Application of the Method for Concurrent Programs Properties Proof to Real-World Industrial Software Systems

Taras Panchenko[1]

Taras Shevchenko National University of Kyiv,
64/13, Volodymyrska st., Kyiv, 01601, Ukraine,
tp@infosoft.ua,
WWW home page: https://www.facebook.com/tpanchenko

Abstract. Software correctness is an actual topic in many industries nowadays. Significant system properties should be checked in order to use these software safely. The properties proof task is applicable to previously developed systems also, if the property is new one or when the system is transferring to another runtime environment – for example, to multi-core processor. The method for software properties proof in interleaving concurrent environment with communication via shared memory was developed to solve the problem of simultaneous check the required property over the family of programs being run arbitrary times in parallel – instead of doing separate proofs for every number of program instances being run concurrently. Application of this method to real-world industrial tasks is demonstrated in this work.

Keywords: software correctness, shared memory concurrency, safety property, interleaving concurrency, formal methods
Key Terms: Characteristic, Environment, Process, Research, Development

1 Introduction

Software correctness problem (or programs' properties check & proof) is important and challenging in many industries and for different reasons [1]. The problem of software correctness has transformed over the time but still remains the actual one. The huge variety of program system's properties needs to be proven (or checked and warranted somehow) in order to use this software safely. First of all, it concerns safety-critical systems, but also these questions are important for many other software systems:

- do software program behaves as expected? (software validation)
- do software give us the results we need when it completes the work? (safety properties verification)
- is software system processing the request in a timely manner? (program liveness verification)

ICTERI 2016, Kyiv, Ukraine, June 21-24, 2016

There are three main approaches to solve these problems [2]:

- testing,
- model checking and
- theorem proving.

While testing techniques itself can't cover all the use cases, even behavioral, they can't warrant the total safety of the system. Moreover, Dijkstra in [3] insists on the main testing limitation – it can show an error, but it can't demonstrate the absence of errors. The two other approaches mentioned above tries to do that. But, the model checking has well-known state explosion problem [4] while pure theorem proving is rather hard technique, even for professionals. One of the advanced approaches address the problem is symbolic model checking which incorporates the pros of both latter. Detailed comparative analysis can be found in many sources, for example in [2].

Software correctness problem is actual in many cases and for different industries:

- when we deal with safety-critical (or life-critical) systems whose failure or malfunction may result in death or serious injury to the people OR loss or damage to the equipment or property OR environmental harm (aircraft, space, nuclear industries etc.),
- concerning any system especially important or mission-critical ones (financial, banking, real-time systems etc.) – to prove its correct behavior (or its correct results of work) and other formal properties,
- to check the properties of existing (already developed) systems in new circumstances – to be sure if previously known properties keep *true* for the system being migrated to another environment with changed characteristics (for example, transferring the system from single-CPU environment to multi-processor or multi-threading architecture).

So, methods for program properties check and proof are required in wide range of situations.

2 The Task and the Method

2.1 Problem Overview and Complexity

Classical questions of the program safety (partial correctness) and liveness (total correctness) are complicated by many contemporary programming techniques like object-oriented programming (OOP), concurrency, Web Services, closures and so on. For example, OOP actualizes type checking problem due to dynamical but strong typing, signature problems due to polymorphism and inheritance (plus virtual methods and dynamic linking) and visibility in the scope problems (or accessibility in the scope, i.e. kind of security). Many of them can be addressed to [1], where OOP is deeply investigated from the start and closely integrated into the method. Web Services usage, for example, arouses questions

on accessibility, reliability and discoverability (including semantic specifications like RDF) etc. But we will concentrate on concurrency and its' challenges in this work.

Main troubles of concurrency are non-determinism and inter-process communications complexity. The definitive difference between parallel and sequential worlds is that the state of program (shared memory data or messages in communication channels) can be changed between two consequent statements of one of concurrently running processes by another one, being executed in interleaving manner. In other words, the (global) state can be influenced by side effect of another concurrent process, which is impossible behavior for pure sequential programs.

Concurrency as the ability to execute more than one operation somehow at the same time can be formalized via the inter-process communication (IPC) mechanism. There exist two main approaches for IPC: message passing and shared memory. While the first approach is investigated in details, for example, in [5], the latter is rather less researched [2]. Moreover, the classical Hoare logic for reasoning on programs [6] can't be directly applied to interleaving shared memory concurrency because of possible side effect of parallel programs to each other [7].

