УДК 681.3

Т.В. Панченко

МЕТОД ДОВЕДЕННЯ ВЛАСТИВОСТЕЙ ПРОГРАМ В КОМПОЗИЦІЙНО-НОМІНАТИВНИХ МОВАХ IPCL

Викладено композиційний метод верифікації систем спеціального класу – моделі багатоекземплярного виконання програм у серверному середовищі з паралелізмом у режимі почергового виконання з переключенням і взаємодією через спільну пам'ять. У роботі специфіковано задачу, побудовано відповідні моделі, сформульовано два варіанти часткової коректності програм на введених композиційних мовах та запропоновано методологію верифікації, що включає метод з лінійною складністю замість експоненційної.

Формулювання задачі. Модель – стандартне серверне середовище. Узагальнення

Щодня ми працюємо з системами, які використовують взаємодію через спільну пам'ять (shared memory concurrency, або паралелізм з почерговим виконанням, на противагу розподіленим системам з передачею повідомлень – distributed systems with message passing) [1, 2]. Це операційні системи, системи управління базами даних, системи з централізованим сховищем даних, служби операційних систем, сервери в клієнт-серверних середовищах тощо. Дійсно, маємо широкий клас середовищ та прикладних систем, які використовують механізм взаємодії через спільну пам'ять. Зрозуміло, що на таких серверних платформах побудовано безліч критичних (мається на увазі – до збоїв у роботі) систем – банківські, медичні, аерокосмічні тощо. Отже, актуальним є питання розробки формального методу для доведення коректності систем, що працюють в перерахованих середовищах.

Існує ряд підходів до вирішення такої задачі [1]. Одними з перших спроб доведення коректності паралельних програм є роботи Ешкрофта [3] та Хоара [4]. Овіцкі та Грісом в [5] зроблено спробу побудувати систему висновків щодо паралельних програм шляхом узагальнення методу Хоара. Ця спроба набула подальшого розвитку, зокрема Джонс [6, 7] перетворив метод у композиційний [8]. Харел і Пнуелі акцентували увагу на зміщенні акценту від функціональних систем до реактивних [9], коли мова йде про паралельне виконання. Так, Пнуелі розвинув підхід Ешкрофта введенням темпоральної логіки для суджень про паралельні програми [10]. Лемпорт узагальнив твердження про стани до акцій (actions) – тверджень про пари станів, або переходи (transitions) [11]. Подальший розвиток цього підходу – мова Unity [12] Ченді та Місри. Щодо загальної картини, одні методи (наприклад, TLA [13, 14]) програють композиційним методам [15 – 18] за рядом параметрів (розрив між специфікацією та реальною програмою, підтримка механізму абстракції тощо [1]), а інші (метод Овіцкі–Гріса [5] та його варіанти, що є розвитком аксіоматичного методу Хоара [19, 4]) є надто складними для застосування в реальних системах. Останні також ускладнюють доведення – збільшують кількість кроків у випадку "чистого" методу Овіцкі–Гріса (квадратична складність) або ускладнюють побудову проміжних тверджень (в методах типу rely-guarantee) [1]. Аксіоматичні методи вважаються взагалі більш складними у застосуванні на практиці щодо паралельних систем. Отже, перевага надається методам, що базуються на моделюванні та використовують принцип інваріанта (незмінної або постійної властивості) у своїй основі.

Проаналізуємо серверну архітектуру, наприклад, роботу Web-сервера чи SQL-сервера. Так, на будь-якому сервері, як правило, запущені служби, що чекають

вхідних сигналів і реагують на них шляхом запуску відповідного процесу або виконанням певних дій. Нехай $A, B, \ldots, C$ – відповідні реакції на різні види вхідних сигналів (запитів). (Власне, це і є задача сервера – опрацьовувати запити і давати на них відповіді). Оскільки служби "слухають" вхідний канал (або вхідні канали) постійно (тобто – чекають вхідних сигналів) і при надходженні запиту одразу запускають відповідний процес-обробник, може виникнути (і реально – часто виникає) ситуація, коли один і той же процес має кілька екземплярів (інстанцій – many instances – тобто виявляється запущеним декілька разів $A^n, B^m, \ldots C^k$). Причому, оскільки служби працюють в певному операційному середовищі сервера, яке в переважній більшості випадків (Windows NT, Windows 2000, Windows 2003 Server, FreeBSD, Linux та ін.) підтримує конкурентний ("витискаючий") паралелізм – фактично, паралелізм з почерговим виконанням – має місце ситуація $A^n \parallel B^m \parallel \ldots \parallel C^k$. Іншими словами, зазвичай ми запускаємо послідовні (на противагу паралельним) програми в багатозадачному серверному середовищі з паралелізмом з почерговим виконанням в режимі переривання.

Зауваження щодо позначень:

1) будемо позначати композицію паралельного виконання в режимі чергування, як засіб побудови програми, $f \parallel g$, хоча ця композиція має розумітись як паралельне виконання в режимі чергування (interleaving) у формі $P \parallel\parallel Q$ в сенсі [4]. Однак обрана нотація ближча до позначень у KIV [20, 21] – методиці та однойменному наборі утиліт для доведення властивостей програм, які переважно ґрунтуються на перевірці моделі;

2) $P \parallel P \parallel \ldots \parallel P$ (n раз), тобто паралельне виконання в режимі чергування та переключення n програм P, будемо позначати P^n.

Процеси $A, B, \ldots, C$ – це програми, написані в деякій реальній мові програмування. Практика показує, що самі процеси дуже рідко вимагають паралелізму в рамках них самих (тобто для їх написання і реалізації), а отже, програми $A, B, \ldots, C$, як правило, – послідовні. На користь цього свідчить наступне. Техніка програмування, в якій допускається розпаралелювання одного процесу на декілька (під час його виконання), називається багатопотоковістю (multithreading) [4]. Взагалі, багатопотоковість – складна та схильна до помилок техніка, яка не рекомендується до використання, окрім, найменших (найпростіших) програм [4]. Як підсумок, Тоні Хоар в своїй праці [4] пише, що паралелізм може бути дозволений лише на найбільш зовнішньому (найбільш глобальному, найвищому) рівні роботи (завдання, програми), а його використання на нижчих (вкладених) рівнях слід уникати [4].

До речі, якщо подивитись на сучасну ситуацію з hardware, а саме з процесорами (CPU), то на сьогодні нарощування обчислювальної потужності здійснюється шляхом нарощування кількості ядер процесору, а не потужності самого ядра (див. повідомлення відомих виробників – Intel, AMD). Відомо, що щільність розміщення елементів на базі (транзисторів, тригерів на кремнієвому кристалі) вже сягнула тієї межі, яка освоєна практично на сучасному рівні розвитку фізики та технології. Моделлю цієї процесорної взаємодії, виступає паралелізм зі спільною пам'яттю, адже всі ядра (як і всі процесори в мультипроцесорних комплексах) мають доступ до єдиного спільного пристрою пам'яті, який вони ділять між собою.

Уточнення задачі. Мова IPCL. Синтаксис

Розглянемо клас композиційних мов *IPCL* (Interleaving Parallel Composition Languages) з паралелізмом в режимі чергування [22]. Синтаксис таких мов задається таким виглядом:

$P ::= \bar{x} := \bar{e} \mid P_1;P_2 \mid$ **if** b **then** P_1 **else** $P_2 \mid$ **while** b **do** $P \mid P_1 \parallel P_2$,

де оператор присвоювання є векторним та атомарним, композиції послідовного виконання, розгалуження та циклування розуміються стандартним чином [15–18] (причому в операторах розгалуження та циклування умова обчислюється атомарно, але виконання може бути перерване після обчислення умови і до вико-

нання першої дії з відповідного програмного блоку [8]), а паралельна композиція розуміється як недетерміноване попереміння виконання (interleaving) послідовних атомарних дій підпроцесів, які є аргументами композиції (наприклад, в стандартному розумінні [8]). Також будемо опускати зайві дужки в нотації, зокрема: $P_1 \| (P_2 \| (\dots \| P_n)\dots)$ будемо записувати як $P_1 \| P_2 \| \dots \| P_n$.

