УДК 004.415, 681.3

ЕКВІВАЛЕНТНІСТЬ ДВОХ СИСТЕМ ПАРАЛЕЛЬНОГО ВИКОНАННЯ

Т.В. Панченко, Sunmade Fabunmi

Досліджується метод доведення властивостей паралельних програм, що виконуються багатоекземплярно в режимі почергового покрокового переключення і взаємодіють через спільну пам'ять. У роботі розглянуто дві системи паралельного виконання програм та наведено обґрунтування взаємної виразності двох цих підходів. Один – з фіксованою, але параметричною, кількістю паралельно виконуваних програм. Другий – реалізує модель породження (start) і приєднання після зупинки (join) паралельних програм (також називається multithreading). Введено відповідні дві базові функції та задано їх семантику. Також наведено семантику інших функцій стосовно паралельного виконання, управління ресурсами та синхронізації доступу. Наведено теорему щодо (функціональної) еквівалентності двох систем та її обґрунтування. Програма в даному випадку розглядається як функція над даними. Стверджується, що для довільної програми в одній з систем паралелізму можна побудувати відповідну їй програму в іншій системі, яка повертає той самий результат (тобто є функціонально еквівалентною). Тільки продуктивні програми розглядаються тут у контексті взаємної виразності, оскільки у протилежному випадку вони "зависають" і не повертають жодного результату, отже є за межами нашого розгляду. Отриманий результат дозволяє звести роботу у більш складній (за будовою) системі з динамічним породженням екземплярів до більш простої (для доведень) системи з параметричною кількістю однакових програм, виконуваних у паралель. Визначено також питання для подальших досліджень у цьому напрямку.
Ключові слова: коректність програмного забезпечення, доведення часткової коректності, паралельна програма, interleaving, IPCL, композиційно-номінативні мови, формальна верифікація.

Исследуется метод доказательства свойств параллельных программ, которые выполняются многоэкземплярно в режиме поочередного пошагового переключения и взаимодействуют через общую память. В работе рассмотрены две системы параллельного выполнения программ и приведено обоснование взаимной выразимости этих двух подходов. Один – с фиксированным, но параметрическим, количеством параллельно выполняемых программ. Второй – реализует модель порождения (start) и присоединения после остановки (join) параллельных программ (также называется multithreading). Введено соответствующие две базовые функции и задано их семантику. Также задана семантика других функций касательно параллельного выполнения, управления ресурсами и синхронизации доступа. Приведены теорема об (функциональной) эквивалентности двух систем и ее обоснование. Программа в данном случае рассматривается как функция над данными. Утверждается, что для произвольной программы в одной из систем параллелизма можно построить соответствующую ей программу в другой системе, которая возвращает тот же результат (то есть функционально эквивалентна). Только продуктивные программы рассматриваются здесь в контексте взаимной выразимости, поскольку в противном случае они "зависают" и не возвращают никакого результата, поэтому лежат вне области нашего рассмотрения. Полученный результат позволяет свести работу в более сложной (по строению) системе с динамическим порождением экземпляров к более простой (для доказательств) системе с параметрическим количеством одинаковых программ, выполняемых параллельно. Также указаны вопросы для дальнейших исследований в этом направлении.
Ключевые слова: корректность программного обеспечения, доказательство частичной корректности, параллельная программа, interleaving, IPCL, композиционно-номинативные языки, формальная верификация.

The method for properties proof for parallel programs running multiple-instance interleaving with shared memory is investigated. Two systems for parallel execution of programs are considered and the justification of the mutual expressiveness of these two approaches are presented in this paper. The first one is with a fixed yet parametric number of programs executing in parallel. The second one implements a generation model (start) and joining after the stop (join) of parallel programs (also called multithreading). The corresponding two basic functions are provided, and their semantics are given. Also, the semantics of other functions related to parallel execution, resource management and access synchronization are presented in this paper. The theorem on the (functional) equivalence of these two systems and its justification are presented. The program in this case is considered as a function over the data. It is argued that for an arbitrary program in one of the systems of parallelism it is possible to construct the corresponding program in another system, which returns the same result (that is, functionally equivalent). Only productive programs are considered here in the context of mutual expressiveness, because otherwise they "hang" and do not return any result, thus they are out of our scope. The obtained result allows us to move reasoning from the more complex system (by structure) with a dynamic generation of parallel program instances to the simpler system (for proofs) with a parametric number of identical programs executed in parallel. Questions for further research in this direction are also identified.
Key words: software correctness, safety property proof, concurrent program, interleaving, IPCL, composition-nominative languages, formal verification.

