприймається як простіше.

Інфіксне представлення виразу назвемо нормалізованим, якщо в ньому використовується мінімально можлива кількість дужок з огляду на старшинство та асоціативність операцій.

Стверджується, що кожен арифметичний вираз має єдине нормалізоване представлення, тому можна побудувати функцію нормалізації виразів N. Неважко переконатися, що функція N є ідемпотентною, тобто, для будь–якого виразу e в інфіксному представленні виконується співвідношення

$$N(N(e)) = N(e) .$$

Бібліотека ExprLib реалізує функцію T перетворення арифметичних виразів з інфіксного (дужкового) представлення в постфіксне (бездужкове), а також функцію зворотного перетворення T*, які будемо називати далі прямим та оберненим перетвореннями.

Перетворення T* є оберненим до T в тому смислі, що для будь–якого інфіксного виразу e

послідовне застосування до нього прямого та оберненого перетворень нормалізує цей вираз:

$$T^*(T(e)) = N(e)\ . \qquad (1)$$

Зрозуміло, що на множині нормалізованих виразів перетворення T* буде оберненим до T в загальноприйнятому смислі: для будь–якого нормалізованого виразу E:

$$T^*(T(E)) = E\ . \qquad (2)$$

Зауважимо, що пряме та обернене перетворення виразів використовують лише відношення асоціативності, але не застосовують відношень комутативності, дистрибутивності чи якихось інших. Тому ці перетворення можуть лише усувати зайві дужки та змінювати порядок виконання деяких операцій.

Відношення дистрибутивності дозволяє арифметичний вираз перетворити до вигляду полінома від багатьох змінних. Такий поліном будемо представляти у вигляді сум (різниць) одночленів, кожен з яких – це добуток довільної кількості цілих чисел та змінних.

Властивість комутативності й асоціативності операцій множення та додавання дозволяє побудувати однозначні представлення одночленів та поліномів, які назвемо впорядкованими.

Одночлен назвемо впорядкованим, якщо він має такі властивості:

Якщо одночлен має числовий множник, тоді він є першим і єдиним числовим множником; всі числові множники одночлена пересуваються на початок одночлена і перемножуються;

Якщо одночлен має числовий множник, який рівний 1, і нечислові множник, тоді він опускається згідно співвідношення $1 * x = x$;

Унарні операції застосовуються лише до коефіцієнта, змінюючи його знак відповідно до таких відношень:

$$-n = (-n), \quad +n = n,$$

де n – довільне ціле число, а (-n) – число n з протилежним знаком;

Нечислові множники одночлена впорядковані за алфавітом.

Стверджується, що впорядковане представлення одночлена – єдине. Символом M позначимо перетворення, яке для будь–якого одночлена видає його впорядковане представлення.

Якщо одночлен не вироджується в число і має числовий множник, тоді останній назвемо коефіцієнтом одночлена, а добуток інших множників – його основою.

Одночлени з однаковими основами назвемо подібними. Зведення подібних одночленів задається співвідношеннями

$$n * x + m * x = (n + m) * x, \quad n + m = (n + m),$$

де n, m– довільні цілі числа, (n + m)– їх сума, а x – довільна основа одночлена.

Поліном назвемо впорядкованим якщо:

Всі його одночлени є впорядкованими;

Серед одночленів немає подібних; до подібних одночленів застосовується зведення;

Серед коефіцієнтів одночленів немає нулів; тобто, до одночлена з нульовим коефіцієнтом та довільною основою x застосовується співвідношення $0 * x = 0$;

Число може бути лише першим одночленом полінома; всі числові одночлени підсумовуються;

Це число не може бути нулем (хіба що весь поліном вироджується в 0); тобто, застосовується співвідношення $0 + x = x$;

Одночлени впорядковані за їх основами в алфавітному порядку;

Всі коефіцієнти одночленів можуть бути лише додатними; від'ємний коефіцієнт першого одночлена полінома перетворюється в унарну операцію – (мінус), яка застосовується до всього одночлена; від'ємний коефіцієнт наступних одночленів перетворюються на бінарну операцію віднімання, яка застосовується до попереднього одночлена полінома та поточного.

Стверджується, що описане впорядковане представлення полінома – єдине. Символом P позначимо перетворення впорядкування полінома. Очевидно, перетворення M та P також є ідемпотентними.

Бібліотека ExprLib надає реалізацію перетворень T, T*, M та P. Поліном, побудований перетвореннями T, M та P, назвемо стандартизованим.