В роботі [8] вводиться ще одна композиція – **await** *b* **then** *P* **end**, яка слугує для синхронізації та взаємного виключення (якщо умова *b* істинна, то *P* виконується одразу і без переривань, якщо ні – то *P* так само виконується, як тільки *b* набуде значення істини), але вона відсутня в *IPCL*. Це пояснюється тим, що по-перше, така композиція відсутня в деяких мовах, які працюють в паралельному середовищі (наприклад, SQL), по-друге, її можна за необхідності промоделювати через інші базові композиції (шляхом введення додаткових змінних).

Композиційна семантика мов класу IPCL. Паралелізм в композиційному програмуванні

Мови класу *IPCL* ґрунтуються на композиційно-номінативних системах $< NDS, F, \{1, 2\}, C, arity >$. Тут *NDS* – система іменних або номінативних даних (надалі – просто дані), *F* – набір функцій для перетворення даних, *C* – композиції над функціями з *F*. $F = Oper \cup Pred$, де $Oper = D \times D \to D \times D$, а $Pred = D \times D \to \{\text{True, False}\}$, де перше входження *D* в декартів добуток – "глобальні дані", а друге – "локальні" для поточного процесу; функції, що повертають значення з множини {True, False} (предикати), не змінюють поточний стан даних $D \times D$ – вони використовуються як умова в операторах розгалуження та циклування; $D = ND(V,W)$ у звичайному сенсі. Композиціями є:

- послідовне виконання, позначається ";" – аналог звичайної композиції множення або аплікування, послідовної функціональної композиції (sequential functional composition) з абстрактним трактуванням "$\circ$" [15];

- розгалуження, позначається "**if**" – аналог композиції "◊" [15];

- циклування, позначається "**while**" – аналог композиції "$*$" [15];

- присвоювання, позначається ":=" – аналог сукупності (певної комбінації) композицій іменування та розіменування [15] і деяких обчислень;

- паралельне виконання в режимі чергування (переривання, interleaving, shared time), позначається "||".

Зауважимо, що базовою в *IPCL* є композиція послідовного виконання (sequential composition [15]) "$\circ$", яку для зручності позначаємо ";", при застосуванні якої її аргументи вважаються такими, що беруть на вхід дане і повертають повне дане (а не лише зміни у ньому). Ця композиція використовується при абстрактному трактуванні послідовного виконання. Можна було використовувати композицію поступового виконання (succession composition ";" в [15]), яка з'являється при номінативному трактуванні послідовного виконання – в цьому випадку аргументи композиції вказують лише на зміни, що відбулися у вхідному даному. Але абстрактна більш адекватна подальшого моделювання, відображає по суті перетворення стану пам'яті (даних), а не лише його частини, яке потім необхідно накласти (∇) на поточний стан пам'яті (даних), щоб отримати цілковиту картину після перетворення.

Визначимо композиційну семантику мов *IPCL*. Семантика трьох перших композицій є стандартною, семантика композиції паралельного виконання в режимі чергування визначається через семантику її аргументів та синтаксичний контекст[1] традиційним чином, враховуючи її алгебраїчні та семантичні властивості. Індексом-параметром функції семантики виступає синтаксичний контекст – фрагмент програми або сама програма. Отже

семантика ";":

$sem_{A\,;\,B}(d) = d\ \nabla'\ sem_A(d)\ \nabla'\ sem_B(d\ \nabla'\ sem_A(d))$,

[1] пріоритет композиції || найменший, це треба враховувати при читанні виразів та розстановці дужок

де ∇' – покомпонентне накладання декартового добутку звичайних іменних (номінативних) даних:

$$d_1 \nabla' d_2 = (Pr_1(d_1) \nabla Pr_1(d_2), Pr_2(d_1) \nabla Pr_2(d_2)).$$

$Pr_i(d)$ – проекція декартового добутку d за i-ю компонентою, $d_1, d_2 \in D \times D$,

(така невелика видозміна викликана тим, що дане розглядається як двокомпонентне – таке, що містить "глобальну" і "локальну" частини), тут ∇ – звичайна операція накладання даних (двох іменних або номінативних множин) згідно з [15–18],

семантика ":=": $sem_{x := e}(d) = d \nabla' f(d)$,

де $f \in Oper$ – (семантична) функція, яка відповідає синтаксичному оператору $x := e$, причому саме присвоєння (результат f) дає двокомпонентний результат, адже обчислюються глобальні та локальні дані, а накладання відбувається на відповідні компоненти даного d: глобальні змінні – на першу компоненту; локальні – на другу; x та e – вектори імен та значень (виразів) відповідно,

де $A, B, C, P, P', Q, R \in Terms$ – програми в мові *IPCL* (терми в алгебрі $IPCL_A$ [22]), b – синтаксичне позначення відповідної умови предиката *pred* з *Pred*, тобто $b(d) = sem_b(d) = pred(d)$, а $d \in D \times D$.

В усіх наведених визначеннях, якщо значення $b(d)$ – невизначене, то і значення лівої частини відповідної рівності буде також невизначеним. Так само, якщо в будь-якому з визначень значення правої частини рівності не визначено, то і значення лівої частини – невизначено.

Зауважимо, що семантика задається в синтактико-семантичному підході, адже вона визначається не тільки через безпосередні компоненти (складові) конструкції, а суттєво враховує структуру (тобто їх взаємне розташування, особливо щодо композиції паралелізму), що взагалі кажучи, характерно для систем з паралелізмом. Глінн Вінскел зазначає у своїй роботі [23], що неможливо змоделювати паралельне виконання програми, обмежившись зв'язками між конфігураціями команд та заключними станами – тоді необхідно вико-

семантика "**if**":

$sem_{\textbf{if } b \textbf{ then } P \textbf{ else } Q}(d) = sem_P(d)$, якщо $b(d) = \text{True}$,

$sem_{\textbf{if } b \textbf{ then } P \textbf{ else } Q}(d) = sem_Q(d)$, якщо $b(d) = \text{False}$,

семантика "**while**":

$sem_{\textbf{while } b \textbf{ do } P}(d) = sem_{P\,;\,\textbf{while } b \textbf{ do } P}(d)$, якщо $b(d) = \text{True}$,

$sem_{\textbf{while } b \textbf{ do } P}(d) = d$, якщо $b(d) = \text{False}$,

далі – семантика "||":

|| – асоціативна і комутативна:

$sem_{(A \,\|\, B) \,\|\, C}(d) = sem_{A \,\|\, (B \,\|\, C)}(d)$

$sem_{A \,\|\, B}(d) = sem_{B \,\|\, A}(d)$

на синтаксичному рівні це означає відповідно:

$((A \,\|\, B) \,\|\, C)(d) = (A \,\|\, (B \,\|\, C))(d)$

$(A \,\|\, B)(d) = (B \,\|\, A)(d)$

|| відносно "**if**":

$sem_{\textbf{if } b \textbf{ then } P \textbf{ else } Q \,\|\, R}(d) = sem_{P \,\|\, R}(d)$, якщо $b(d) = \text{True}$,

