УДК 004.415.52, 681.3

PETERSON'S ALGORITHM TOTAL CORRECTNESS PROOF IN IPCL

A.A. Zhygallo

The total correctness of the Peterson's Algorithm has been proved. States and transitions were fixed by the program. Runtime environment considered is interleaving concurrency with shared memory. Invariant of the program was constructed. All reasoning provided in terms of Method for software properties proof in Interleaving Parallel Compositional Languages (IPCL). Conclusions about adequacy of the Method usage for such a kind of tasks (thanks to flexibility of composition-nominative platform) and its practicality as well as ease of use for real-world systems have been made based on this and other author's works.

Key Words: Peterson's algorithm, mutual exclusion, software total correctness, formal verification, liveness property, concurrent program, interleaving, IPCL, composition-nominative languages.

Доведено тотальну коректність алгоритму Пітерсона. За програмою зафіксовано стани та переходи транзиційної системи. Середовище виконання – паралельне з почерговим переключенням зі спільною пам'яттю. Сформульовано інваріант. Судження проведено в рамках методу доведення властивостей програм в Interleaving Parallel Compositional Languages (IPCL). Спираючись на дану та інші роботи автора зроблено висновки щодо адекватності застосування методу для подібних задач завдяки гнучкості композиційно-номінативної платформи та його практичності і легкості застосування для реальних систем.

Ключові слова: алгоритм Пітерсона, взаємне виключення, тотальна коректність програм, формальна верифікація, liveness property, паралельна програма, interleaving, IPCL, композиційно-номінативні мови.

Доказана тотальная корректность алгоритма Петерсона. За программой зафиксированы состояния и переходы транзиционной системы. Среда выполнения – параллельная с поочередным переключением с общей памятью. Сформулировано инвариант. Суждение происходит в рамках метода доказательства свойств программ в Interleaving Parallel Compositional Languages (IPCL). Руководствуясь этой и другими работами автора сделано вывод об адекватности использования метода для подобных задач исходя из гибкости композиционно-номинативной платформы и его практичности и легкости в использовании для реальных систем.

Ключевые слова: алгоритм Петерсона, взаимное исключение, тотальная корректность программ, формальная верификация, liveness property, параллельная программа, interleaving, IPCL, композиционно-номинативные языки.

Introduction

Almost every day while operating on up-to-date computers or working with modern software products we have to deal with systems which are based on shared memory concurrency [1]. Those are supercomputers with UMA and NUMA memory architecture, SMP-based computer hardware architectures, operating systems, database management systems (DBMS), centralized databases, server-side software in client-server environments, etc. At the same time, due to the power wall in silicon technology all state of the art computing devices have multiple processing units and the transition to chip multiprocessors is happening very fast. Therefore, these days parallel programming becomes a necessity while the problem of software verification is still acute and open to debate.

Although there is a broad range of approaches to handle this issue most of them have remarkable disadvantages in terms of using them on practice as they are too complicated or too theorized. Moreover, some of them are not applicable for coping with real tasks in general. For instance, without specifying the details, original Owicki-Gries method [2] requires the quadratic number of verifications relating to the program operators amount. While the extended version of the rely-guarantee Owicki-Gries method [3] needs the implementation of additional variables as well as non-evident formulating of rely- and guarantee- conditions in order to tackle this task. In TLA [4] (Temporal Logic of Actions) Lamport offers to construct a model which is not much easier than the two previous ones. Moreover, TLA is characterized with a big difference between the program and its proving formula. In such a way Interleaving Parallel Composition Language (IPCL) [5] might be one of the most efficient solutions. While IPCL is up to solve the verification problem of the parallel software it describes the so-called serializability mechanism. Though ultimately, we will work with sequential processes which steps will be interrupted by parallel running programs in an unpredictable way.