Представлення виразів деревами. Для обробки арифметичних виразів бібліотека ExprLib представляє їх деревами. Дерево арифметичного виразу визначимо за допомогою рекурсивного класу Tree з атрибутом op і двома орієнтованими асоціаціями – лівою lt та правою rt (рис. 1).

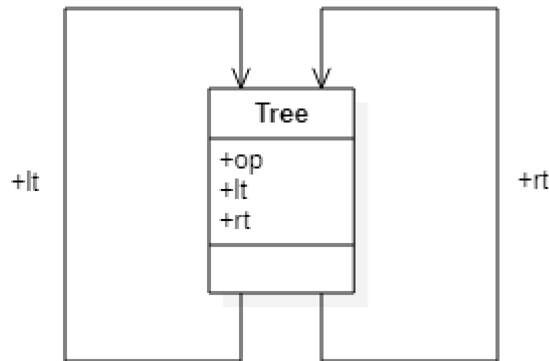

Рис. 1. Діаграма класу Tree / Class diagram of a tree

Об’єкти класу *Tree* задають вузли дерева виразу – його корінь, внутрішні вузли та листки, а атрибут *op* – текстове представлення операцій, змінних та чисел виразу. Асоціації *lt* та *rt* задають спадкоємців вузлів дерева. Якщо вузол дерева є листком, тоді його обидві асоціації пусті: *lt = rt = None*.

Якщо вузол дерева представляє бінарну операцію, тоді його обидві асоціації непусті і задають піддерева. Якщо вузол дерева представляє унарну операцію, тоді одна з асоціацій є пустою, а інша – ні й задає піддерево. Для визначеності будемо вважати, що ознакою унарної операції є пустота її лівої асоціації: *lt = None*.

Деревом арифметичного виразу будемо називати множину вузлів класу *Tree*, зв’язаних орієнтованими асоціаціями в ієрархічну структуру з одним коренем [5].

Рис. 2 демонструє представлення арифметичного виразу *-2 * (b + c)* за допомогою об’єктів класу *Tree*.

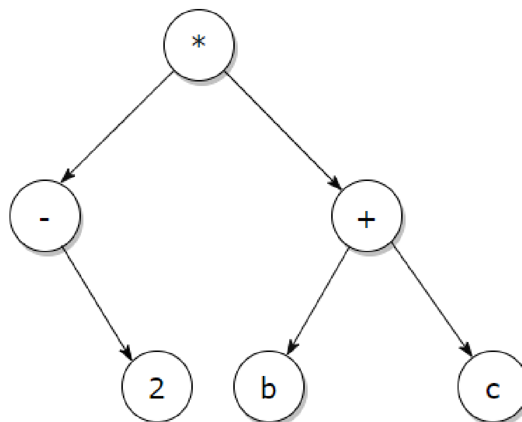

Рис. 2. Представлення виразу деревом / Tree representation of an epression

Частина методів класу Tree є простими, наприклад, наведені в приклад 1 предикати для визначення типу поточної вершини дерева.

Приклад 1. Предикати для визначення типу поточної вершини дерева.

```
def unary(self):
    return self.lt is None and self.rt is not None
def binary(self):
    return self.lt is not Noneand self.rt is not None
def leaf(self):
    res = self.lt is None and self.rt is None
    return res
```

Інші методи класу *Tree* є достатньо складними. Для того, щоб не ускладнювати класу, вони реалізуються функціями, які розташовуються в тому ж модулі *tree*.

Стандартний конструктор класу *Tree* утворює пусте дерево, тому для побудови кожного дерева потрібно було б створювати спеціальну програму. Для полегшення побудови дерев виразів в модулі *tree* реалізовано функцію *exprtree*, яка утворює дерево з заданого виразу. Ця функція фактично служить ще одним конструктором класу *Tree*.

Функція *exprtree* побудована з використанням функції *polish*, яка перетворює арифметичний вираз зі звичайної інфіксної форми в постфіксну з застосуванням стеку [5]. Функція *exprtree* фактично реалізує перетворення *T*, але не зупиняється на постфіксному представлення виразів, а одразу переходить до деревного.

Метод *tostring* перетворює дерево арифметичного виразу в інфіксну форму, також оминаючи постфіксне представлення виразу, фактично реалізуючи перетворення T^*. Очевидно, така підміна не впливає на співвідношення (1) та (2).

Модуль *tree* надає також функцію *equaltree*, яка визначає, чи два дерева задають один арифметичний вираз, функцію *copytree*, яка утворює копію заданого дерева виразу, та функцію *istarexpr*, яка перевіряє, чи її аргумент є деревом арифметичного виразу.