$sem_{\textbf{if } b \textbf{ then } P \textbf{ else } Q \,\|\, R}(d) = sem_{Q \,\|\, R}(d)$, якщо $b(d) = \text{False}$,

$sem_{\textbf{if } b \textbf{ then } P \textbf{ else } Q\,;\,P' \,\|\, R}(d) = sem_{P\,;\,P' \,\|\, R}(d)$, якщо $b(d) = \text{True}$,

$sem_{\textbf{if } b \textbf{ then } P \textbf{ else } Q\,;\,P' \,\|\, R}(d) = sem_{Q\,;\,P' \,\|\, R}(d)$, якщо $b(d) = \text{False}$,

|| відносно "**while**":

$sem_{\textbf{while } b \textbf{ do } P \,\|\, R}(d) = sem_{P\,;\,\textbf{while } b \textbf{ do } P \,\|\, R}(d)$, якщо $b(d) = \text{True}$,

$sem_{\textbf{while } b \textbf{ do } P \,\|\, R}(d) = sem_R(d)$, якщо $b(d) = \text{False}$,

$sem_{\textbf{while } b \textbf{ do } P\,;\,P' \,\|\, R}(d) = sem_{P\,;\,\textbf{while } b \textbf{ do } P\,;\,P' \,\|\, R}(d)$, якщо $b(d) = \text{True}$,

$sem_{\textbf{while } b \textbf{ do } P\,;\,P' \,\|\, R}(d) = sem_{P' \,\|\, R}(d)$, якщо $b(d) = \text{False}$,

|| відносно ";":

$sem_{(A\,;\,B) \,\|\, P}(d) = sem_{A\,;\,(B \,\|\, P)}(d)$,

$sem_{A \,\|\, P}(d) = sem_{A\,;\,P}(d)$,

ристовувати зв'язки між окремими атомарними (неподільними) кроками при виконанні і такою дією дозволяти одній команді впливати на стан іншої команди, з якою вона виконується в паралель. Взагалі, семантика систем з паралельним виконанням у режимі чергування, як правило, визначається саме в синтактико-семантичних традиціях, адже цей підхід є більш природнім для розуміння подібних систем.

Функції з F є атомарними транзакціями, неподільними в сенсі паралелізму, їх виконання не може бути перервано. Отже, зокрема умови у відповідних композиціях теж атомарні.

Конкретна мова класу *IPCL* утворюється шляхом фіксації F.

Легко показати, що семантика описана повно – тобто, що вона допускає всі можливі трактування на моделях. Для цього достатньо довести, що семантика спорідненої мови (без оператора **await**) з [8] виражається в термінах семантики *IPCL* – для доведення необхідно використати індукцію за структурою програми. До речі, в [8] показано, що семантика наведеної мови композиційна, звідси семантика *IPCL* теж композиційна (це можна довести, виразивши семантичну функцію *sem* у термінах семантики мови з [8]). Ці два взаємні вираження семантик показують, що завдання семантики *IPCL* коректні та повні (щодо загальновживаної семантики аналогічної мови з [8]) – семантичні та прагматичні.

Уточнення – звуження класу

З усіх можливих програм розглядатимемо клас багатоекземплярного виконання послідовних програм у серверному середовищі (у режимі почергового переключення між ними): $SeqILProgs = \{ Prog \mid Prog = A^n \parallel B^m \parallel \ldots \parallel C^k;$
$A, B, \ldots, C \in SeqTerms; \quad n, m, \ldots, k \in Z^+ \}$.
Z^+ – додатні цілі числа, *SeqTerms* – програми в мові *IPCL* (терми в алгебрі $IPCL_A$ [22]), побудовані за допомогою композицій {:=, ;, **if**, **while**}, без застосування композиції ||.

Даний клас програм, хоча і є звуженням класу всіх програм (термів) у мові *IPCL* (в алгебрі $IPCL_A$), насправді, досить широкий з практичної точки зору. Детальне обґрунтування цього факту вище наведено. До речі, в [5] та [24] розглядається саме таке звуження мови (класу програм), точніше – навіть звуження цього звуження, адже там степені мають бути фіксованими (деталі буде проаналізовано після викладення методу).

Сформулюємо види властивостей щодо коректності програмного за безпечення у мовах *IPCL*.

Два види властивостей програм

Зафіксуємо задачу. Перевірити (довести) виконуваність властивості для програми $Prog \in SeqILProgs$. Розглянемо два варіанти цієї задачі.

1) (властивість 1 типу), яку треба довести, записуємо у вигляді $\{PreCond\}\ Prog\ \{PostCond\}$[2] з властивості передумови *PreCond* після виконання програми *Prog* випливатиме післяумова *PostCond*;

2) (властивість 2 типу), яку треба довести – інваріантна, тобто вона має виконуватись протягом всього часу роботи програми *Prog*. (Цей тип властивості буде уточнено далі).

Важливо зазначити, що будь-яку властивість можна записати у вигляді предиката її істинності. Отже, виконуваність властивості еквівалентна істинності відповідного їй предиката. Обидва варіанти властивостей є варіаціями поняття властивості "безпеки" (**safety property**). В даній роботі зупинимось на розгляді лише цих типів властивостей.

Отже, перший варіант відображає традиційні погляди (зокрема, Хоара) на часткову коректність (*послідовних* !) програм. Другий варіант "лежить в руслі" переходу до інваріанту замість перед- та постумов і відображає погляди Ешкрофта-Лемпорта на коректність (*паралельних* !) програм. (В роботі [1] детально описано еволюцію поглядів щодо коректності паралельних програм).

Уточнимо ці варіанти. Перший варіант – це узагальнена часткова коректність програми. Тобто припускаємо, що вхідні

[2] в сенсі трійок Тоні Хоара (C.A.R. Hoare), тобто: передумова – дія – постумова

дані (точніше – початковий стан програми) задовольняє деякій умові *PreCond*, і необхідно довести, що після виконання програми буде виконуватись умова *PostCond*, тобто вихідні дані (точніше – заключний стан при виконанні програми) задовольняють цій умові. Другий варіант – більш сильна вимога до програми. Доведення такої властивості буде гарантувати, що деяка умова виконується після виконання кожної дії програми – обчислення значення базової функції з F та/або передачі керування на будь-якому кроці згідно композиційної структури програми – (а, значить, і після завершення виконання, зокрема) за умови, що вона мала місце на початку її роботи.

Методика розв'язання задачі (доведення властивостей)

Ураховуючи комутативність та асоціативність композиції ||, а також її семантику відносно інших композицій, процес обчислень програми $P \in SeqILProgs$ можна уявити як послідовне виконання, причому на кожному кроці виконується одна чергова команда однієї з підпрограм, причому підпрограма обирається випадково з тих, які ще недовиконані. Можна також представити процес обчислень з синтаксичної точки зору як переписування терму програми з метою усунення композиції || з її нотації. Оскільки процедура переписування (перетворення) терму, враховуючи вищенаведену семантику композиції ||, є недетермінованою, на кожному кроці робиться вибір правила переписування як однієї з альтернатив. У цьому випадку можна говорити, що цим самим обирається один з можливих шляхів виконання програми P. Саме виконання елементарних (атомарних) дій – функції з F – це, власне, перетворення даних. Кожна підпрограма може взаємодіяти зі спільними глобальними та своїми локальними даними.

Побудова моделі. Транзиційна система

Побудуємо модель виконання довільної програми $P \in SeqILProgs$ у відповідності з вищенаведеними міркуваннями та згідно семантики мов *IPCL*.