2.2 Interleaving Concurrency with Shared Memory

Thus here we will concentrate on interleaving concurrency with shared memory environment, which is present de facto in many cases (runtime environments) in different industries and in various systems (hardware as well as software) all over us:

- hardware multi-processor and multi-core architectures (of course with shared memory), primarily SMP-architectures (Symmetric Multi-Processing),
- supercomputing, namely with UMA and NUMA (Uniform and Non-Uniform Memory Access) architectures,
- multi-tasking Operating Systems,
- software services and applications running in parallel on hardware with shared memory (like UMA, NUMA),
- servers (hardware as well as software ones, like a reactive system executing request – response cycles):
  - Web Server
  - Application Server
  - SQL Server (DataBase Management Systems as well as Data Warehouse Systems)

All samples mentioned above have common characteristics of environment:

- interleaving concurrency – means that control can be passed to any other concurrent pretender to CPU time at any time moment (also known as 'process context switch' system operation),

– shared memory – is the mean for IPC, where each process can write the values to and read the values from the so-called shared variables (at any time moment) in order to communicate with each other,
– it represents the most wide MIMD model (Multiple Instruction Multiple Data) in Flynn's taxonomy.

2.3 The Model of Environment

The formal model for interleaving concurrency with shared memory proposed in [8, 11] is the special subclass of IPCL (Interleaving Parallel Compositional Languages) [8]. Such programs will have the following form: ${P_1}^{k_1}||{P_2}^{k_2}||...||{P_n}^{k_n}$ where P_j – are sequential programs running in parallel (in interleaving concurrent mode) k_j times, and $||$ – is the interleaving concurrency operation sign. Proposed construction is an adequate model for program systems being executed in interleaving concurrent runtime environment with shared memory [8, 2].

We should notice also that proposed model follows the recommendations ("best practices") and considers warnings (avoidances) by C.A.R. Hoare [5] about parallel execution operation (like fork), that, in general, multithreading is an incredibly complex and error-prone technique, not to be recommended in any but the smallest programs and, thus, concurrency can be afforded only at the outermost (most global) level of a job, and its use on a small scale is discouraged.

2.4 Program Properties

The problem formulation and the method for program properties proof can be found in [8, 2]. Two types of properties defined there are:

1. Hoare-style: $\{PreCondition(S)\}Program\{PostCondition(S)\}$
2. Invariant-style: $Inv(S)$, which keeps *true* throughout all execution time

where $PreCondition(S)$, $PostCondition(S)$ and $Inv(S)$ are predicates over the current state (S).

The state S includes current "instruction pointers" (labels or marks) of each sequential sub-program running in parallel as well as the data – shared and local for every such sub-program. These two property types are the kinds of safety properties, while the liveness properties (first of all – termination problem) are out of our scope here because of its algorithmically undecidability in common case.

Here are some examples of such properties:

– race condition analysis,
– critical section condition,
– "transactionality" (integrity, consistency) of the system,

3 Known Alternative Approaches Overview

We should note that the right way which addresses software correctness problem is formal specification combined with stepwise refinement from an abstract model (specification) to the specific one (final code) with verification of these steps. In this case we obtain correct programs by construction. Testing techniques can be complementary only to other, more precise or formal methods, because of impossibility to test things totally. Formal methods can also be useful for proving properties of already existing systems – for example, while being transferred to another run-time environment with different characteristics or when we need to be sure in some (probably new) properties of the system and we can't (or don't want) replace current system (for some new system with these properties being proven).

Starting our consideration from famous Petri Nets regarding verification of parallel programs, we should note that this formalism was developed for concurrent processes behavior modelling and not for verification purposes. So, it can not cope with state-based properties check and proof, and fits for reasoning about flow-like properties only (data-flow or concurrent processes behavior).

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 to cope with real tasks in general. For instance, without specifying the details, original Owicki-Gries method [9] requires the quadratic number of verifications relating to the program operators count. While the extended version of the rely-guarantee Owicki-Gries method [7] 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 [10] (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 IPCL [8, 2, 11] might be one of the most efficient solutions. While IPCL is up to solve the verification problem of the parallel software, in fact, 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.