Вступ. Постановка задачі

У роботах [1–3] та інших, де досліджується метод доведення властивостей програм у композиційних мовах *IPCL* з почерговим паралелізмом (shared memory interleaving concurrency, системи зі спільною пам'яттю та паралелізмом із почерговим переключенням), судження спираються на модель виконання $A^n \parallel B^m \parallel \ldots \parallel C^k$ [1], тобто є певна кількість однотипних підпрограм, запущених паралельно, із покроковим виконанням (операційна семантика) і попереміннною передачею управління між окремими підпрограмами. Вказані кількості підпрограм ($n, m, \ldots, k$) є параметрами моделі, яка розглядається, і всі судження проводяться для довільних їх значень одночасно, тобто – незалежно від кількості паралельно виконуваних екземплярів підпрограм. Крайнім випадком такого виконання підпрограм може бути їх строго послідовне виконання, одна за одною, але в загальному випадку матиме місце деяка траса комбінованого покрокового виконання.

Разом з тим, у реальних програмах часто застосовується техніка породження-завершення паралельних підпрограм. Тобто, вводиться деяка специфічна функція *start*(*P*), яка породжує екземпляр підпрограми *P* (додає +1 екземпляр до таких же, виконуваних паралельно, – зі стартом у точці входу, порожнім початковим даним (або переданими локальними даними в якості параметра ініціації, якщо передбачено механізмом породження) та доступом до спільного даного) і повертає код *ID* породженої підпрограми (*ID* процесу в операційних системах, для внутрішнього обліку та можливості подальших звернень до нього), а також функція *join*(*ID*), яка чекає на завершення підпрограми за кодом *ID* (раніше породженого екземпляру програми *P*), та після цього продовжує призупинене (власне) виконання.

Далі наведено обґрунтування взаємної виразності двох підходів до паралельного виконання програм, яке було сформульоване, зокрема, у [4]. Відповідно, буде розглянуто дві системи паралельного виконання програм. Одна – з фіксованою, але параметричною, кількістю паралельно виконуваних програм [1–3]. Друга – що реалізує модель породження (*start*) – приєднання після зупинки (*join*) паралельних програм (часто називається *multithreading*). У *IPCL* [1–3] введемо відповідні дві базові функції та задамо їх семантику. Наведемо теорему щодо еквівалентності двох систем.

Факт еквівалентності є важливим, оскільки дозволить звести роботу в більш складній за побудовою системі з динамічним породженням екземплярів паралельних підпрограм до більш простої для доведень системи з параметричною (але довільною, не фіксованою) кількістю однакових програм, виконуваних паралельно в режимі почергового переключення.

Семантика динамічного породження (*start*) та приєднання (*join*) паралельних програм

Уточнимо семантику цих двох нових операцій в транзиційній моделі виконання [1–3].

По-перше, дещо модифікуємо подання самої моделі відносно [1–3] для зручності, дана зміна не впливає на потужність моделі або виразну силу – лише додає можливість «масштабування» надалі.

Отже, стан будемо подавати у вигляді

$$S = \left(\left(\begin{pmatrix} m_{11} \\ m_{12} \\ \vdots \\ m_{1n_1} \end{pmatrix}, \begin{pmatrix} m_{21} \\ m_{22} \\ \vdots \\ m_{2n_2} \end{pmatrix}, \dots, \begin{pmatrix} m_{k1} \\ m_{k2} \\ \vdots \\ m_{kn_k} \end{pmatrix} \right), \left(d, \begin{pmatrix} d_{11} \\ d_{12} \\ \vdots \\ d_{1n_1} \end{pmatrix}, \begin{pmatrix} d_{21} \\ d_{22} \\ \vdots \\ d_{2n_2} \end{pmatrix}, \dots, \begin{pmatrix} d_{k1} \\ d_{k2} \\ \vdots \\ d_{kn_k} \end{pmatrix} \right) \right),$$

де

- m_{ij} – мітка, в якій перебуває (паралельна) підпрограма номер j, яка є екземпляром деякої P_i – для моделі виконання $P_1^{n_1} || \, P_2^{n_2} || \, \dots || P_k^{n_k}$, тобто $m_{ij} \in M_i$ – множина міток P_i [1];
- d_{ij} – відповідне локальне дане цієї підпрограми – P_{ij} ;
- d – глобальне (спільне, загальнодоступне всім підпрограмам) дане,

кожне дане має номінативну структуру – набір пар «ім'я – значення».