Отже, розглянемо програму $P = A^n \parallel B^m \parallel \ldots \parallel C^k \in SeqILProgs$. Нехай $numprocs=n+m+\ldots+k$. Зробимо помітки для кожного входження функцій з F в терми програм $A, B, \ldots, C$. Також для кожної програми $A, B, \ldots, C$ введемо помітку завершення роботи програми – помітку, що знаходиться після тексту програми, не помічає жодну функцію із програми, але фіксує стан одразу після завершення виконання програми – аналог E з [8]. Маємо для кожної програми $A, B, \ldots, C$ множину поміток $A_{marks}, B_{marks}, \ldots, C_{marks}$ відповідно. Зрозуміло, що глобальне сховище (простір спільних змінних) має чітко відрізнятися та прозоро відокремлюватися від локального сховища (простору локальних змінних) [4]. Тоді $S \in A_{marks}{}^n \times B_{marks}{}^m \times \ldots \times C_{marks}{}^k \times D \times \times D^{numprocs}$ будемо називати **станом системи**, що виконує програму P, або **станом програми** P (або просто **станом**, якщо з контексту зрозуміло, про який стан іде мова), де перше входження D в декартів добуток – "глобальні дані", а друге – "локальні" для всіх окремих послідовних підпрограм $(A^n, B^m, \ldots, C^k)$ з P, записані в тій же послідовності, що і самі підпрограми в термі програми P. Перша частина декартового добутку (до даних) – це позиції виконання кожної підпрограми з P (мітки функцій, які будуть обчислені – або дій, які будуть виконані – відповідними підпрограмами, коли дійде черга до їх виконання – тобто відбудеться "переключення на них", на найближчому кроці). Весь наведений декартів добуток будемо називати **множиною станів** і позначати *States*. Отже, стан $S \in States$ містить в собі інформацію як про стан даних (глобальних і локальних), так і про стан керування, тому цей сукупний стан ще іноді називають конфігурацією.

Будемо виділяти множину **початкових станів** $StartStates \subseteq States$. Зокрема, в усіх елементів цієї множини перші (*numprocs*) компоненти декартового добутку, які відображають мітки, (всі окрім да-

них) мають значення початкових (перших, які зустрічаються в записі терму) міток відповідних процесів. Можна також виділити множину **заключних станів**, в один з яких програма P потрапляє після завершення виконання – $StopStates \subseteq States$. Для всіх елементів s цієї множини характерно, що ті ж самі перші (*numprocs*) їх компоненти будуть мати значення заключних (останніх в записі терму) міток відповідних процесів, а також їх досяжність. Останнє означає, що для кожного стану $s \in StopStates$ існує набір станів $s_1, s_2, \ldots, \ldots, s_l \in States$ такий, що

$$s_1 \in StartStates \;\&\; s_l = s \;\&\; (\forall i \in \{1, 2, \ldots, \ldots, l-1\} \bullet (s_i, s_{i+1}) \in Step),$$

де $Step$ – функція кроку виконання програми P (про функцію $Step$, про розстановку міток див. далі).

Кількість підпрограм програми P, що знаходяться на деякій позиції $mark$ в стані S будемо позначати $P_{S[mark]}$. Такі величини, що є цілими невід'ємними числами – корисні для формулювання перед- та постумов, а також інваріантів, при доведенні властивостей. Наприклад, для довільної програми $A \in SeqTerms$ і стану $S \in States$ завжди виконується $\sum_{mark \in A_{marks}} A^n{}_{S[mark]} = n$.

З цього зауваження зрозуміло, що ми дозволяємо явно використовувати стан керування в формулюванні предикатів *PreCond*, *PostCond* та *Inv*.

Функцію $Step: States \to States$ будемо називати **функцією кроку виконання програми** P, якщо вона визначає всі можливі перетворення станів, тобто переходи від одного до іншого стану – за один крок під час виконання програми P і тільки їх. Тобто, якщо поточний стан програми P є S, то за один крок виконання програми P потрапляємо в стан $S' \in Step(S)$. Під кроком виконання програми P (в певний момент) розуміємо обчислення однієї з функцій із F, які доступні до обчислення згідно композиційної семантики програми (в цей момент).

З виконанням кожного кроку програми P, тобто обчисленням деякої функції, окрім можливої зміни даних, пов'язана зміна значення однієї з перших компонент стану – а саме компоненти, що вказувала на мітку обчисленої функції. Її значення стає рівним мітці, куди програма потрапляє після обчислення функції з поточного кроку. Значення нової мітки визначається за семантикою композиційного контексту даної функції.

Функція $Step$, очевидно, багатозначна (недетермінована) часткова функція. (З точки зору теорії множин її скоріше слід було б назвати відношенням, тобто $Step \subseteq States \times States$.)

Так, маємо справу з **транзиційною системою** (State Transition System), що складається з множини всіх станів $States$ та функції переходів $Step$. Точніше – ініціалізована нескінченна (за кількістю станів) нерозмічена транзиційна система, initialized infinite unlabelled transition system, в сенсі, наприклад, ASM [25]. Також додатково ми виділяємо множину початкових станів $StartStates$ та множину заключних станів $StopStates$, які є підмножинами $States$. Зрозуміло, що для кожної програми з $SeqILProgs$ можна побудувати відповідну транзиційну систему. Отже, будемо розглядати такі транзиційні системи в якості моделей програм.

Тепер, визначивши модель програми – як транзиційну систему – та повертаючись до визначення *PreCond*, *PostCond* та *Inv*, слід уточнити, що ці умови є предикатами над станом, тобто

$PreCond, PostCond, Inv : States \to \mathbf{Bool}$, де $\mathbf{Bool} = \{\text{True}, \text{False}\}$.

Розстановка міток та визначення функції Step

Наведемо рекурсивний (за структурою терму коректно побудованої програми) алгоритм визначення функції зміни (точніше – розстановки) міток і одночасного визначення функції $Step$.

```
функція Розставити_мітки ( початкова_мітка, кінцева_мітка, терм )
{
   якщо ( терм = терм1 ; терм2 )
   то
   {
      створити  нова_мітка;
      розставити_мітки ( початкова_мітка, нова_мітка, терм1 );
      розставити_мітки ( нова_мітка, кінцева_мітка, терм2 )
   }
   інакше
   якщо ( терм = if ( терм1 ) then терм2 else терм3) )
   то
   {
      створити  нова_мітка1, нова_мітка2;
      створити_перехід_з_умовою ( початкова_мітка, терм1(d)=True, нова_мітка1 );
      розставити_мітки ( нова_мітка1, кінцева_мітка, терм2 );
      створити_перехід_з_умовою ( початкова_мітка, терм1(d)=False, нова_мітка2 );
      розставити_мітки ( нова_мітка2, кінцева_мітка, терм3 )
   }
   інакше
   якщо ( терм = while ( терм1 ) do терм2 )
   то
   {
      створити  нова_мітка;
      створити_перехід_з_умовою ( початкова_мітка, терм1(d)=True, нова_мітка );
      створити_перехід_з_умовою ( початкова_мітка, терм1(d)=False, кінцева_мітка );
      розставити_мітки ( нова_мітка, початкова_мітка, терм2 )
   }
   інакше
   якщо ( терм = x := e )
   то
      створити_перехід_з_перетворенням( початкова_мітка, f, кінцева_мітка )
}
```

де f – семантика відповідної операції $x := e$.

Кожній функції, поміченій міткою (присвоювання або умова), відповідає її семантика з множини F згідно з вищевизначеною функцією семантики *sem*.

Основний виклик функції розставити_мітки() треба виконати для кожного з варіантів підпроцесів P, тобто для $A, B, \ldots, C$ (по одному екземпляру для кожного варіанта підпроцесів) у нашому попередньому визначенні – даючи кожного разу вхідними параметрами різні (всі попарно відмінні) початкові й кінцеві мітки, а термом – відповідний терм $A, B, \ldots, C$.

