УДК 681.3, 004.05

Панченко Т. В., к.ф.-м.н., доц.

Метод доведення коректності паралельних програм з використанням спрощеного стану та його застосування

Київський національний університет імені Тараса Шевченка, Україна, 01601, місто Київ, вул. Володимирська, 64/13
e-mail: pantaras@ukr.net

T. V. Panchenko, PhD, docent.

Parallel Programs Correctness Proof Method Using Simplified State with Applications

Taras Shevchenko National University of Kyiv, 64/13, Volodymyrska Street, City of Kyiv, Ukraine, 01601
e-mail: pantaras@ukr.net

Доведення коректності паралельних програм є складною задачею, оскільки потоки виконання можуть впливати один на інший через спільні змінні або механізм повідомлень. Класичні методи типу Флойда-Хоара не застосовні у цьому випадку напряму і необхідні нові нетривіальні підходи, які враховують це ускладнення. Розроблений метод для доведення коректності програм (для визначеного спеціального класу програм – а саме серверного програмного забезпечення в архітектурі SMP як-то сервер баз даних або веб-сервер) – є предметом даної статті. Його простота у випадку відсутності локальних даних підпрограм і застосування до класичної задачі-прикладу паралельного додавання до спільної змінної подано у роботі.

Ключові слова: коректність програм, паралельні програми, композиційно-номінативні методи.

The correctness proof for programs with parallelism is complicated problem due to possible inference of one execution path on another via shared data or messaging mechanics. Classical methods like Floyd-Hoare cannot be applied in this situation directly and new non-trivial methods are required to cope with such complexity. Developed method for software correctness proof (for defined custom class of programs – namely server software for SMP architecture like DB-server or Web-server) is the subject of this article. Its simplicity in case of local subroutine data absence and application to classical sample task of shared variable concurrent increment are shown here.

Key Words: software correctness, concurrent programs, composition-nominative methods.

Статтю представив д.ф.-м.н., проф. Буй Д.Б.

Introduction

The methodology for software correctness proof was developed in [1] and presented there in details. Using this methodology one can formulate and prove properties of programs developed in Interleaving Parallel Composition Language (IPCL) [1]. The syntax and operational semantics of IPCL presented ibid. Here we concentrate on simplified method for program properties (including correctness) proof for some subclass of all IPCL programs. The notion of simplified state for this kind of reasoning (introduced in [1]) as well as clarification on specific IPCL subclass mentioned above are also discussed here.

IPCL Subclass and Method Description

Following [2], the composition languages class [3] IPCL and methodology proposed by the author for programs properties (including correctness) proof in composition languages IPCL [2] are well suited for the specification and verification of the server-side software in client-server environments.

But it should be noted that in the absence of local data in programs *A*, *B*, …, *C* (which are subroutines of program *P*), it is advisable to consider the concept of a **Simplified State** (here we use the notation and follow the terms according to [2], in particular, the program $P = A^n \parallel B^m \parallel \ldots \parallel C^k \in SeqILProgs$, where

SeqILProgs is the sequential programs subclass of all IPCL programs). Simplified state is an aggregated state of the following form $N^{Pmq}\times D$, where $Pmq = ||A_{marks}||+||B_{marks}||+\ldots+||C_{marks}||$, i.e. the sum of all tags amounts (A_{marks}, B_{marks}, …, C_{marks}) for programs *A*, *B*, …, *C* (which are subroutines of *P*), where A_{marks} is the set of labels for labeled program *A* – i.e. when we labeled every particular atomic operator (or operation, function call etc.) in the text notation of program *A*, $||A|| = card(A)$ is the power of the set *A*, and *D* contains the global (shared, common) data for all routines in the form of *nominative data* [3], which means a set of pairs *name* $\mapsto$ *value*.

First $||A_{marks}||$ components of such a state include the number of programs that are executing now operators at appropriate labels of program *A* in this state, and the sum of these components in each state for the program *P* is equal to *n* – i.e. the number of instances (copies) of *A* program in *P* (in terms of program *P*). The following $||B_{marks}||$ components contain the number of programs that are executing now operators at appropriate labels of *B* program in this state, and the sum of these components in each state for the program *P* is equal to *m* (the number of instances of the program *B* in *P*) in terms of *P* programs and so on. The last component *D* contains global shared data for all routines of *P*.