В цій моделі кожний крок виконання має вигляд: $(m_{ij}, d, d_{ij}) \to (m'_{ij}, d', d'_{ij})$, тобто програма P_{ij}, знаходячись у мітці m_{ij}, після виконання чергової інструкції перейде у мітку m'_{ij} та перетворить (можливо, змінить) глобальне дане d на d', а локальне d_{ij} – на d'_{ij}. Таким чином, в стані *S* за один крок може змінитись три компоненти.

Множину всіх станів можна задати для певної фіксованої кількості різновидів підпрограм *k* як

$$States = \dot{\bigcup_{n_1, n_2, \dots, n_k \in N}} \{ s \in \left((M_1^{n_1}, M_2^{n_2}, \dots, M_k^{n_k}), (D, D^{n_1}, D^{n_2}, \dots, D^{n_k}) \right) \}$$

тобто – як об'єднання множин станів для всіх можливих кількостей екземплярів $P_i, i \in \{1, \dots, k\}$, тобто всіх комбінацій натуральних степенів множин міток M_i та даних *D*.

Семантика базових функцій та композицій спадкується з [1] без змін, причому кожне перетворення залишає стан в тій же компоненті об'єднання (стосовно набору $n_1, n_2, \dots, n_k \in N$).

В модифікованій моделі зручно задавати семантику нових двох операцій. Так, розмічена операція $[m_b]\ id := start(P_x)\ [m_a]$, виконана у підпрограмі P_{ij}, перетворює стан S у наступний ($x, i \in \{1, \dots, k\}, j \in \{1, \dots, n_i\}$):

$$S' = \left(\left(\begin{pmatrix} m'_{11} \\ m'_{12} \\ \vdots \\ m'_{1n_1} \end{pmatrix}, \dots, \begin{pmatrix} m'_{x1} \\ m'_{x2} \\ \vdots \\ m'_{xn_x+1} \end{pmatrix}, \dots, \begin{pmatrix} m'_{k1} \\ m'_{k2} \\ \vdots \\ m'_{kn_k} \end{pmatrix} \right), \left(d', \begin{pmatrix} d'_{11} \\ d'_{12} \\ \vdots \\ d'_{1n_1} \end{pmatrix}, \dots, \begin{pmatrix} d'_{x1} \\ d'_{x2} \\ \vdots \\ d'_{xn_x+1} \end{pmatrix}, \dots, \begin{pmatrix} d'_{k1} \\ d'_{k2} \\ \vdots \\ d'_{kn_k} \end{pmatrix} \right) \right),$$

де

- $m'_{ij} = m_a$, якщо $m_{ij} = m_b$,
- $d'_{ij} = d_{ij}\nabla[id \mapsto Id]$ і $d' = d$, якщо id – ім'я в локальному даному d_{ij} або ж $d'_{ij} = d_{ij}$ і $d' = d\nabla[id \mapsto Id]$, якщо id – ім'я в глобальному даному d, Id – значення функції $start(P_x)$, тобто код щойно породженого нового процесу P_{xn_x+1};
- $m'_{xn_x+1} = m_{x[start]}$ – має значення початкової мітки підпрограми P_x;
- $d'_{xn_x+1} = \emptyset$ (початкове дане нового екземпляру P_x – порожнє);
- $m'_{yz} = m_{yz}, d'_{yz} = d_{yz}$ для решти компонент стану,

при цьому змінюється розмірність двох векторів у складі стану (стан переходить до іншої компоненти з об'єднання *States*) – додається по одній компоненті стану (m'_{xn_x+1}) та даного (d'_{xn_x+1}) для нового, щойно створеного та запущеного на виконання, екземпляру P_x.