Аналіз одночленів. Для представлення впорядкованих одночленів бібліотека *ExprLib* надає клас *Monom*. У ньому одночлен представлений числовим коефіцієнтом та основою. Модуль *monom*, у якому розташований опис класу *Monom*, визначає ще декілька цікавих функцій.

Функція *similar* визначає подібність двох заданих одночленів. Функція *treetomono* перетворює одночлен, заданий об'єктом класу *Tree*, на об'єкт класу *Monom*. При цьому числові коефіцієнти перемножуються, а нечислові – впорядковуються за алфавітом. Функція *monototree*, навпаки, об'єкт класу *Monom* перетворює на об'єкт класу *Tree*.

Аналіз поліномів. Роботу з поліномами реалізує модуль *polynom*. Для цього в ньому описано клас *Polynom* впорядкованих поліномів. Функція *treetopoly* заданий деревом поліном перетворює на об'єкт класу *Polynom*, а функція *polytotree* здійснює обернене перетворення. Функція *distrall* циклічно застосовує до дерева виразу відношення дистрибутивності, копіюючи піддерева за допомогою функції *copytree*.

Коли доданки стають одночленами, вони піддаються впорядкуванню функціями та методами з модуля *monom*. Функція *combinesim* зводить подібні члени полінома, розпізнаючи їх за допомогою функції *similar*. Одночлени впорядковуються за алфавітом., ігноруючи їх коефіцієнти.

Перетворення арифметичного виразу в поліном фактично реалізує описані вище еквівалентні перетворення T, M, P та T^* арифметичного виразу до найпростішого вигляду в інфіксній нотації, який назвемо стандартизованим. Наприклад, арифметичний вираз $(-1) * (((q + 1) * y) + (r - y))$ приводиться до такого стандартизованого вигляду: $-q * y - r$.

Відношення арифметичних виразів. Найпростішими елементами логічних виразів служать відношення арифметичних виразів. Бібліотека *ExprLib* дозволяє такі операції відношення: < (менше), <= (менше або дорівнює), > (більше), >= (більше або дорівнює), == (дорівнює), != (не дорівнює).

Еквівалентними перетвореннями відношення приводяться до такої нормальної форми:

- Всі відношення виду *x* < у перетворюються на *y* > *x*, а відношення виду *x* <= у – на *y* >= *x*;
- Переносом арифметичних виразів з правої частини в ліву зі зміною їх знаку всі відношення приводяться до одного з таких чотирьох видів:

  z > 0, z >= 0, z == 0, z != 0;
- Всі відношення виду (*0 == 0*) та (*0 >= 0*) замінюються логічною константою *True*, а відношення виду (0 != 0) та (0 > 0) замінюються логічною константою *False*;
- Всі відношення виду *(n) == 0*, де *(n)* – додатне або від'ємне число, замінються логічною константою *False*, а відношення виду *(n) != 0* – логічною константою *True*;
- Всі відношення виду *(n) > 0* та *(n) >= 0*, де *(n)* – додатне число, замінються логічною константою *True*, а відношення виду *(n) > 0* та *(n) >= 0*, де *(n)* – від'ємне число, заміняються логічною константою *False*.

Описана вище функція exprtree поширена на відношення арифметичних виразів, перетворюючи їх в постфіксну форму, а далі – в об'єкти класу *Tree*. Функція *treeexpr* здійснює обернене перетворення. Описані вище перетворення забезпечують однозначне представлення і відношень, зберігаючи властивість ідемпотентності. Для представлення відношень стандартизованих поліномів бібліотека *ExprLib* надає клас *Relation*.

Логічні вирази. З відношень за допомогою логічних операцій мови Python можна будувати більш складні вирази. В логічних виразах виразах можна застосовувати лише дві логічних операції – кон'юнкції *and* та заперечення *not*.

Логічні вирази представляються тими ж об'єктами класу *Tree*, що й арифметичні вирази та їх відношення. Функції *exprtree* та *treeexpr* поширено і на логічні вирази.

А наступні співвідношення дозволяють позбутися застосування операцій заперечення до відношень арифметичних виразів:

(*not (x < y) = (x >= y)*), (*not (x <= y) = (x > y)*), (*not (x > y) = (x <= y)*),
(*not (x >= y) = (x < y*), (*not (x == y) = (x != y*), (*not (x != y) = (x == y)*).

Співвідношення (*a and a = a*) усувають з логічних виразів повторення, а співвідношення (*False and x = False*), (*True and x = x*) дозволяють позбутися логічних констант – крім випадку, коли весь логічний вираз представляє собою логічну константу.