Тут функції створити_перехід_з_умовою (*мітка*$_1$, *умова*, *мітка*$_2$) та створити_перехід_з_перетворенням (*мітка*$_1$, *функція*, *мітка*$_2$) означають уточнення функції *Step* (поповнення декартового добутку *Step* парами (*State*, *State'*)), а саме – розглянемо, наприклад обробку терму A.

Так, створити_перехід_з_умовою (*мітка*$_1$, *умова*, *мітка*$_2$) додасть до *Step* наступну множину пар:

$$\{ (S_1, S_2) \mid S_1=(a_1, a_2, \ldots, a_n, b, \ldots, c, d, d_1, \ldots, d_{numprocs}) \;\&\; S_2=(a_1', a_2', \ldots, a_n', b, \ldots, c, d, d_1, \ldots, d_{numprocs}) \;\&\; \forall i \in N_n \bullet (a_i \in A_{marks} \;\&\; a_i' \in A_{marks}) \;\&\; b \in B_{marks}^{\,m} \;\&\; \ldots \;\&\; c \in C_{marks}^{\,k} \;\&\; d \in D \;\&\; \forall i \in N_{numprocs} \bullet d_i \in D \;\&\; \exists i \in N_n \bullet \forall j \in N_n \setminus \{i\} \bullet (a_i = мітка_1 \;\&\; a_i' = мітка_2 \;\&\; a_j' = a_j) \;\&\; умова(S_1) \},$$

де $N_n=\{1, 2, \ldots, n\}$, а створити_перехід_з_перетворенням(*мітка*$_1$, *функція*, *мітка*$_2$) –

$\{ (S_1, S_2) \mid S_1=(a_1, a_2, \ldots, a_n, b, \ldots, c, d, d_1, \ldots, d_{numprocs}) \ \&\ S_2=(a_1', a_2', \ldots, a_n', b, \ldots, c, d', d_1', \ldots, d_n', d_{n+1}, \ldots, d_{numprocs}) \ \&\ \forall i \in N_n \bullet (a_i \in A_{marks} \ \&\ a_i' \in A_{marks}) \ \&\ b \in B_{marks}^{\ m} \ \&\ \ldots \ \&\ c \in C_{marks}^{\ k} \ \&\ d \in D \ \&\ d' \in D \ \&\ \forall i \in N_{numprocs} \bullet d_i \in D \ \&\ \forall i \in N_n \bullet d_i' \in D \ \&\ \exists i \in N_n \bullet \forall j \in N_n \setminus \{i\} \bullet (a_i = мітка_1 \ \&\ a_i' = мітка_2 \ \&\ a_j' = a_j \ \&\ d_j' = d_j \ \&\ (d', d_i') = функція(d, d_i)) \}$.

Аналогічно легко можна виписати ці дві функції, що довизначають *Step*, для інших підпроцесів P : *B*, …, *C*. Очевидно, що для різних підпроцесів вони будуть змінювати деяку частину декартового добутку стану (а саме – не більше трьох компонент), залишаючи інші незмінними. Так можна промоделювати виконання всієї програми *P* як почергове виконання деякої з поточних операцій (обчислення значень функцій), зокрема, перевірку умови і перехід у відповідності з результатом до блоку "then" або до блоку "else" і відповідну зміну стану[3].

Спрощена модель (відсутність локальних даних)

Слід зауважити, що у випадку відсутності локальних даних у програм *A, B*, …, *C* (підпрограм *P*), доцільно розглядати поняття спрощеного стану [26] – агрегованого стану такого вигляду: $N^{Pmq} \times D$, де $Pmq = ||A_{marks}|| + ||B_{marks}|| + \ldots + ||C_{marks}||$.[4] Перші $||A_{marks}||$ компонент такого стану містять кількість програм, що перебувають у відповідних мітках програми *A* в цьому стані, причому сума цих компонент в кожному стані для програми *P* дорівнює *n* – кількості інстанцій (екземплярів) програми *A* в *P* – в термінах програми *P*. Наступні $||B_{marks}||$ компонент – містять кількість програм, що перебувають у відповідних мітках програми *B* у цьому стані, причому сума цих компонент у кожному стані для програми *P* дорівнює *m* (кількості інстанцій програми *B* в *P*) у термінах програми *P* і т.д. Остання компонента *D* містить глобальні спільні дані для всіх підпрограм *P*.

Спрощений стан оперує з кількостями програм, що знаходяться на кожній мітці виконання замість ідентифікації позиції (мітки) кожної інстанції кожної окремої підпрограми (*A, B*, …, *C*). Насправді, підпрограми є нерозрізнюваними з точністю до поточної мітки виконання, якщо вони не змінюють (фактично, не мають) локальних даних, а лише оперують зі спільними глобальними даними.

Отримати спрощений стан за станом можна наступним чином. Оскільки множини $A_{marks}, B_{marks}, \ldots, C_{marks}$ – скінченні, то нехай $A_{marks}=\{A_1, \ldots, A_a\}$, …, $C_{marks}=\{C_1, \ldots, C_c\}$. $Pr_i(S)$ – *i*-а компонента кортежу *S*. Візьмемо деякий стан *S*. Тоді відповідний йому спрощений стан *SS* має вигляд $(a_1, \ldots, a_a, \ldots, c_1, \ldots, c_c, d)$, де $a_j = ||\{ i \mid Pr_i(S)=A_j , i \in N_{Pmq} \}|| = P_{S[A_j]}$, $\forall j \in N_a$, …, $c_j = ||\{ i \mid Pr_i(S)=C_j , i \in N_{Pmq} \}|| = P_{S[C_j]}$, $\forall j \in N_c$, $d = Pr_{numprocs+1}(S)$.

Відповідним чином вводяться множина спрощених станів *SStates*, спрощені початкові стани *SStartStates* (всі вони мають структуру $(n, 0, \ldots, 0, \ldots, k, 0, \ldots, 0, d)$), спрощені заключні стани *SStopStates* (їх структура – $(0, \ldots, 0, n, \ldots, 0, \ldots, 0, k, d)$) та функція кроку виконання програми *P* над спрощеними станами – *SStep*: *SStates*$\rightarrow$ $\rightarrow$ *SStates*. Ця функція зменшує на одиницю значення деякої компоненти вектора *SStates* (наприклад, першу) і водночас збільшує значення деякої іншої (наприклад, другої), тобто відбувається передача керування від однієї мітки (в даному випадку – A_1) до іншої мітки (в даному випадку – A_2) для однієї з програм (в даному випадку – *A*), а також змінює значення останньої компоненти (глобальних даних), якщо чергова функція належить підкласу *Oper* класу *F*. Всі ці об'єкти легко отримати шляхом переведення станів у спрощені стани або безпосередньо.

Зокрема, створити_перехід_з_умовою($мітка_1$, *умова*, $мітка_2$) додасть до *SStep* наступну множину пар:

$\{ (SS_1, SS_2) \mid SS_1=(a_1, a_2, \ldots, a_a, \ldots, c_1, \ldots, c_c, d) \ \&\ SS_2=(a_1', a_2', \ldots, a_a', \ldots, c_1, \ldots,$

[3] Запропонований апарат моделювання, зокрема, дозволяє виразити "миттєвий" **if** – якщо вважати, що на одному кроці обчислень відбувається обрахунок значення умови композиції **if** та одразу передача управління відповідному блоку ("**then**" або "**else**") і виконання першого оператора цього блоку, як, наприклад, відбувається в KIV Tools [175,176] – тобто, по суті, є більш потужним, ніж запропонована семантика *IPCL*. Також цей апарат дозволяє промоделювати оператор **await** в явному вигляді (без проміжного моделювання в мові *IPCL*).