Семантика розміченої операції приєднання $[m_b]\, join(Id)\, [m_a]$, виконаної у підпрограмі P_{ij}, де Id – ідентифікатор екземпляру номер q підпрограми P_x ($q \in \{1, \dots, n_x\}$) – P_{xq}, перетворює стан S у наступний ($x, i \in \{1, \dots, k\}, j \in \{1, \dots, n_i\}$):

$$S' = \left(\left(\begin{pmatrix} m'_{11} \\ m'_{12} \\ \vdots \\ m'_{1n_1} \end{pmatrix}, \dots, \begin{pmatrix} m'_{x1} \\ m'_{x2} \\ \vdots \\ m'_{xn_x} \end{pmatrix}, \dots, \begin{pmatrix} m'_{k1} \\ m'_{k2} \\ \vdots \\ m'_{kn_k} \end{pmatrix}\right), \left(d', \begin{pmatrix} d'_{11} \\ d'_{12} \\ \vdots \\ d'_{1n_1} \end{pmatrix}, \dots, \begin{pmatrix} d'_{x1} \\ d'_{x2} \\ \vdots \\ d'_{xn_x} \end{pmatrix}, \dots, \begin{pmatrix} d'_{k1} \\ d'_{k2} \\ \vdots \\ d'_{kn_k} \end{pmatrix}\right)\right),$$

де

- $m'_{ij} = m_a$, якщо $m_{ij} = m_b$ та $m_{xq} = m_{x[final]}$ – фінальна мітка підпрограми P_x (тобто програма P_{xq} завершилась),
- $m'_{yz} = m_{yz}, d'_{yz} = d_{yz}$ для всіх решти компонент стану,

інакше – стан не змінюється (якщо наведена щойно умова «P_{xq} завершилась» не виконується), $S' = S$ – тобто підпрограма P_{ij} не зрушує з місця і буде, фактично, чекати, поки P_{xq} завершиться.

Семантика функцій fork, lock, unlock та інших операцій

Також специфікуємо семантику інших важливих та практичних функцій для реального використання *multithreading* паралелізму в транзиційній моделі виконання програм *IPCL*.

Операція *fork*(), яка створює точну копію поточного процесу (включно з поточним станом пам'яті) та запускає його на виконання з поточної інструкції, широко застосовується у Unix-подібних операційних системах для породження паралельних процесів, зокрема у сервісних системах. Функція *fork*(), як і *start*(), породжує новий екземпляр паралельно виконуваної підпрограми, проте запускає її з поточного місця на поточних даних екземпляра-породжувача. Так, розмічена операція $[m_b]\, id := fork()\, [m_a]$, виконана у підпрограмі P_{ij}, перетворює стан S у наступний ($i \in \{1, \dots, k\}, j \in \{1, \dots, n_i\}$):

$$S' = \left(\left(\begin{pmatrix} m'_{11} \\ m'_{12} \\ \vdots \\ m'_{1n_1} \end{pmatrix}, \dots, \begin{pmatrix} m'_{i1} \\ m'_{i2} \\ \vdots \\ m'_{in_i+1} \end{pmatrix}, \dots, \begin{pmatrix} m'_{k1} \\ m'_{k2} \\ \vdots \\ m'_{kn_k} \end{pmatrix}\right), \left(d', \begin{pmatrix} d'_{11} \\ d'_{12} \\ \vdots \\ d'_{1n_1} \end{pmatrix}, \dots, \begin{pmatrix} d'_{i1} \\ d'_{i2} \\ \vdots \\ d'_{in_i+1} \end{pmatrix}, \dots, \begin{pmatrix} d'_{k1} \\ d'_{k2} \\ \vdots \\ d'_{kn_k} \end{pmatrix}\right)\right),$$