Many approaches have been adopted to deal with shared memory concurrency, but majority of them have different disadvantages [2]. One of the lacks is that known methods can be applied to a fixed program whilst we often have deal with the family of the same fixed structure program running arbitrary instances of it in parallel. So do Web and SQL servers, which run any arbitrary number of the same scripts (requests, queries) concurrently. So does operating system, executing various count of the same services and applications at any given time slice.

Consequently, we rather need the method for reasoning over the family of programs than the method which operates with a fixed program when the run-time environment is shared memory interleaving concurrency. In the latter case

we need to reason over many programs (each number of program copies running concurrently) instead of parameterized form of one of them.

4 The Method and Its Applications

The method for program properties proof in compositional nominative languages IPCL with interleaving concurrency and shared memory communication mechanism is described in [8, 2, 11] in details.

It operates over IPCL, which includes common compositions: ";", "if-then-else", "while-do", vector assignment operator ($(a, b) := (c, d)$) and the interleaving concurrency "——", which has obvious structural operational (not pure compositional) semantics [8, 2, 11]. Because of it, IPCL is similar to any common universal (structural, imperative) programming language. Thus, every program (from C, PHP, C#, Java, JavaScript, even SQL etc.) could be rewritten in IPCL without substantial problems – to apply the method.

Main stages of the method are the following [8]:

- to label (to mark-up) the program in IPCL according to its syntax tree – each particular operator should have a label (a mark),
- to specify the notion of a state – the vector, which components include:
  - labels (marks) of every particular sequential sub-program of the whole IPCL-program,
  - the single shared data of the whole program,
  - and local data for every particular sequential sub-program (if present),
- to construct transition system (i.e. the model of the program execution, or operational semantics), which fix all possible transitions between states (really, macro-transitions or transitions schema because of its countable quantity) – this could be done algorithmically by the program syntax tree using labels and states fixed above [8],
- to fix start and final states of the transition system (of the program),
- to formulate the required precondition and postcondition over start and final states respectively,
- the main step is to formulate invariant of the system – it should incorporate the program behavior as a whole, in a single predicate, and it could be completed by human only for now,
- to prove that
  - each of macrotransitions keeps invariant true,
  - precondition on starting states implies invariant,
  - invariant on final states implies postcondition.

We will not go into details of the method, but concentrate on examples of the method applications.

Here we note that by the years the method mentioned above was successfully applied to prove some properties of the industrial software systems, which are the banking systems mainly. Let us discuss those examples.

Here we consider some of the method applications to the classical algorithms and tasks (for demonstration purposes) as well as to the commercial (industrial) software systems.

4.1 Methodological Samples

First, we would like to mention some "classical" examples:

- parallel addition to shared variable consistency [12],
- Peterson's Algorithm for mutual exclusion correctness proof [13].

In first article it is proven that $i := i + 1||i := i + 1$ increases the value of i by 2. The main result is that the proof is two times shorter that in any other known formal techniques [12].

Correctness of the well-known Peterson's Algorithm (for mutual exclusion) is the subject of the latter paper. The fact that two concurrent processes can not appear in their critical section at the same time is proved in [13]. The proof again is a half of length of else known classical one (for example, in TLA).

More of that, the more generous task was discussed there – because the proof was made simultaneously over the family of each number of program run concurrently in first case, and it could be done in the second case.

4.2 Presentation System

Infosoft e-Detailing 1.0 is commercial presentation support system designed to hold the almost synchronous online+offline presentations with one lecturer (manager) and one-or-more listeners (clients). The usage of this system basically lies in switching slides on a manager's device which is almost immediately followed by an automatic switching to the same slide on each of the clients' devices connected to the current presentation session.

The most important from the correct functioning point of view is to make sure that every client will see the same slide that manager has switched to. Work of the system consists of cycles, namely switching a slide on manager's device and then switching a slide on all of clients' devices. The amount of such cycles is unlimited, the only stopping criteria is that everyone has left their presentation session. Typical cycle would look like this: manager sends to the server, and clients are reading from it, the index of a current slide (currently using HTTP + AJAX + Long Poll technologies) – in such a way the asynchronous slide replication is achieved on all the devices.