Total Correctness Proof in IPCL. Problem Statement

In this particular work the total correctness of Peterson's algorithm will be proven with a help of Method for program properties proof in IPCL. Paterson's algorithm is a concurrent programming [6] algorithm for mutual exclusion (i.e. no two concurrent processes are in their critical section at the same time [7]) that allows two processes to share a single-use resource without conflict, using only shared memory for communication.

The partial correctness (the safety) was proved in the other author's paper [8]. The aim of this work is to prove the total correctness (the liveness) of the mentioned algorithm. That is to show that the algorithm will stop for sure.

We will do the reasoning based on IPCL [5] – the class of compositional [9–11] languages, which provide an adequate model for interleaving concurrency with shared memory [12]. We will also use the Method for software properties proof in IPCL (including safety and correctness ones) [12].

IPCL Syntax

The syntax of such languages is defined as follows:

$P ::= P1;P2\ |$

*if b then P*1 *else P*2 |
while b do P |
x:=*e*|
*P*1 || *P*2,

where *x*:=*e* is an atomic, vector assigning and *P*1 || *P*2 implies interleaving parallelism.

IPCL Semantics

Let us define compositional semantics of IPCL. The semantics of the first three compositions is standard while semantics of interleaving parallelism is defined through its arguments semantics and syntactic context in traditional way with its algebraic and semantic properties. An Index-parameter of the semantics function is a syntactic context - piece of the program or the program itself. So:

Semantics of "**;**":

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

where ∇' – component-wise superposition of Cartesian product of nominative data:

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

$Pr_i(d)$ is a projection of the Cartesian product d by the i-th component, $d_1, d_2 \in D{\times}D$,

This small modification is due to the fact that data is considered as two-component one – such that contains "global" and "local" parts and ∇ – is a simple operation of the data superposition (two nominal or nominative sets) in accordance with [9–11] here.

Semantics of "**:=**":

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

where $f \in Oper$ – is a (semantic) fucntion, which complies with syntactic operator $x := e.$ An assignment (the result of f) provides two-component result, as global and local data are evaluated, and superposition is occurring to the relevant components of d data: global variables – to the first component, local variables – to the second; x and e are the vectors of names and values (expressions) respectively,

semantics of "**if**":

$$sem_{\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q}(d) = sem_P(d), \text{ iff } b(d) = \text{True},$$
$$sem_{\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q}(d) = sem_Q(d), \text{ iff } b(d) = \text{False},$$

semantics of "**while**":

$$sem_{\mathbf{while}\ b\ \mathbf{do}\ P}(d) = sem_{P\,;\,\mathbf{while}\ b\ \mathbf{do}\ P}(d), \text{ iff } b(d) = \text{True},$$
$$sem_{\mathbf{while}\ b\ \mathbf{do}\ P}(d) = d, \text{ iff } b(d) = \text{False},$$

semantics of "||":

|| is associative and commutative:

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

on the syntactic level it means respectively:

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

|| relatively to "**if**":

$$sem_{\mathbf{\mathit{if}}\ b\ \mathbf{\mathit{then}}\ P\ \mathbf{\mathit{else}}\ Q\,||\,R}(d) = sem_{P\,||\,R}(d),\ iff\ b(d) = True,$$
$$sem_{\mathbf{\mathit{if}}\ b\ \mathbf{\mathit{then}}\ P\ \mathbf{\mathit{else}}\ Q\,||\,R}(d) = sem_{Q\,||\,R}(d),\ iff\ b(d) = False,$$

$$sem_{\mathbf{\mathit{if}}\ b\ \mathbf{\mathit{then}}\ P\ \mathbf{\mathit{else}}\ Q\,;\,P'\,||\,R}(d) = sem_{P\,;\,P'\,||\,R}(d),$$
$$iff\ b(d) = True,$$
$$sem_{\mathbf{\mathit{if}}\ b\ \mathbf{\mathit{then}}\ P\ \mathbf{\mathit{else}}\ Q\,;\,P'\,||\,R}(d) = sem_{Q\,;\,P'\,||\,R}(d),$$
$$iff\ b(d) = False,$$

|| relatively to "**while**":

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

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

|| relatively to "**;**":

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

where $A, B, C, P, P', Q, R \in Terms$ – programs in IPCL (terms in IPCL Algebra), b is a syntax notation of an appropriate condition of the predicate *pred* from *Pred* that is $b\ (d) = sem_{b}\ (d) = pred\ (d)$ and $d \in D \times D$.