де

- $m'_{ij} = m_a$, якщо $m_{ij} = m_b$;
- $d'_{ij} = d_{ij}\nabla[id \mapsto Id]$ і $d' = d$, якщо id – ім'я в локальному даному d_{ij} або ж $d'_{ij} = d_{ij}$ і $d' = d\nabla[id \mapsto Id]$, якщо id – ім'я в глобальному даному d, Id – значення функції $start(P_x)$, тобто код щойно породженого нового процесу P_{in_i+1};
- $m'_{in_i+1} = m_a$ – має значення мітки підпрограми P_i після щойно виконаної функції *fork*() (як і у P_{ij});
- $d'_{in_i+1} = d_{ij}\nabla[id \mapsto 0]$ і $d' = d$, якщо id – ім'я в локальному даному d_{ij} або ж $d'_{in_i+1} = d_{ij}$ і $d' = d\nabla[id \mapsto 0]$, якщо id – ім'я в глобальному даному d;
- $m'_{yz} = m_{yz}, d'_{yz} = d_{yz}$ для решти компонент стану,

при цьому змінюється розмірність двох векторів у складі стану (стан переходить до іншої компоненти з об'єднання *States*) – додається по одній компоненті стану (m'_{in_i+1}) та даного (d'_{in_i+1}) для нового, щойно створеного та запущеного на виконання, екземпляру P_i.

Функції *lock*() та *unlock*() дозволяють виконати (захищені) оператор(и) в ексклюзивному режимі (в режимі ексклюзивного доступу), коли лише одна з програм – паралельних екземплярів-претендентів може їх виконувати. Такі операції або їх аналоги є основою для реалізації синхронізації, критичних секцій, моніторів та взаємовиключного доступу (взаємних блокувань) у операційних системах при управління доступом до ресурсів.

Розмічена функція $[m_b]\ lock(res);\ [m_a]$, виконана у підпрограмі P_{ij}, перетворює стан S у наступний ($i \in \{1, \dots, k\}, j \in \{1, \dots, n_i\}$):

$$S' = \left(\left(\begin{pmatrix} m'_{11} \\ m'_{12} \\ \vdots \\ m'_{1n_1} \end{pmatrix}, \dots, \begin{pmatrix} m'_{i1} \\ m'_{i2} \\ \vdots \\ m'_{in_i} \end{pmatrix}, \dots, \begin{pmatrix} m'_{k1} \\ m'_{k2} \\ \vdots \\ m'_{kn_k} \end{pmatrix} \right), \left(d', \begin{pmatrix} d'_{11} \\ d'_{12} \\ \vdots \\ d'_{1n_1} \end{pmatrix}, \dots, \begin{pmatrix} d'_{i1} \\ d'_{i2} \\ \vdots \\ d'_{in_i} \end{pmatrix}, \dots, \begin{pmatrix} d'_{k1} \\ d'_{k2} \\ \vdots \\ d'_{kn_k} \end{pmatrix} \right) \right),$$

де

- $m'_{ij} = m_a$, якщо $m_{ij} = m_b$ та $res \Longrightarrow (d) = 0$ (тобто ресурс, асоційований з *res*, був вільний – значення змінної з іменем *res* у глобальному даному дорівнює 0);
- $d' = d\nabla[res \longmapsto 1]$, тобто ресурс, асоційований з *res*, тепер «зайнятий», причому важливо гарантувати, що *res* – іменує «системну змінну», тобто ніхто інший не може змінювати її значення (ніякі інші функції та оператори, окрім функцій *lock* та *unlock*);
- $m'_{yz} = m_{yz}, d'_{yz} = d_{yz}$ для решти компонент стану,

інакше (якщо $res \Longrightarrow (d) \neq 0$, тобто ресурс, асоційований з *res*, не був вільний) – стан не змінюється, *S'* = *S*, тобто підпрограма P_{ij} не зрушує з місця і буде, фактично, чекати, поки відповідний ресурс звільниться ($res \Longrightarrow (d)$ стане рівним 0).

Функцією *lock*() зазвичай відкривається робота з критичною секцією (критичним ресурсом). У кінці критичної секції має іти виклик функції *unlock*(), що вказує на її завершення та звільнення критичного ресурсу.