[4] $||A|| = card(A)$ – потужність множини *A*

c_c, d) & $\forall i \in N_a \bullet$ ($a_i \in N$ & $a_i' \in N$) & … & $\forall i \in N_c \bullet$ $c_i \in N$ & $d \in D$ & $\exists i \in N_a \bullet \exists j \in N_a \bullet \forall k \in N_a \setminus \{i, j\} \bullet$ (A_i=*мітка*$_1$ & A_j=*мітка*$_2$ & $a_i > 0$ & $a_i - 1 = a_i'$ & $a_j + 1 = a_j'$ & $a_k = a_k'$) & *умова*(SS_1) },

де $N_n = \{1, 2, \ldots, n\}$, а створити_перехід_з_перетворенням(*мітка*$_1$, *функція*, *мітка*$_2$) –

{ (SS_1, SS_2) | $SS_1 = (a_1, a_2, \ldots, a_a, \ldots, c_1, \ldots, c_c, d)$ & $SS_2 = (a_1', a_2', \ldots, a_a', \ldots, c_1, \ldots, c_c, d')$ & $\forall i \in N_a \bullet$ ($a_i \in N$ & $a_i' \in N$) & … & $\forall i \in N_c \bullet$ $c_i \in N$ & $d \in D$ & $d' \in D$ & $\exists i \in N_a \bullet \exists j \in N_a \bullet \forall k \in N_a \setminus \{i, j\} \bullet$ (A_i=*мітка*$_1$ & A_j=*мітка*$_2$ & $a_i > 0$ & $a_i - 1 = a_i'$ & $a_j + 1 = a_j'$ & $a_k = a_k'$) & (d', [])=*функція*(d, []) },

де [] – порожнє іменне дане.

Також аналогічно до описаної далі техніки доводяться властивості двох типів, які досліджуються [26]. Будемо розглядати спрощені стани, де це доцільно [26].

Метод доведення властивостей

Отже, нехай нам потрібно довести властивість 1 типу з передумовою-предикатом *PreCond* та постумовою-предикатом *PostCond*, або властивість 2 типу з інваріантом-предикатом *Inv*, щодо деякої програми *P*. Всі три предикати є предикатами від аргументу-стану (або спрощеного стану).

Покажемо спочатку, що ці два типи властивостей рівнопотужні в тому сенсі, що коли виконується умова 1 типу для деяких *PreCond* та *PostCond*, то завжди можна підібрати такий *Inv* (що залежатиме від *PreCond* та *PostCond*), для якого буде виконуватись умова 2 типу. Будемо використовувати поняття рефлексивно-транзитивного замикання відношення *Step* (або *SStep*) і позначати його *Step** (або *SStep**). Якщо відношення *Step* моделює виконання одного кроку програми *P*, то *Step** моделює виконання довільної кількості кроків програми *P* (зокрема, і виконання всієї програми). Введемо предикат *InvCond* (предикат вищого порядку – над іншими предикатами-аргументами), який нам знадобиться далі:

InvCond(*Inv*, *PreCond*, *PostCond*) = $\forall S \in StartStates \bullet$ (*PreCond*(S) $\rightarrow$ *Inv*(S)) & $\forall S \in StopStates \bullet$ (*Inv*(S) $\rightarrow$ *PostCond*(S)) & $\forall (S, S') \in Step \bullet$ (*Inv*(S) $\rightarrow$ *Inv*(S')).

Цей предикат істинний, якщо на всіх початкових станах $S \in StartStates$ з *PreCond*(S) логічно випливає *Inv*(S), на всіх заключних станах $S \in StopStates$ з *Inv*(S) логічно випливає *PostCond*(S) та для кожної пари станів (S, S')$\in$ *Step* (кроку програми) маємо *Inv*(S) $\rightarrow$ *Inv*(S') (збереження істинності інваріанта).

Нагадаємо, що програма *P* при моделюванні перетворюється в транзиційну систему, яка складається з множини станів *States*, функції переходу *Step*, початкових станів *StartStates* та кінцевих станів *StopStates*, тобто програма однозначно визначає всі перелічені компоненти моделюючої транзиційної системи.

Рівнопотужність двох типів властивостей

Твердження. Властивість 1 типу виконується для деяких *PreCond*(S), *PostCond*(S) та програми *P* тоді і тільки тоді, коли виконується властивість 2 типу для цієї програми *P* і деякого інваріанта *Inv*(S).

Сформулюємо це твердження іншими словами.

Теорема 1. Нехай програма *P* (функція переходу *Step**) переводить початковий стан S ($S \in StartStates$) в кінцевий стан S' ($S' \in StopStates$). Тоді для будь-якої пари таких станів S та S' з *PreCond*(S) випливає *PostCond*(S') тоді і тільки тоді, коли існує інваріант *Inv*(s) ($s \in States$), такий, що є логічним наслідком *PreCond*(s) для кожного початкового стану ($\forall s \in StartStates \bullet$ $\bullet$ (*PreCond*(s) $\rightarrow$ *Inv*(s))), з нього випливає *PostCond*(s) в кожному заключному стані ($\forall s \in StopStates \bullet$ (*Inv*(s) $\rightarrow$ $\rightarrow$ *PostCond*(s))) і він зберігає істинність для кожного переходу з *Step* ($\forall (s, s') \in$ $\in Step \bullet$ (*Inv*(s) $\rightarrow$ *Inv*(s'))).

Формально це твердження можна записати таким виглядом:

$\forall$*Step*, *States*, *StartStates*, *StopStates* $\bullet$ $\forall$*PreCond* $\bullet$ $\forall$*PostCond* $\bullet$ ($\forall S \in StartStates \bullet$

$\forall S' \in StopStates \bullet$ ($PreCond(S)$ & $(S, S') \in$ $\in Step^* \rightarrow PostCond(S')$) $\Leftrightarrow$ $\exists Inv \bullet$ $\bullet InvCond(Inv, PreCond, PostCond)$),

де всі позначення (*Step*, *States*, *StartStates*, *StopStates*, *PreCond*, *PostCond*, *Inv*) використовуються та розуміються у вищевведеному смислі.

Доведення наведено в [1].

Тепер повернемось до методики доведення властивостей.

Формулювання методу. Роль інваріанта

Схема доведення властивостей така:

1) за програмою у мові *IPCL* (згідно наведеного алгоритму) будуємо функцію *Step* та розставляємо мітки;

2) фіксуємо множини початкових та кінцевих станів (згідно наведених визначень) – *StartStates* та *StopStates*;

3) формулюємо перед- та постумови та знаходимо за ними відповідний інваріант, або ж одразу формулюємо властивість типу інваріанта;

4) маючи $Inv(S)$, $S \in States$, робимо доведення-перевірку за наступними правилами.

Щоб довести властивість 2 типу, необхідно показати, що кожне перетворення, допустиме згідно функції *Step* в деякий поточний момент ("просування" виконання програми шляхом обчислення однієї чергової функції), зберігає істинність інваріанта $Inv(S)$. Тобто необхідно довести наступне:

$\forall S \in States \bullet \forall S' \in States$ (($Inv(S)$ & $(S, S') \in Step$) $\rightarrow Inv(S')$).

Тоді, якщо $\forall S \in StartStates \bullet Inv(S)$, що також слід перевірити, то за індукцією $Inv(S)$ має місце для всіх станів S, які є досяжними для програми P (згідно функції *Step*), зокрема, і $\forall S \in StopStates \bullet Inv(S)$. Властивість 2 типу, так, буде доведено.