In [14] the safety property proof of Infosoft e-Detailing 1.0 software system using correctness proof methodology [8] in IPCL is presented. Operational semantics of the system with interleaving concurrency was described by means of transition system, program Invariant as well as Pre- and Post- conditions were formulated in accordance with the methodology. Application of the method to the real-world system and its usage simplicity were demonstrated in [14].

The detailed proof was given in [15] also. Thus partial correctness of an Infosoft e-Detailing 1.0 software system was proven.

4.3 Electronic Exchange

In the case with Currency Stock Electronic Exchange for Joint-Stock Company "State Oschadny Bank of Ukraine" compositional-nominative languages and the

method were used to formulate properties of the system (that each bid/ask offer is presented to stock exchange participants and that buy/sell deal is closed only once for each offer, with only one dealer, in despite of their concurrent work) and then to prove it [16]. We will not go into details here because of more interesting next example.

4.4 Remittances Payment System

Now let us consider the system for international remittances from Vigo Remittance Corp. paying out developed for Joint-Stock Company "State Oschadny Bank of Ukraine".

In this case the fact that no money remittance can be paid out twice needs to be proven [17]. The task was stated, transitional system according to the method was built for the model with simplified state (i.e. without local data), and the program invariant was formulated and proved to keep *true* over the software system at any given time. Conclusions about the convenience and adequacy of method application to prove the correctness of parallel systems were made as well as correctness property of the banking system for remittances payments was proven, namely that each money transfer can be authorized to pay out only once (due to authorization procedure) [17].

The simplified state [18] modification of the method is applicable when there is no local data within processes. In this case the state itself as well as the reasoning over the system can be significantly simplified [18].

5 Conclusions

Application of the method for concurrent programs, communicating via shared memory, properties proof to real-world industrial software was demonstrated in this work.

There are many questions left out of scope:

- software tools for automatic support for reasoning over such families of programs,
- more deep research of the proposed methodology through more industry samples (i.e. real-life software systems),
- the absence of native support of OOP in IPCL and compositional languages for now,
- non-compositional nature of interleaving concurrency etc.

But now we can affirm that the method developed is applicable and usable to proof the properties of real industry software systems.

Partial correctness of the software systems, namely Infosoft e-Detailing 1.0, Currency Stock Electronic Exchange for Joint-Stock Company "State Oschadny Bank of Ukraine" and the system for international money transfers from Vigo Remittance Corp. paying out according to an initial problem statement has been

proven using the correctness proof methodology [8, 11] in an IPCL language. Considering the difficulties in the process of such proofs in parallel environments, one can state:

- correctness proof method in IPCL is well suited for the verification of parallel programs or the software correctness proof in terms of safety properties;
- the method allows shortening the proof at the expense of choosing an adequate abstraction level [19] due to universality of a compositional nominative approach [19, 20] and by fixing the appropriate basic function set of semantic algebra.

Taking into account the flexibility of the methodology, existence of 'simplified state' model for reasoning in some cases [2, 12], and universal nature of the approach [21], it can be recommended for program properties proof (particularly safety property or partial correctness) for wide range of industrial software which is executed in interleaving concurrency environment with shared memory interaction, primarily for server-side software of client-server complexes, because of the method's usability and advances. The same conclusion was obtained in [14, 15, 17, 22] also.

6 Acknowledgments

Firstly, I would like to acknowledge my scientific mentors, Mykola S. Nikitchenko and Volodymyr N. Red'ko for their time of long discussions and attention to all my questions.

I would also like to thank the anonymous reviewers, who provided important comments and suggestions that improved this paper.

But especially I would mention and thank my dear co-authors of referenced works, namely Nataliia Polishchuk, Mykyta Kartavov, Yuliia Ostapovska and Andrii Zhygallo. These results were obtained due to their huge work as well.