Так, розмічена функція $[m_b]\ unlock(res);\ [m_a]$, виконана у підпрограмі P_{ij}, перетворює стан S у наступний ($i \in \{1, \dots, k\}, j \in \{1, \dots, n_i\}$):

$$S' = \left(\left(\begin{pmatrix} m'_{11} \\ m'_{12} \\ \vdots \\ m'_{1n_1} \end{pmatrix}, \dots, \begin{pmatrix} m'_{i1} \\ m'_{i2} \\ \vdots \\ m'_{in_i} \end{pmatrix}, \dots, \begin{pmatrix} m'_{k1} \\ m'_{k2} \\ \vdots \\ m'_{kn_k} \end{pmatrix} \right), \left(d', \begin{pmatrix} d'_{11} \\ d'_{12} \\ \vdots \\ d'_{1n_1} \end{pmatrix}, \dots, \begin{pmatrix} d'_{i1} \\ d'_{i2} \\ \vdots \\ d'_{in_i} \end{pmatrix}, \dots, \begin{pmatrix} d'_{k1} \\ d'_{k2} \\ \vdots \\ d'_{kn_k} \end{pmatrix} \right) \right),$$

де

- $m'_{ij} = m_a$, якщо $m_{ij} = m_b$;
- $d' = d\nabla[res \longmapsto 0]$, тобто ресурс, асоційований з *res*, тепер «вільний», причому важливо гарантувати, що *res* – іменує «системну змінну», тобто ніхто інший не може змінювати її значення (ніякі інші функції та оператори, окрім функцій *lock* та *unlock*);
- $m'_{yz} = m_{yz}, d'_{yz} = d_{yz}$ для решти компонент стану.

Аналогічно можна задати семантику вбудованих атомарних операцій процесорів *test_and_set*(*a, b*), *swap*(*a, b*) роботу з семафорами (зокрема, функції Дейкстри *P*(*semaphore*) і *V*(*semaphore*), або *wait*(*semaphore*) і *signal*(*semaphore*) відповідно) та інших, що є основою атомарного виконання і механізмів синхронізації та управління доступом в операційних системах – зокрема, *enter_critical_section*(*section_id*) та *exit_critical_section*(*section_id*), монітори тощо.

Еквівалентність двох систем

Тепер – про взаємну звідність. Ми розглядаємо лише продуктивні програми, тобто такі, що завершуються і повертають дане, яке містить результат їх роботи (обчислень). Тобто, розглядаємо паралельні програми з функціональної точки зору. Програми, що «зациклюються», тобто не завершуються, не являють собою суттєвого інтересу не тільки в контексті даної роботи, але і в цілому, у більш широкому контексті, адже, по суті, вони є помилковими (бо не повертають ніякого результату над вхідними даними).

Теорема 1. Дві семантичні моделі виконання – *IPCL* $P_1^{n_1}||\ P_2^{n_2}||\ ...\ ||P_k^{n_k}$ (з фіксованими степенями паралельних програм, див. [1]) та модель з динамічним породженням паралельних підпрограм *start* / *join* (специфікована вище) – еквівалентні за обчислювальною потужністю. Тобто, ці дві сукупності програм в *IPCL* (розширеному *start* / *join*) – взаємно-виразні для довільної «продуктивної» програми (тієї, що зупиняється).

Обґрунтування. Виразність моделі з фіксованими степенями ($P_1^{n_1}||\ P_2^{n_2}||\ ...\ ||P_k^{n_k}$) у моделі зі *start* / *join* – очевидна – на початку, в «основному потоці», треба породити потрібну кількість екземплярів кожної з підпрограм $P_1 \dots P_k$ за допомогою *start*, і одразу ж «приєднати» їх всі, у довільній послідовності, за допомогою *join*. В зворотному напрямку – покажемо від супротивного. Припустимо, є деяка програма, яка задана у розширеному *start* / *join IPCL*, яка не має еквівалента (не може бути подана) у моделі з фіксованими степенями виконання ($P_1^{n_1}||\ P_2^{n_2}||\ ...\ ||P_k^{n_k}$). Тоді можливі два варіанти. Якщо програма у розширеній моделі не зупиняється («зациклюється», наприклад, нескінченно породжує нові процеси за допомогою *start*) – то вона не є продуктивною, отже не підпадає під теорему і не цікавить нас взагалі (саме з причини непродуктивності). Якщо ж програма зупиняється, то нехай вона це робить за *N* кроків. Тоді цю поведінку можна виразити «напряму» у моделі $P_1^N||\ P_2^N||\ ...\ ||P_k^N$, але така програма у *IPCL* буде мати і багато інших трас виконання. Аби написати програму, яка точно відтворюватиме поведінку, потрібно ввести в глобальне дане змінні (зі значеннями: 0 «не стартувала», 1 «виконується» та 2 «завершилась») за кількістю реально породжених програм (точніше – модель двовимірного масиву *run* у номінативних даних з індексами номеру програми {1,…,*k*} та номеру екземпляру {1,…,n_i}). Тоді:

1) *start*(.) – збільшує останнє збережене n_i на 1 та присвоює *run*[.][n_i] := 1 (тобто дає сигнал «запуску» відповідному екземпляру паралельної підпрограми);

2) *join*(.) – переходить до наступної інструкції тільки якщо відповідна підпрограма з кодом *Id* (значення аргументу) – завершилась, тобто елемент масиву *run*, що відповідає її стану, дорівнює 2;

3) самі паралельні підпрограми (всі, окрім «основного потоку», тобто стартової, «точки входу») модифікуються таким чином, що на старті (перед кодом самої підпрограми) очікують *run*[.][.] = 1 (відповідна до свого номеру екземпляру підпрограми комірка масиву *run*) у нескінченому циклі

while *run*[.][.] = 0 ***do*** *skip*;

а при завершенні – встановлюють у ту ж саму комірку *run*[.][.] = 2 (ознака завершення роботи), на старті ж усі *run*[.][.] = 0 (тобто, відповідна підпрограма ще не була запущена).

Ці зміни приводять до точного відтворення роботи (поведінки, можливих трас) програми у розширеній (*start* / *join*) мові *IPCL*. Так, зокрема, ведеться облік (у т.ч. – кількості) запущених паралельних програм (щоб не було накладок з нумерацією – n_i), вони стартують лише у «дозволені» моменти (*run*[.][.] = 1) та враховано специфіку *join*.

Зауваження. Звісно, розглянуті два класи не є еквівалентними в повному розумінні і не для кожної програми можна побудувати аналог в іншій системі. Ми розглядаємо лише програми зі скінченими трасами (як вже було зазначено вище), які завершують своє виконання та повертають деякий результат (результуюче дане може бути отримане з кінцевого стану ланцюжку переходів – виконання – у транзиційній системі). Так, для програми

P: *id* := *start*(***P***);

немає аналогу в системі з фіксованими степенями паралельно виконуваних підпрограм, але ж наведена програма, при цьому, не є продуктивною – її виконання теоретично ніколи не завершується (адже вона постійно та нескінченно породжує нові екземпляри самої себе), отже вона не повертає дане – результат свого виконання. Такі програми нас не цікавлять з очевидних причин – їх непродуктивності.

Висновки

В роботі введено модель паралелізму в *IPCL* з динамічним породженням екземплярів (паралельних) підпрограм. Для цього визначено семантику нових функцій *start* та *join* (породження та приєднання / очікування завершення / фіналізації) у транзиційній системі з [1–3]. Сформульовано теорему про еквівалентність (рівнопотужність) двох підходів до побудови програм і, відповідно, двох семантик – розширеної та базової ([1–3]) транзиційної. Показано, що взаємна виразність можлива у випадках програм, що завершуються.

Саме обґрунтування не є доведенням у строгому математичному смислі, тож подальші роботи можуть рафінувати наведені міркування до формального доведення.

У цілому, це – важливий результат, адже дозволяє звести роботу у більш складній (за будовою) системі з динамічним породженням екземплярів до більш простої (для доведень) системи з параметричною кількістю однакових програм, виконуваних у паралель.

Також треба дослідити, як зміняться доведення [4-7] для моделі виконання з породженням екземплярів програм у *IPCL*.