In all the above definitions if the value of $b\ (d)$ is undefined then the value of the left side of the relevant equality will also be undefined. Similarly, if any of the definitions of the right side of equality is undefined then the value of the left side is undefined.

F is a set of functions for the data conversion, C is the compositions over functions from F. $F{=}Oper{\cup}Pred$, where $Oper{=}D{\times}D \rightarrow D{\times}D$ and $Pred{=}D{\times}D \rightarrow$ {True, False}, where the first occurrence of D to the Cartesian product is "global data" while the second is "local data" for the current process; functions which return values from the set {True, False} (predicates) do not change the current state (i.e. does not have "side effect") of data $D{\times}D$ – they are used as conditions in branching and cycle operators; $D{=}ND(V,W)$ in the usual sense (as simple nominative data [9–11]).

The specific IPCL class language is formed by fixing F.

The functions from F are atomic transactions which are indivisible in the sense of parallelism – their execution cannot be interrupted. Thus, the conditions in compositions are also atomic.

Peterson's algorithm notation in IPCL

The main idea of the Peterson's algorithm is to use three variables, *flag*1, *flag*2 and *turn*. *flag*1 or *flag*2 value of 1 indicates that the process *n* wants to enter the critical section. Entrance to the critical section is granted for process *T*1 if *T*2 does not want to enter its critical section or if *T*2 has given priority to *T*1 by setting *turn* to 1. Also entrance to the critical section is granted for process *T*2 if *T*1 does not want to enter its critical section or if *T*1 has given priority to *T*2 by setting *turn* to 2.

According to the method for program's properties correctness proof [1, 5, 12] the sources of *T*1 and *T*2 programs with labels will have the form:

```
T1≡ [M11]  flag1:=1;

    [M12]  turn:= 2;

    while [M13]  (flag2 = 1 && turn = 2)  do
    skip;

    [M14]  r:=do_critical_operations();  //critical section

    [M15]  flag1:= 0;[M16]

T2≡ [M21]  flag2:=1;

    [M22]  turn:= 1;

    while [M23]  (flag1 = 1 && turn = 1)  do
    skip;

    [M24 ]  r:=do_critical_operations();  //critical section

    [M25]  flag2:= 0;[M26]
```

In this work while-cycle condition (for instance in *T1* program: □□□□2 = 1 && □□□□ = 1) is considered as an atomic operation while we may as well use the method dividing this process into three independent stages, like:

□□□$_{□□□□2}$ *= □□□□2;*

```
□□□$_{□□□□}$= □□□□;
□h ???( ? ? ?$_{? ?? ?2}$ = 1 && ? ? ?$_{?? ??}$ = 1) do
begin
        ? ? ?$_{? ?? ?2}$ = ? ?? ?2;
        ? ? ?$_{?? ??}$ = ?? ??;
end;
```

This case could be researched as a separate question in further works.

The whole program system will have the following structure:

? ??? ? ??? = ? ?|| ? ?.

States and Transitions

The state of this program will be as follows:

? ?? ??= (?1, ?2,[? ?? ?1 ↦ ?$_1$, ? ?? ?2 ↦ ?$_2$, ?? ?? ↦ □],□1, □2),

where □1 ∈ {? ??, ? ??, ? ??, ? ??, ? ??, ? ??} – labels of *T1*; ?2 ∈ {□ □□, □ □□, □ □□, □ □□, □ □□, □ □□} – labels of *T2*; [□□□□1 ↦ □$_1$, □□□□2 ↦ □$_2$, □□□□ ↦ □] – global data; *d*1 and *d*2 are the local data of *T*1 and *T*2 respectively. *States* will denote the set of all possible states.