References

1. The RAISE Method Group. The RAISE Development Method. BCS Practitioner Series. Prentice Hall. 493 p. (1995)
2. Panchenko, T. Compositional Methods for Software Systems Specification and Verification (PhD Thesis) [in Ukrainian]. Kyiv. 177 p. (2006)
3. Dijkstra, E. Why Correctness must be a Mathematical Concern. The Correctness Problem in Computer Science. By Boyer, R.S., Moore, J.S., (eds.) London: Academic Press. 1–8 (1981)
4. Clarke, E., Wing, J. et al. Formal methods: state of the art and future directions. ACM Computing Surveys. Vol. 28, no. 4. 626–643 (1996)
5. Hoare, C.A.R. Communicating Sequential Processes. Prentice Hall International. 238 p. (1985)
6. Hoare, C.A.R. An Axiomatic Basis for Computer Programming. Communications of the ACM. Vol. 12, no. 10. 576–583 (1969)

7. Xu, Q., de Roever, W.-P. and He, J. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. Formal Aspects of Computing. Vol. 9, no. 2. 149–174 (1997)
8. Panchenko, T. The Method for Program Properties Proof in Compositional Nominative Languages IPCL [in Ukrainian]. Problems of Programming. No. 1. 3–16 (2008)
9. Owicki, S. and Gries, D. An Axiomatic Proof Technique for Parallel Programs. Acta Informatica. Vol. 6, no. 4. 319–340 (1976)
10. Lamport, L. Verification and Specification of Concurrent Programs. A Decade of Concurrency. deBakker, J., deRoever, W., Rozenberg, G. (eds.). Berlin: Springer-Verlag. Vol. 803. 347–374 (1993)
11. Panchenko, T. The Methodology for Program Properties Proof in Compositional Languages IPCL [in Ukrainian]. Proceedings of the International Conference "Theoretical and Applied Aspects of Program Systems Development" (TAAPSD'2004). Kyiv. 62–67 (2004)
12. Panchenko, T. Parallel Addition to Shared Variable Correctness Proof in IPCL [in Ukrainian]. Bulletin of Taras Shevchenko National University of Kyiv. Series: Physical and Mathematical Sciences. No. 4. 187–190 (2007)
13. Zhygallo, A., Ostapovska, Yu. and Panchenko, T. Peterson's Algorithm for Mutual Exclusion Correctness Proof in IPCL. Bulletin of Taras Shevchenko National University of Kyiv. Series: Physical and Mathematical Sciences. No. 4. 119–124 (2015)
14. Polishchuk, N., Kartavov, M. and Panchenko, T. Safety Property Proof using Correctness Proof Methodology in IPCL. Proceedings of the 5th International Scientific Conference "Theoretical and Applied Aspects of Cybernetics". Kyiv: Bukrek. 37–44 (2015)
15. Kartavov, M., Panchenko, T. and Polishchuk, N. Infosoft e-Detailing System Total Correctness Proof in IPCL [in Ukrainian]. Bulletin of Taras Shevchenko National University of Kyiv. Series: Physical and Mathematical Sciences. No. 3. 80–83 (2015)
16. Panchenko, T. Using the Formal Specifications for Development of Electronic Stock Exchange of Oschadny Bank [in Ukrainian]. Problems of Programming. No. 1-2. 161–167 (2002)
17. Ostapovska, Yu., Panchenko, T., Polishchuk, N. and Kartavov, M. Correctness Property Proof for the Banking System for Money Transfer Payments [in Ukrainian]. Problems of Programming. No. 2-3. 119–132 (2016)
18. Panchenko, T. Simplified State Model for Properties Proof Method in IPCL Languages and its Usage with Advances [in Ukrainian]. Proceedings of the International Scientific Conference "Theoretical and Applied Aspects of Program Systems Development" (TAAPSD'2007). 319–322 (2007)
19. Nikitchenko, N. A Composition Nominative Approach to Program Semantics. Technical Report IT-TR: 1998-020. Technical University of Denmark. 103 p. (1998)
20. Redko, V. Compositions of Programs and Composition Programming [in Russian]. Programming. No. 5. 3–24 (1978)
21. Panchenko, T. Formalization of Parallelism Forms in IPCL [in Ukrainian]. Bulletin of Taras Shevchenko National University of Kyiv. Series: Physical and Mathematical Sciences. No. 3. 152–157 (2008)
22. Kartavov, M., Panchenko, T. and Polishchuk, N. Properties Proof Method in IPCL Application To Real-World System Correctness Proof. International Journal "Information Models and Analyses". Sofia, Bulgaria, ITHEA. Vol. 4, No. 2. 142–155 (2015)