Для представлення кон'юнкції стандартизованих відношень арифметичних виразів бібліотека *ExprLib* надає клас *Conjunct*.

Співвідношення комутативності й асоціативності операцій кон'юнкції та диз'юнкції дозволяють впорядкувати всі відношення кожної кон'юнкції аналогічно впорядкуванню стандартизованих поліномів.

Стверджується, що описане цими співвідношеннями перетворення всі еквівалентні вирази приводить до єдиної форми, яку назвемо стандартизованою. Таке перетворення, очевидно, теж є ідемпотентним.

Умови коректності. Відповідно до теорії Флойда–Хоора [2] задача перевірки коректності програми зводиться до деякого набору логічних виразів, які називаються умовами верифікації. Кожна умова верифікації представляє собою імплікацію, антецедент і консеквент якої, як правило, є кон'юнкціями відношень.

Система *VerPro* передбачає, що антецеденти і консеквенти умов верифікації є кон'юнкціями відношень арифметичних виразів. Для представлення таких умов бібліотека *ExprLib* надає клас *Implication*.

Верифікація анотованої програми здійснюється шляхом побудови умов верифікації та приведенням їх антецедентів та консеквентів до стандартизованого вигляду. Далі з консеквенту кожної умови коректності викреслюються ті кон'юнкти, які присутні в її антецеденті, відповідно до наступних еквівалентних співвідношень:

$$a \rightarrow a \text{ and } b = a \rightarrow b \text{ and } a = a \rightarrow b \quad (3)$$

$$a \text{ and } b \rightarrow a = True \quad (4)$$

Співвідношення (3–4) назвемо правилами редукції [1, 4]. Правила редукції теж задають еквівалентне перетворення і дозволяють спрощувати імплікації аж до перетворення їх в константи. Якщо всі умови верифікації програми істинні, тоді стверджується, що програма коректна.

Інакше ті умови верифікації, з якими не вдалося справитися еквівалентним перетворенням, передаються системі *Z3* [8]. Якщо Z3 не зуміє встановити тотожню істинність якоїсь умови, тоді для неї Z3 знаходить контрприклад. Аналізуючи останній, користувач може знайти причину невдачі і внести зміни в анотовану програму.

Система *VerPro* надає інструменти для дослідження проблеми верифікації програм та випробовування таких інструментів аналізу програм, як символьне виконання, зворотне протягування умов, побудови інваріантів тощо. В [3] наведено декілька ранніх реалізацій комп`ютерних систем верифікації програм.

Системи верифікації програм переважно реалізувалися як специфічні системи логічного виводу, що сильно ускладнювало їх побудову та функціонування, однак створювало враження їх універсальності. Дана стаття демонструє, що систему верифікації програм можна ефективно будувати на принципах еквівалентного перетворення виразів.

На відміну від них система *VerPro* надає користувачу сучасний гнучкий діалоговий віконний інтерфейс та звичну інфіксне представлення умов та виразів. Система застосовується до програм та функцій сучасної популярної мови програмування Python. Система базується на строго описаних еквівалентних перетвореннях бібліотеки *ExprLib*.

Для встановлення коректності програми часто недостатньо стандартних співвідношень арифметики та логіки. Інколи для цього приходиться залучати співвідношення, які специфікують предметну область програми чи саму програму. Такими співвідношеннями потрібно доповнювати умови коректності. Спрощення умов коректності розглядається як необхідний початковий етап верифікації, що дозволяє зрозуміти, яких співвідношень не вистачає для верифікації.

Бібліотека *ExprLib* дозволяє перетворювати арифметичні та логічні вирази до найпростішого вигляду. Наявність операцій переходу від внутрішнього представлення виразів деревами до інфіксного представлення надає користувачу можливість контролювати процес перетворення.

Комп'ютерна реалізація еквівалентного перетворення виразів найкраще представлена системою *Mathematica* [7]. Не претендуючи на універсальність, для цілей верифікації програм система *VerPro* надає аналогічні еквівалентні перетворення.

В роботах [10–12] обговорювалася проблема відповідності програми та її анотації. В системі *VerPro* було вперше розглянуто задачу знаходження контрприкладів для умов верифікації, які дозволяють побачити причину невідповідності програми та її анотації, виправити програму або її анотацію і продовжити дослідження.

Висновки

За результатами виконаного дослідження можна зробити такі висновки.