The transition system will have the following scheme of transitions:

□□□□□□□□□□□□= {□$_1$ → □$_2$ | □$_1$, □$_2$∈ ? ?? ??? ∧
(? ?$_{11}$(?$_1$, ?$_2$) ∨ ?$_{12}$(?$_1$, ?$_2$) ∨ ?$_{13}$(?$_1$, ?$_2$) ∨
? ?$_{14}$(?$_1$, ?$_2$) ∨ ?$_{15}$(?$_1$, ?$_2$) ∨ ?$_{16}$(?$_1$, ?$_2$) ∨
? ?$_{21}$(?$_1$, ?$_2$) ∨ ?$_{22}$(?$_1$, ?$_2$) ∨ ? ?$_{23}$(?$_1$, ?$_2$) ∨
? ?$_{24}$(?$_1$, ?$_2$) ∨ ?$_{25}$(?$_1$, ?$_2$) ∨ □□$_{26}$(□$_1$, □$_2$)},

where each predicate □□$_j$ □= 1:2, □= 1:6 describes the possible program step between states.

For program *T*1 we have 6 different steps:

1. Move **M11** →**M12** (assigning variable *flag*1 to 1):

$Tr_{11}(S_1,S_2) = (Pr_1(S_1) = \mathbf{M11}) \wedge (Pr_1(S_2) = \mathbf{M12}) \wedge (Pr_2(S_1) = Pr_2(S_2)) \wedge (Pr_3(S_1) = d) \wedge (Pr_3(S_2) = d \nabla [flag1 \mapsto 1]) \wedge (Pr_4(S_1) = Pr_4(S_2)) \wedge (Pr_5(S_1) = Pr_5(S_2))$

2. Move **M12** → **M13** (assigning variable *turn* to 1):

$Tr_{12}(S_1,S_2) = (Pr_1(S_1) = \mathbf{M12}) \wedge (Pr_1(S_2) = \mathbf{M13}) \wedge (Pr_2(S_1) = Pr_2(S_2)) \wedge (Pr_3(S_1) = d) \wedge (Pr_3(S_2) = d \nabla [turn \mapsto 2]) \wedge (Pr_4(S_1) = Pr_4(S_2)) \wedge (Pr_5(S_1) = Pr_5(S_2))$

3. Move **M13** → **M13** (*true* value of while-cycle condition):

$Tr_{13}(S_1,S_2) = (Pr_1(S_1) = \mathbf{M13}) \wedge (Pr_1(S_2) = \mathbf{M13}) \wedge (Pr_2(S_1) = Pr_2(S_2)) \wedge (Pr_3(S_1) = Pr_3(S_2) = [flag1 \mapsto f_1, flag2 \mapsto f_2, turn \mapsto t]) \wedge (Pr_4(S_1) = Pr_4(S_2)) \wedge (Pr_5(S_1) = Pr_5(S_2)) \wedge (f_2 = 1) \wedge (t = 2)$

4. Move **M13** → **M14** (*false* value of while-cycle condition):

$Tr_{14}(S_1,S_2) = (Pr_1(S_1) = \mathbf{M13}) \wedge (Pr_1(S_2) = \mathbf{M14}) \wedge (Pr_2(S_1) = Pr_2(S_2)) \wedge (Pr_3(S_1) = Pr_3(S_2) = [flag1 \mapsto f_1, flag2 \mapsto f_2, turn \mapsto t]) \wedge (Pr_4(S_1) = Pr_4(S_2)) \wedge (Pr_5(S_1) = Pr_5(S_2)) \wedge (f_2 \neq 1 \vee t \neq 2)$