Для доведення властивості 1 типу слід, згідно зі сформульованим і доведеним твердженням, знайти інваріант (предикат) $Inv(S)$, такий, що задовольняє умові $InvCond(Inv, PreCond, PostCond)$ для заданих *PreCond* та *PostCond*. Фактично, якщо такий інваріант буде знайдено, то доведення можна вважати завершеним. Так само ми доведемо істинність предиката-інваріанта на кожному кроці виконання програми, якщо до цього кроку можна дійти, стартувавши виконання з деякого початкового стану, для якого виконувалась умова $PreCond(S)$, а оскільки з істинності $Inv(S)$ випливає істинність $PostCond(S)$ на заключних станах ($\forall S \in StopStates \bullet (Inv(S) \rightarrow PostCond(S))$), то доведемо те, що було необхідно (властивість 1 типу).

Висновки щодо методу. Порівняння з іншими методами. Постаналіз

З усіх кроків креативним є лише третій, тобто знаходження (або формулювання) відповідного інваріанта. Один з можливих інваріантів сформульований при доведенні твердження в явному вигляді [1], але він надто складний для практичних доведень і має лише теоретичний зміст, адже він є, фактично, функціоналом вищого порядку. На практиці необхідно знаходити інваріант, який якомога точніше відображав би поведінку програми (для цього розроблений ряд підходів, зокрема, С.Л. Кривим [27] та ін., але в загальному випадку проблема залишається частково розв'язною). Інші кроки методу є механічними, тому можуть бути автоматизовані.

Зауважимо, що доведення за наведеним методом можна проводити для зазначеного вигляду (класу) програм ($P \in SeqILProgs$), де кількості паралельно виконуваних (під)програм є довільними натуральними (з 0) числами, фіксованими є тільки їх типи ($A, B, \ldots, C$), причому складність таких доведень буде лінійною (за кількістю кроків) щодо кількості операторів у (під)програмах (див. приклади, додаток Б), на відміну від багатьох інших методів. У такому випадку отримаємо свого роду символічні доведення над загальною формою програми, тобто реально викладений метод може оперувати "узагальненим виглядом програми". Він не може бути зведений до простої перевірки моделей (model checking), адже по-перше, всі стани не можна перелічити, бо не є фіксованою кількість (ступінь) кожної з паралельних підпрограм. По-друге, в кожному конкретному випадку, при великих

кількостях паралельних підпрограм, кількість станів буде також значною. Даний метод не є варіантом символьної перевірки моделей (symbolic model checking), який не використовує ні евристики, ні інші покращення, викладені в літературі. Також зрозуміло, що запропонований метод не є варіантом доведення теорем (theorem proving) в чистому вигляді, але останні є високоінтерактивними (мають високій ступінь взаємодії з людиною, тобто людського втручання [28]), адже в даному методі необхідно лише один раз спочатку сформулювати інваріант – решта (інші кроки методу), як вже зазначалось, може бути виконано автоматично. До того ж, доведення теорем посилається на логіку і є математичним підходом, водночас як даний метод спирається на текст програми, семантику та моделюючу транзиційну систему, є змішаним підходом. Щодо порівняння з методами перевірки моделей, то запропонований метод позбавлений "проблеми вибуху" кількості станів (але залежність складності доведення є лінійною за відношенням до кількості операторів у нотації програми).

В запропонованому методі сконцентровано ряд ідей. Як і в більшості методів, що базуються на станах, вводиться поняття стану та використовується операційна семантика для моделювання поведінки програми. Поняття стану тут інкорпорує в себе як стан пам'яті, так і стан керуючого пристрою (стан керування – позиції в контрольних точках програми). Моделювання програми здійснюється, як і в методиці ASM (Abstract State Machines) [25, 29] Гуревича, за допомогою деякої абстрактної машини над цими станами (машина – "перетворювач станів"), фактично – транзиційної системи, аналог яких – дискретні перетворювачі, можна знайти у роботах Глушкова. Поняття інваріанта, яке бере корені в роботах Флойда [30], Хоара [31], а згодом і Ешкрофта [3], для паралельних програм, та широко використовується в TLA (Лемпорт, нагадаємо, наголошує на виключній важливості інваріанта для доведень та суджень над паралельними програмами), відіграє в методі одну з ключових ролей. Також метод бере базові ідеї щодо коректності програми, сформульовані Хоаром у вигляді перед- та постумов. Доведення в методі виходить в загальному випадку схожим на доведення в методі TLA, хоча базується на простішій та ближчій до тексту програми моделі. В методі не передбачено введення додаткових допоміжних змінних (наприклад, в методах Овіцкі-Гріса), отже, він позбавлений недоліків, згадуваних у [32, 33] (як мінімум, складнощів з розумінням ролі та смислового навантаження таких змінних, а також з їх придумуванням), зате можна використовувати явно стан керування програми. Це дозволяє зв'язувати передумову та післяумову (взагалі, стани та судження про них) в єдиний інваріант для доведення явно, а не за допомогою "незрозумілих змінних" [33].

Разом з тим запропонований метод має відмінності від усіх перелічених. По-перше, він базується на композиційній семантиці, тобто базові функції мають бути задані в термінах композиційно-номінативної семантики, сама семантика мов *IPCL* є композиційною отже, метод має композиційно-номінативну природу. По-друге, метод має конкретні відмінності від інших. А саме, на відміну від TLA, метод суттєво прив'язаний до тексту програми на мові *IPCL*, яка схожа на звичайні мови імперативного програмування (на яких, власне, і пишеться переважна більшість реального програмного забезпечення). Переваги такого підходу обговорювались вище. Метод пропонує іншу, аніж в ASM, модель виконання, більш схожу на операційну семантику звичайних мов програмування (нагадаємо, в ASM моделлю є нескінченний цикл з паралельно виконуваних умовних операторів). Те ж саме зауваження відноситься і до мови Unity Ченді та Місри [12]. Мова, яка розглядається в роботі Ешкрофта, містить конструкцію fork, яка має іншу семантику, ніж композиція паралельного виконання в режимі чергування, введена тут. Щодо робіт Хоара, то в них розглядаються лише паралельні середовища з обміном повідомленнями (розподілені системи), розвиваються відповідні моделі та методи. Відмінність від методу Овіцкі-Гріса (та його композиційного або rely-

guarantee [6, 7] розширення Джонсом чи assumption-commitment [12] розширення Ченді та Місри) полягає в моделі, яка базується на аксіоматичній семантиці та її розширенні, водночас коли даний метод є ближчим до операційної семантики та моделювання (належить до методів, що базуються на станах). Зрештою, клас програм, розглядається в рамках наведеного методу, в явному вигляді не розглядається в жодному іншому підході, а в деяких (напевно, в більшості) взагалі не матиме адекватного представлення.

Висновки

Отже, запропонований метод це методологія доведення (критичних) властивостей програм в композиційних мовах *IPCL* [22]. Він дійсно нова комбінація відомих технік та методик щодо доведення коректності програмного забезпечення (особливо, паралельних систем, серверного програмного забезпечення для клієнт-серверних середовищ), побудований на фундаменті композиційно-номінативних методів для спеціального, але досить широкого класу програм. Відмінності методу від інших підходів детально проаналізовано вище. Сформульований метод має на меті побудувати "місток" між реальними програмами, написаними на звичайних імперативних мовах програмування, та математичними об'єктами (твердження, формули, властивості, доведення тощо). Відсутність якого можемо бачити в більшості відомих методів, які стартують одразу від математичних формул та систем (алгебр, транзиційних систем тощо).