Simplified state operates with the number of routines which are executing now at appropriate label instead of label identification for each instance of every individual routine (*A*, *B*, ..., *C*). In fact, routines are not distinguished from each other accurate to the label of current execution if they do not change local data (actually – do not have them at all), but only operate with shared global data.

One can obtain a simplified state for the state $S \in A_{marks}^{\,n}\times B_{marks}^{\,m}\times\ldots\times C_{marks}^{\,k}\times D\times D^{numprocs}$, $numprocs=n+m+\ldots+k$, in the following way. As sets A_{marks}, B_{marks}, …, C_{marks} are finite, let $A_{marks}=\{A_1, \ldots, A_a\}$, …, $C_{marks}=\{C_1, \ldots, C_c\}$. Let $Pr_i(S)$ be the i-th component of a tuple *S*. Then for any regular state *S* the corresponding simplified state will be $SS=(a_1, \ldots, a_a, \ldots, c_1, \ldots, c_c, d)$, where $a_j=||\{\ i \mid Pr_i(S)=A_j\ ,\ i\in N_{Pmq}\ \}||= P_{S[A_j]}$, $\forall j\in N_a$, …, $c_j=||\{\ i \mid Pr_i(S)=C_j\ ,\quad i\in N_{Pmq}\ \}||= P_{S[C_j]}$, $\forall j\in N_c$, $d=Pr_{numprocs+1}(S)$.

The set of simplified states *SStates* introduced accordingly. The same goes with simplified initial states *SStartStates* (they all have the structure (n, 0, ..., 0, ..., k, 0, ..., 0, d), which means all the routines are at their appropriate start labels), simplified final states *SStopStates* (their structure is (0 ..., 0, n, ..., 0, ..., 0, k, d), which means all the routines have stopped execution at their exit ("after-program") label) and the transition function (one step execution of *P* program) over the simplified states *SStep*: *SStates* $\rightarrow$ *SStates*. For every particular possible transition of execution control for each routine this new transition function *SStep* decreases the value of some components of the *SStates* vector by 1 (eg. first component) and at the same time increases the value of some other component by 1 (eg. second component), which means the transfer of execution control from one label (in this case A_1) to another label (in this case A_2) for one of the routines (in this case *A*), and also changes the value of the last components (global data) if the current function is in subclass *Oper* of class *F*. All these objects could be easily obtained by transferring states into their simplified states appropriately or by direct construction.

Sample Correctness Task: Parallel Increment To Shared Variable. Discussion

To prove the properties of both types (defined in [1,2]) one could apply the methodology given in [1,2]. To demonstrate the simplicity of the method [1,2] and the convenience of simplified state model usage, let us discuss a sample that – de facto – (see. [4], [5] and others) become "standard" to check the "efficiency" of parallel programs and methods concerning modeling, execution, model checking and correctness proof. Let the program twice (separately, independently) increments some shared (global) variable concurrently. In the IPCL notation program *P* will look like $x:=x+1 \parallel x:=x+1$, where *x* is a shared (global) variable and the set $F=\{f_1\}$, where $f_1(d) \equiv d \nabla [\ x \mapsto (x \Rightarrow (d)+1)\ ]$ (here the semantic function f_1 has syntactic form of $x:=x+1$). It is clear that every action is atomic (read, add +1, write back the variable value). The problem is formulated as follows: to prove that when the initial value of shared (global) variable *x* is 0, the final value of *x* (after the program stops) will be equal to 2.

Without going into details, "pure" Owicki-Gries method [4] requires program properties to be formulated and tested for each operator taking into account their potential interference and "cross"-interference effects. Recall that inspections quantity is quadratic with respect to the program operator count.

The extended version of the rely-guarantee Owicki-Gries method modification [5] requires two additional variables (for such a trivial program!)

introduction and formulating of nontrivial (!) rely- and guarantee- conditions for application of the method to this task. The proof itself takes more than one page space [5].

TLA [6] offers (for the very similar task) to build a model that is not much easier than the first two, and the formulation of the model and, in fact, the proof itself (with explanations) is again at least the page of formulas in space.