5. Move **M14** → **M15** (execution of the critical section as ‘atomic’ operation):

$Tr_{15}(S_1,S_2) = (Pr_1(S_1) = \mathbf{M14}) \wedge (Pr_1(S_2) = \mathbf{M15}) \wedge (Pr_2(S_1) = Pr_2(S_2)) \wedge (Pr_3(S_1) = Pr_3(S_2)) \wedge (Pr_4(S_1) = d_1) \wedge (Pr_4(S_2) = func_1(d_1)) \wedge (Pr_5(S_1) = Pr_5(S_2))$

6. Move **M15** → **M16** (assigning variable *flag*1 to 0):

$Tr_{16}(S_1,S_2) = (Pr_1(S_1) = \mathbf{M15}) \wedge (Pr_1(S_2) = \mathbf{M16}) \wedge (Pr_2(S_1) = Pr_2(S_2)) \wedge (Pr_3(S_1) = d) \wedge (Pr_3(S_2) = d \nabla [flag1 \mapsto 0]) \wedge (Pr_4(S_1) = Pr_4(S_2)) \wedge (Pr_5(S_1) = Pr_5(S_2))$

Similar transitions could be fixed for program *T*2.

The set of the fixed Starting states will have the following structure:

$StartStates = \{S \in States \mid$
$Pr_1(S) = \mathbf{M11} \wedge Pr_2(S) = \mathbf{M21}\}$

The Final states are:

$FinalStates = \{S \in States \mid$
$Pr_1(S) = \mathbf{M16} \wedge Pr_2(S) = \mathbf{M26}\}$

Invariant

In these terms the mutual exclusion condition for the algorithm, that no two concurrent processes are in their critical section at the same time, can be formulated as:

$\neg\, (Pr_1(S) \in \{\mathbf{M14}, \mathbf{M15}\} \wedge Pr_2(S) \in \{\mathbf{M24}, \mathbf{M25}\})$

The invariant of the ***program*** system is:

$Inv(S) = I_1(S) \wedge I_2(S) \wedge I_3(S) \wedge I_4(S)$,

where

$I_1(S) = Pr_1(S) \in \{\mathbf{M12}, \mathbf{M13}, \mathbf{M14}, \mathbf{M15}\} \rightarrow flag1 \Rightarrow (Pr_3(S)) = 1$,

$I_2(S) = Pr_1(S) \in \{\mathbf{M14}, \mathbf{M15}\} \rightarrow$
$(Pr_2(S) \notin \{\mathbf{M24}, \mathbf{M25}\} \wedge$
$Pr_2(S) = \mathbf{M23} \rightarrow turn \Rightarrow (Pr_3(S)) = 1)$,

$I_3(S) = Pr_2(S) \in \{\mathbf{M22}, \mathbf{M23}, \mathbf{M24}, \mathbf{M25}\} \rightarrow flag2 \Rightarrow (Pr_3(S)) = 1$,

$I_4(S) = Pr_2(S) \in \{\mathbf{M24}, \mathbf{M25}\} \rightarrow$
$(Pr_1(S) \notin \{\mathbf{M14}, \mathbf{M15}\} \wedge$
$Pr_1(S) = \mathbf{M13} \rightarrow turn \Rightarrow (Pr_3(S)) = 2)$

The idea of invariant $Inv(S)$ is clear and follows from the definition of critical section. If $T1$ is located previously to while-cycle {**M12**, **M13**} or is inside its critical section {**M14**, **M15**}, then $flag1 = 1$, namely it wants to enter critical section (getting access to the resource). If $T1$ "goes through" while-cycle {**M14**, **M15**}, that means it comes out of the cycle (entrance to critical section and getting access to the resource), then the other process ($T2$) is outside critical section (i.e. not in {**M24**, **M25**}), and also if $T2$ is located previously to critical section {**M23**} – then it does not have an access, because it is $T1$'s turn to entrance now: $turn = 1$.

The proof that invariant holds true over all transitions is rather technical and will not be presented here [8].