Система *VerPro* надає інструменти для верифікації програм, допомагаючи користувачу в пошуку інваріантів та помилок відповідності програми та її анотації. Система базується на строго визначених еквівалентних перетвореннях виразів, реалізованих функціями та класами бібліотеки *ExprLib*.

Систему верифікації програм можна ефективно будувати на принципах еквівалентного перетворення виразів, доповнюючи їх застосуванням зовнішніх систем доведення теорем та пошуку контрприкладів.

Зручний діалоговий віконний графічний інтерфейс системи *VerPro* дозволяє успішно застосовувати її в ході навчання студентів основам програмування. Бібліотека *ExprLib* та система VerPro також надають хорошу основу для виконання курсових та дипломних робіт.

Література

1. Feys R. (1965). Modal LogicsParis. Gauthier-Villars.
2. Floyd R.W. (1967). Assigning meanings to programs. Proceedings of a Symposium on Applied

Mathematics. 19: Mathematical Aspects of Computer Science. 18–32.
3. Gries D. (1981). The science of programming. Springer-Verlag.
4. Kleene S.C. (2009). Introduction to metamathematics. New York: Ishi Press.
5. Knuth D. (1997). The Art of Computer Programming, vol. 1: Fundamental Algorithms. Addison-Wesley.
6. Lamy R. (2013). Instant SymPy Starter. Packt Publishing.
7. Wolfram Mathematica. https://www.wolfram.com/.
8. Z3 Guide. https://microsoft.github.io/z3guide/docs/logic/intro/.
9. Костирко В. С., Плеша В. І. Застосування бібліотеки Z3PY для перевірки умов коректності та завершимості програм. Матеріали XXVI Міжнародної науково-практичної конференції „Інформаційні технології в економіці, менеджменті і бізнесі. Проблеми науки, практики і освіти“. Київ : Вид-во Європейського університету, 2020. С. 80–83.
10. Stump A. (2016). Verified Functional Programming in Agda. Association for Computing Machinery and Morgan & Claypool. https://doi.org/10.1145/2841316.
11. Vardi, M.Y. (2021). Program verification: vision and reality. Communications of the ACM (CACM), 65 (7). 46–55. https://doi.org/10.1145/3469113.
12. Zilberstein N., Dreyer D., Silva A. (2021). Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. Proceedings of the ACM on Programming Languages. 7 (OOPSLA1). 522–550. https://doi.org/10.1145/3586045.
13. Paulson L.C., Nipkow T., Wenzel M. (2019). From LCF to Isabelle/HOL. Formal Aspects of Computing. 31 (6). 675–698. https://doi.org/10.1007/s00165-019-00492-1.

References

1. Feys R. (1965). Modal LogicsParis. Gauthier-Villars.
2. Floyd R.W. (1967). Assigning meanings to programs. Proceedings of a Symposium on Applied Mathematics. 19: Mathematical Aspects of Computer Science. 18–32.
3. Gries D. (1981). The science of programming. Springer-Verlag.
4. Kleene S.C. (2009). Introduction to metamathematics. New York: Ishi Press.
5. Knuth D. (1997). The Art of Computer Programming, vol. 1: Fundamental Algorithms. Addison-Wesley.
6. Lamy R. (2013). Instant SymPy Starter. Packt Publishing.
7. Wolfram Mathematica. https://www.wolfram.com/.
8. Z3 Guide. https://microsoft.github.io/z3guide/docs/logic/intro/.
9. Kostyrko V. S., Plesha V. I. Zastosuvannia biblioteky Z3PY dlia perevirky umov korektnosti ta zavershymosti prohram. Materialy KhXVI Mizhnarodnoi naukovo-praktychnoi konferentsii „Informatsiini tekhnolohii v ekonomitsi, menedzhmenti i biznesi. Problemy nauky, praktyky i osvity“. Kyiv : Vyd-vo Yevropeiskoho universytetu, 2020. S. 80–83.
10. Stump A. (2016). Verified Functional Programming in Agda. Association for Computing Machinery and Morgan & Claypool. https://doi.org/10.1145/2841316.
11. Vardi, M.Y. (2021). Program verification: vision and reality. Communications of the ACM (CACM), 65 (7). 46–55. https://doi.org/10.1145/3469113.
12. Zilberstein N., Dreyer D., Silva A. (2021). Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. Proceedings of the ACM on Programming Languages. 7 (OOPSLA1). 522–550. https://doi.org/10.1145/3586045.
13. Paulson L.C., Nipkow T., Wenzel M. (2019). From LCF to Isabelle/HOL. Formal Aspects of Computing. 31 (6). 675–698. https://doi.org/10.1007/s00165-019-00492-1.