Література

1. Панченко Т.В. Метод доведення властивостей програм в композиційно-номінативних мовах IPCL. *Проблеми програмування*. 2008. № 1. С. 3–16.
2. Панченко Т.В. Методологія доведення властивостей програм в композиційних мовах IPCL. Доповіді Міжнародної конференції "*Теоретичні та прикладні аспекти побудови програмних систем*" (TAAPSD'2004). К., 2004. С. 62–67.
3. Панченко Т.В. Композиційні методи специфікації та верифікації програмних систем. Автореферат дисертації на здобуття наук. … канд. фіз.-мат. наук. К. 2006. 17 с.
4. Іванов Є., Панченко Т.В., Fabunmi Sunmade. Модель паралелізму в IPCL з породженням екземплярів. Праці Міжнар. конф. "*Теоретичні та прикладні аспекти побудови програмних систем*" (TAAPSD'2017). Київ. 2017. С. 131–133.
5. Polishchuk N.V., Kartavov M.O. and Panchenko T.V. Safety Property Proof using Correctness Proof Methodology in IPCL. Proceedings of the 5th International Scientific Conference "*Theoretical and Applied Aspects of Cybernetics*". Kyiv: Bukrek, 2015. P. 37–44.
6. Kartavov M., Panchenko T., Polishchuk N. Properties Proof Method in IPCL Application To Real-World System Correctness Proof. International Journal "*Information Models and Analyses*". Sofia, Bulgaria: ITHEA. 2015. Vol. 4. N 2. P. 142–155.
7. Panchenko T.V. Application of the Method for Concurrent Programs Properties Proof to Real-World Industrial Software Systems. Proceedings of the 12th International Conference on ICT in Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer (ICTERI 2016), edited by Vadim Ermolayev et al. Kyiv. 2016. P. 119–128 (CEUR-WS. – Vol. 1614. – ISSN:1613-0073, available online: http://ceur-ws.org/Vol-1614/).

References

1. PANCHENKO, T. (2008) The Method for Program Properties Proof in Compositional Nominative Languages IPCL [in Ukrainian]. Problems of Programming. 1. P. 3–16.
2. PANCHENKO, T. (2004) The Methodology for Program Properties Proof in Compositional Languages IPCL [in Ukrainian]. In Proceedings of the International Conference "Theoretical and Applied Aspects of Program Systems Development" (TAAPSD'2004). Kyiv. P. 62–67.
3. PANCHENKO, T. (2006) Compositional Methods for Software Systems Specification and Verification (PhD Thesis Synopsis) [in Ukrainian]. Kyiv. 17 p.
4. IVANOV, IE., PANCHENKO, T. and FABUNMI, S. (2017) Parallelism Model with Instances Generation in IPCL [in Ukrainian]. In Proceedings of the International Conference "Theoretical and Applied Aspects of Program Systems Development" (TAAPSD'2017). Kyiv. P. 131–133.
5. POLISHCHUK, N.V., KARTAVOV, M.O. and PANCHENKO, T.V. (2015) Safety Property Proof using Correctness Proof Methodology in IPCL. In Proceedings of the 5th International Scientific Conference "Theoretical and Applied Aspects of Cybernetics", Kyiv, Bukrek, P. 37–44.
6. KARTAVOV, M., PANCHENKO, T. and POLISHCHUK, N. (2015) Properties Proof Method in IPCL Application To Real-World System Correctness Proof. International Journal "Information Models and Analyses", Sofia, Bulgaria, ITHEA, Vol. 4, N 2, P. 142–155.
7. PANCHENKO, T. (2016) Application of the Method for Concurrent Programs Properties Proof to Real-World Industrial Software Systems. In Proceedings of the 12th International Conference on ICT in Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer (ICTERI 2016), edited by Vadim Ermolayev et al., Kyiv, P.119–128 (CEUR-WS. – Vol. 1614, available online: http://ceur-ws.org/Vol-1614/).

Про авторів:

Тарас Володимирович Панченко,
кандидат фізико-математичних наук, доцент,
доцент кафедри теорії та технології програмування
факультету кібернетики Київського національного університету імені Тараса Шевченка.
Кількість наукових публікацій в українських виданнях – 35.
Кількість наукових публікацій в зарубіжних виданнях – 3.
http://orcid.org/0000-0003-0412-1945,

Sunmade Fabunmi,
аспірант кафедри теорії та технології програмування
факультету комп'ютерних наук та кібернетики
Київського національного університету імені Тараса Шевченка.
Кількість наукових публікацій в українських виданнях – 3.
http://orcid.org/0000-0002-3926-106X.

Місце роботи авторів:

Київський національний університет імені Тараса Шевченка,
03680, Київ, проспект Академіка Глушкова, 4Д.
Тел.: 259 0519.
E-mail: taras.panchenko@gmail.com