It is obvious, that $Inv(S)$ implies that

$Pr_1(S) \in \{\mathbf{M14}, \mathbf{M15}\} \rightarrow Pr_2(S) \notin \{\mathbf{M24}, \mathbf{M25}\} \wedge Pr_2(S) \in \{\mathbf{M24}, \mathbf{M25}\} \rightarrow Pr_1(S) \notin \{\mathbf{M14}, \mathbf{M15}\}$

which is equal to:

$\neg\, (Pr_1(S) \in \{\mathbf{M14}, \mathbf{M15}\} \wedge Pr_2(S) \in \{\mathbf{M24}, \mathbf{M25}\})$

namely, the mutual exclusion condition: for any state S it is not possible to appear in critical section (marks **M14**, **M15** for $T1$ and marks **M24**, **M25** for $T2$) for both processes ($T1$ and $T2$) at the same time.

Total Correctness. Proof

Let us prove that that the program system ? ? ? ? ? ? ? = ? ?||? ? will stop in the shown model.

According to the algorithm structure the only problem space is a cycle. We have to prove that as soon as the process $T2$ has finished its work (*flag2 := 0*), the process $T1$ will leave the cycle (we will not observe the infinite cycle).

Proof by contradiction: let us suppose that the process $T1$ after the label **M13** will get into the label **M13** which means the true value of the while-cycle condition. According to the transition $Tr_{13}(S_1,S_2)$ $flag2 = 1$. From the fact that none of transitions of *T1* will influence the global variable *flag2* and from the condition that the process *T2* has finished and thus due to $Tr_{16}(S_1,S_2)$, we have that $flag2 = 0$. Got contradiction.

The same proof is applicable to show that if the process *T1* has ended then we will not have the infinite cycle in the process *T2.*

Let us consider the situation where both processes *T1* and *T2* have infinite cycles at the same time. This situation is only possible when we observe the transitions in the following order: $Tr_{13}(S_1,S_2)$ -> $Tr_{23}(S_1,S_2)$ -> $Tr_{13}(S_1,S_2)$ ->

$Tr_{23}(S_1,S_2)$ and so on. Due to $Tr_{13}(S_1,S_2)$ global variable $turn = 2$ while due to $Tr_{23}(S_1,S_2)$ $turn = 1 \neq 2$. That means that the situation where both processes have infinite cycles is impossible.

We have proved that the infinite cycle is impossible and that both processes will end their work.

Conclusion

Together with the other results (besides this work) total correctness of well-known Peterson's Algorithm was proved. Namely, we have:

1) noted the algorithm in IPCL terms,

2) provided semantics in terms of states and transitions,

3) formulated invariant of the program,

4) proved the liveness correctness (impossibility of the infinite cycles).

According to the wide range of difficulties of total correctness proofs in parallel environments, we may state that the IPCL method is well adapted to parallel software verification. Moreover, this method let us make the proof shorter by choosing an adequate level of abstraction of the problem due to the universality of composition-nominative languages.