Закладена в основу методу модель дозволяє оперувати програмами, запущеними в паралель довільну кількість разів (багато одночасно виконуваних екземплярів у режимі паралельного почергового виконання). Явно така модель не використовується в жодному з відомих підходів, всі підходи вимагають точної кількості програм та операторів, заданої наперед, особливо це стосується методів перевірки моделей (model checking). Метод базується на композиційно-номінативних системах, що дозволяє вільно обирати найбільш підходящий рівень для моделювання тієї чи іншої задачі та доведення критичних властивостей систем. Метод допускає автоматизацію більшості етапів доведення властивостей. Запропонований метод верифікації має лінійну складність доведення щодо кількості операторів програми замість експоненційної, як в більшості методів перевірки моделей, тобто він позбавлений "проблеми вибуху" кількості станів.

Викладений метод апробовано на реальних системах. Один з таких прикладів – доведення критичних властивостей системи виплати міжнародних грошових переказів Vigo Remittance Corp., реалізованої для ВАТ "Державний ощадний банк України".

1. *Панченко Т.В.* Композиційні методи специфікації та верифікації програмних систем. – Автореф. дис. канд. фіз.-мат. наук. – К., 2006. – 17 с.
2. *Wikipedia*, Internet-енциклопедія (www.wikipedia.org)
3. *Ashcroft E.A.* Proving assertions about parallel programs // J. of Computer and System Sciences. – 1975. – N 10. – P. 110–135.
4. *Hoare C.A.R.* Communicating Sequential Processes. – Prentice Hall International, 1985. – 238 p.
5. *Owicki S. and Gries D.* An Axiomatic Proof Technique for Parallel Programs // Acta Informatica. – 1976. – Vol. 6, N 4. – P. 319–340.
6. *Jones C.B.* Development Methods for Computer Programs Including a Notion of Interference: DPhil. Thesis. – Oxford University Computing Laboratory, 1981. – 315 p.
7. *Jones C.B.* Specification and Design of (Parallel) Programs // Information Processing Letters: IFIP Information Processing'83 (In IFIP 9th World Congress). – 1983. – P. 321–331.
8. *Xu Q., de Roever W.-P., He J.* The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs // Formal Aspects of Computing. – 1997. – Vol. 9, N 2. – P. 149–174.
9. *Harel D., Pnueli A.* On the development of reactive systems // Apt K.R. (ed.) Logics and models of concurrent systems, NATO ASI Series. – Springer-Verlag, 1985. – Vol. F13. – P. 477–498.
10. *Pnueli A.* The temporal logic of programs // Proc. 18th Annual Symposium on the Foundations of Computer Science (Providence). –

New York: IEEE Computer Society Press, 1977. – P. 46–57.
11. *Lamport L.* Verification and Specification of Concurrent Programs // deBakker J., deRoever W., Rozenberg G. (eds.) A Decade of Concurrency. – Berlin: Springer-Verlag, 1993. – Vol. 803. – P. 347–374.
12. *Chandy K.M., Misra J.* Parallel Program Design: A Foundation. – Reading, MA: Addison-Wesley Publishing Company, 1988. – 493 p.
13. *Lamport L.* The temporal logic of actions // ACM Transactions on Programming Languages and Systems. – 1994. – Vol. 16, N 3. – P. 872 – 923.
14. *Manna Z., Pnueli A.* The Temporal Logic of Reactive and Concurrent Systems, Specification. – Berlin: Springer-Verlag, 1992. – 427 p.
15. *Nikitchenko N.* A Composition Nominative Approach to Program Semantics. – Technical Report IT-TR: 1998-020. – Technical University of Denmark. – 1998. – 103 p.
16. *Редько В.Н.* Композиции программ и композиционное программирование // Программирование. – 1978. – № 5. – С. 3–24.
17. *Редько В.Н.* Основания композиционного программирования // Программирование. – 1979. – № 3. – С. 3–13.
18. *Басараб И.А., Никитченко Н.С., Редько В.Н.* Композиционные базы данных. – Киев: Либідь, 1992. – 191 с.
19. *Hoare C.A.R.* An Axiomatic Basis for Computer Programming // Communications of the ACM. – 1969. – Vol. 12, N 10. – P. 576–583.
20. *Heisel M., Reif W., Stephan W.* Implementing Verification Strategies in the KIV-System // Lusk E., Overbeek R. (eds.) Proc. 9th International Conference on Automated Deduction // Lecture Notes in Computer Science. – Berlin: Springer-Verlag, 1988. – Vol. 310 – P. 216–231.
21. *Reif W.* The KIV-approach to Software Verification // Broy M., Jähnichen S. (eds.) KORSO: Methods, Languages, and Tools for the Construction of Correct Software. Final Report // Lecture Notes in Computer Science. –Berlin: Springer-Verlag, 1995. – Vol. 1009. – P. 339–368.
22. *Панченко Т.В.* Методологія доведення властивостей програм в композиційних мовах IPCL // Тези доп. Міжнар. конф. “Теоретичні та прикладні аспекти побудови програмних систем” (TAAPSD’2004). – К.; 2004. – С. 62–67.
23. *Winskel G.* The Formal Semantics of Programming Languages: An Introduction. – London: MIT Press Foundations of Computing Series, 1993. – 361 p.
24. *Takaoka T.* A systematic approach to parallel program verification // Proc. Computing: Australasian Theory Symposium (CATS 96). – Melbourne (Australia), 1996. – P. 48–56.
25. *Reisig W.* The Expressive Power of Abstract-State Machines // Computing and Informatics. – 2003. – Vol. 22, N 3/4. – P. 209–219.
26. *Панченко Т.В.* Модель спрощеного стану для методу доведення властивостей в мовах IPCL та її застосування і переваги // Тези міжнар. наук. конф. TAAPSD`2007, Berdyansk, 4-9 September. – 2007. – P. 319–322.
27. *Кривий С.Л., Матвєєва Л.Е.* Формальні методи аналізу властивостей систем // Кибернетика и системный анализ. – 2003. – № 2. – С. 15–36.
28. *Edmund M. Clarke, Jeannette M.* Wing et al. Formal methods: state of the art and future directions // ACM Computing Surveys. – 1996. – Vol. 28, N 4. – P. 626–643.
29. *Gurevich Y.* Sequential Abstract-State Machines Capture Sequential Algorithms // ACM Transactions on Computational Logic. – 2000. – Vol. 1, N 1. – P. 77–111.
30. *Floyd R.W.* Assigning Meanings to Programs // Proc. Symposium on Applied Mathematics, Mathematical Aspects of Computer Science. – American Mathematical Society, 1967. – Vol. 19. – P. 19–32.
31. *Hoare C.A.R.* An Axiomatic Basis for Computer Programming // Communications of the ACM. – 1969. – Vol. 12, N 10. – P. 576–583.
32. *Dijkstra E.W.* A personal summary of the Gries-Owicki theory // Selected Writings on Computing: A Personal Perspective. – Springer-Verlag; NewYork; Heidelberg, Berlin; 1982. – P. 188–199.
33. *Lamport L.* Control predicates are better than dummy variables for reasoning about program control // ACM Transactions on Programming Languages and Systems (TOPLAS). – 1988. – Vol. 10, N 2. – P. 267 – 281.

Отримано 11.12.2007

Про автора:

Панченко Тарас Володимирович.

Місце роботи автора:

Київський національний університет
імені Тараса Шевченка,
01033, Київ, вул. Володимирська 60.
Тел.: 259 0519.
e-mail: tpanchenko@issystems.com.ua