Sample Correctness Task: Parallel Increment To Shared Variable. Proof

Let us now consider a *detailed* proof of *generalization* of this property in IPCL. Explore generalized version of the task first. Thus, let us have a program $P = Inc^n$ (for some fixed $n \in N$), where $Inc = x:=x+1$. The semantic model described above with $F=\{f_1\}$. We consider a simplified model, as we have no local variables for routines (just shared global x). Labelling algorithm [1] (obviously) will result to the next for *Inc*: *Inc* = **[M1]** $x:=x+1$ **[M2]**, $SStates=\{(s_1, s_2, d) \mid s_1 \in N,\ s_2 \in N,\ d \in D\}$, where s_1 is the number of programs (all performed in parallel) executing currently at the label **[M1]**, s_2 is the number of programs executing at the label **[M2]** (in fact, finished currently), and d is the shared (global) data (containing variable x and its value).

It is clear that $SStep=\{((s_1, s_2, d), (s_1-1, s_2+1, f_1(d))) \mid s_1>0 \ \&\ s_1 \in N \ \&\ s_2 \in N\}$, $SStartStates=\{(n, 0, [\, x \mapsto 0\,])\}$, $SStopStates=\{\, s \mid s=(0, n, d') \ \&\ \exists s_1, s_2, \ldots, s_l \bullet (s_1 \in SStartStates \ \&\ s_l=s \ \&\ (\forall i \in N_{l-1} \bullet (s_i, s_{i+1}) \in SStep)\,)\,\}$ – by definition.

Let $PreCond(SS) \equiv (x \Rightarrow (d)=0)$ and $PostCond(SS) \equiv (x \Rightarrow (d)=n)$, where $SS=(s_1, s_2, d) \in SStates$ – (simplified) state. Consider the *invariant* $Inv(SS) \equiv (s_2=x \Rightarrow (d))$. Let us verify $InvCond(Inv, PreCond, PostCond)$: $\forall S \in SStartStates \bullet (PreCond(S) \to Inv(S)) = PreCond((n, 0, [\, x \mapsto 0\,])) \to Inv((n, 0, [\, x \mapsto 0\,])) = \text{True} \to \text{True} = \text{True}$, $\forall S \in SStopStates \bullet (Inv(S) \to PostCond(S)) = \forall S=(s_1, s_2, d)=(0, n, d') \in SStopStates \bullet ((s_2=x \Rightarrow (d)) \to (x \Rightarrow (d)=n))$, since $s_2=n$, then the implication is True, and the entire predicate is True, $\forall (S, S') \in SStep \bullet (Inv(S) \to Inv(S')) = \forall (S, S')=((s_1, s_2, d), (s_1-1, s_2+1, f_1(d))) \in SStep \bullet ((s_2=x \Rightarrow (d)) \to (s_2+1=x \Rightarrow (f_1(d)))) = \forall (S, S') \in SStep \bullet$ True, since it is obvious that $x \Rightarrow (f_1(d)) = x \Rightarrow (d)+1$.

Hence, *Inv* indeed is invariant for the program *P*, besides at the starting states it is a logical consequence of the pre-condition and it implies post-condition at the final states. Q.E.D.

Sample Correctness Task: Parallel Increment To Shared Variable. Conclusions

Thus, we have proved a *generalized* property. Originally formulated (correctness) property can be derived from a more general just letting $n=2$.

Note that some of sources mentioned here are considered a variant of the example $P' = x:=x+1 \parallel x:=x+2$ with pre-condition $x=0$. To solve this problem with the method proposed one should use another (but obvious) invariant $Inv(SS) \equiv (s_2+2*s_4 = x \Rightarrow (d))$, where $SS=(s_1, s_2, s_3, s_4, d) \in SStates$ is simplified state, for generalized program $P'' = (x:=x+1)^n \parallel (x:=x+2)^m$ apart with labels (labelled program *P*): **[M1]** $x:=x+1$ **[M2]**, **[M3]** $x:=x+2$ **[M4]**, and initial states $SStartStates=\{(n, 0, m, 0, [\, x \mapsto 0\,])\}$ (with $n=m=1$).

Note also that the number of inspections to prove is linear regarding the number of program operators in both examples (in general, for every program). Details on this and other examples as well as detailed theory can be found in [1].

Conclusions

The simplified method with simplified state introduced for program properties proof (including program correctness checking) is appropriate for parallel programs without local variables which use only shared ones. This case is appropriate model of server software (in SMP architecture) like Web-servers and Database-server applications and such a kind of computing. The method and the model as well as Interleaving Parallel Compositional Languages class (IPCL) are described in details in [1] and were first presented in [2] and then in [7] with modifications (simplified state).