1. *Панченко Т.В.* Композиційні методи специфікації та верифікації програмних систем. Дисертація на здобуття наукового ступеня кандидата фізико-математичних наук.: 01.05.03 / Т.В. Панченко – Київ, 2006. – 177 с.
2. *Owicki S.* An Axiomatic Proof Technique for Parallel Programs / S. Owicki, D. Gries // Acta Informatica. – 1976. – Vol. 6, N 4. – P. 319–340.
3. *Xu Q., W.-P. de Roever, J. He.* The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs // Formal Aspects of Computing. – 1997. – Vol. 9, N 2. – P. 149–174.
4. *Lamport L.* Verification and Specification of Concurrent Programs // deBakker J., deRoever W., Rozenberg G. (eds.) A Decade of Concurrency, Vol. 803. – Berlin: Springer-Verlag, 1993. – P. 347–374.
5. *Панченко Т.В.* Методологія доведення властивостей програм в композиційних мовах IPCL // Доповіді Міжнарод-ної конференції "Теоретичні та прикладні аспекти побудови програмних систем" (TAAPSD'2004). – К., 2004. – С. 62–67.
6. *Tanenbaum A.S.* Modern operating systems, 3Ed. – Upper Saddle River, NJ: Pearson Prentice Hall, 2008. – 1104 p.
7. *Peterson G.L.* Myths About the Mutual Exclusion Problem // Information Processing Letters. – 1981. – 12 (3). – P. 115–116.
8. *Жигалло А.А., Остаповська Ю.А., Панченко Т.В.* Доведення у IPCL коректності алгоритму Пітерсона для взаємного виключення // Вісник Київського університету імені Тараса Шевченка. Серія: фізико-математичні науки. – 2015. – Вип. 4.
9. *Редько В.Н.* Композиции программ и композиционное программирование // Программирование. – 1978. – № 5. – С. 3–24.
10. *Редько В.Н.* Основания композиционного программирования // Программирование. – 1979. – № 3. – С. 3–13.
11. *Nikitchenko N.* A Composition Nominative Approach to Program Semantics. – Technical Report IT-TR: 1998-020. – Technical University of Denmark, 1998. – 103 p.
12. *Панченко Т.В.* Метод доведення властивостей програм в композиційно-номінативних мовах IPCL // Проблеми програмування. – 2008. – № 1. – С. 3–16.

References

1. PANCHENKO, T. (2006) Compositional Methods for Software Systems Specification and Verification (PhD Thesis), Kyiv, 177 p.
2. OWICKI S., GRIES D. (1976) An Axiomatic Proof Technique for Parallel Programs. Acta Informatica, Vol. 6, № 4, pp. 319–340.
3. XU Q., de ROEVER W.-P., HE J. (1997) The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. Formal Aspects of Computing, Vol. 9, № 2, pp. 149–174.
4. LAMPORT L. (1993) Verification and Specification of Concurrent Programs. A Decade of Concurrency, deBakker J., deRoever W., Rozenberg G. (eds.), Berlin: Springer-Verlag, Vol. 803, pp. 347–374.
5. PANCHENKO, T. (2004) The Methodology for Program Properties Proof in Compositional Languages IPCL. In Proceedings of the International Conference "Theoretical and Applied Aspects of Program Systems Development" (TAAPSD'2004), Kyiv, pp. 62–67.
6. TANENBAUM, A.S. (2008) Modern operating systems, 3Ed. Upper Saddle River, NJ: Pearson Prentice Hall, 1104 p.
7. PETERSON, G.L. (1981) Myths About the Mutual Exclusion Problem. Information Processing Letters, 12(3), pp. 115–116.
8. ZHYGALLO A., OSTAPOVSKA Yu., PANCHENKO T. (2015) Peterson's Algorithm for Mutual Exclusion Correctness Proof in IPCL. Bulletin of Taras Shevchenko National University of Kyiv. Series: Physical & Mathematical Sciences, N 4.
9. REDKO, V. (1978) Compositions of Programs and Composition Programming. Programming, 5, pp. 3–24.
10. REDKO, V. (1979) Foundation of Composition Programming. Programming, 3, pp. 3–13.
11. NIKITCHENKO, N. (1998) A Composition Nominative Approach to Program Semantics. Technical Report IT-TR: 1998-020. Technical University of Denmark. 103 p.
12. PANCHENKO, T. (2008) The Method for Program Properties Proof in Compositional Nominative Languages IPCL. Problems of Programming. 1. pp. 3–16.

About the author:

Zhygallo Andrey,
undergraduate student, Department of Applied Statistics, Faculty of Cybernetics.
1 Ukrainian publication.
http://orcid.org/0000-0002-2976-3250.

Affiliation:

Taras Shevchenko National University of Kyiv, 4D Academician Glushkov avenue, Kyiv, Ukraine, 03680.
Tel.: +38(063)8795790/
Email: zhigallandrejj@gmail.com