All reasoning presented here are in terms of composition-nominative approach [8,3], but the main formalism for concurrent behavior modelling is Abstract State Machines [9,10], which in line with compositional semantics gives us enough means to describe interleaving parallel execution within operational approach indeed.

Список використаних джерел

1. *Панченко Т.В.* Композиційні методи специфікації та верифікації програмних систем. Дисертація на здобуття наукового ступеня кандидата фізико-математичних наук.: 01.05.03 / Панченко Тарас Володимирович – К., 2006. – 177 с.
2. *Панченко Т.В.* Методологія доведення властивостей програм в композиційних мовах IPCL / Тарас Володимирович Панченко // Доповіді Міжнародної конференції “Теоретичні та прикладні аспекти побудови програмних систем” (TAAPSD’2004). – К., 2004. – С. 62–67.
3. *Nikitchenko N.* A Composition Nominative Approach to Program Semantics / N. Nikitchenko. – Technical Report IT-TR: 1998-020. – Technical University of Denmark, 1998. – 103 p.
4. *Owicki S.* An Axiomatic Proof Technique for Parallel Programs / S. Owicki, D. Gries // Acta Informatica. – 1976. – Vol. 6, № 4. – P. 319–340.
5. *Xu Q.* The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs / Q. Xu, W.-P. de Roever, J. He // Formal Aspects of Computing. – 1997. – Vol. 9, № 2. – P. 149–174.
6. *Lamport L.* Verification and Specification of Concurrent Programs / L. Lamport // deBakker J., deRoever W., Rozenberg G. (eds.) A Decade of Concurrency, Vol. 803. – Berlin: Springer-Verlag, 1993. – P. 347–374.
7. *Панченко Т.В.* Модель спрощеного стану для методу доведення властивостей в мовах IPCL та її застосування і переваги / Тарас Володимирович Панченко // Доповіді міжнародної наукової конференції TAAPSD`2007. – Berdyansk, 2007. – С. 319-322.
8. *Редько В.Н.* Композиции программ и композиционное программирование / Владимир Никифорович Редько // Программирование. – 1978. – № 5. – С. 3–24.
9. *Reisig W.* The Expressive Power of Abstract-State Machines / W. Reisig // Computing and Informatics. – 2003. – Vol. 22, № 3–4. – P. 209–219.
10. *Gurevich Y.* Reconsidering Turing's thesis (toward more realistic semantics of programs) / Y. Gurevich. – Technical report CRL-TR-36-84. – University of Michigan, 1984. – 13 p.

References

1. PANCHENKO, T. (2006) *Compositional Methods for Software Systems Specification and Verification* (PhD Thesis). Kyiv. 177 p.
2. 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 Develpment"* (*TAAPSD'2004*). Kyiv. pp. 62–67.
3. NIKITCHENKO, N. (1998) *A Composition Nominative Approach to Program Semantics*. Technical Report IT-TR: 1998-020. Technical University of Denmark. 103 p.
4. OWICKI, S., GRIES, D. (1976) An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica*. Vol. 6, No 4. pp. 319–340.
5. XU, Q., DE ROEVER, W.-P. and HE, J. (1997) The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects of Computing*. Vol. 9, No 2. pp. 149–174.
6. LAMPORT, L. (1993) Verification and Specification of Concurrent Programs. deBakker, J., deRoever, W. and Rozenberg, G. (eds.) *A Decade of Concurrency*. Vol. 803. Berlin: Springer-Verlag. pp. 347–374.
7. PANCHENKO, T. (2007) The Simplified State Model for Properties Proof Method in IPCL Languages and its use and advantages. In *Proceedings of the International Conference "Theoretical and Applied Aspects of Program Systems Develpment"* (*TAAPSD'2007*). Berdyansk. pp. 319–322.
8. REDKO, V. (1978) Compositions of programs and composition programming. *Programming*. 5. pp. 3–24.
9. REISIG, W. (2003) The Expressive Power of Abstract-State Machines. *Computing and Informatics*. Vol. 22, No. 3–4. pp. 209–219.
10. GUREVICH, Y. (1984) *Reconsidering Turing's thesis (toward more realistic semantics of programs)*. Technical report CRL-TR-36-84. University of Michigan. 13 p.

Надійшла до редколегії 05.05.2015