Applied Computer Science, vol. 14, no. 2, pp. 20–37
doi:10.23743/acs-2018-10

Submitted: 2018-03-22
Revised: 2018-06-01
Accepted: 2018-06-21

graphical representations, Petri nets, Control Flow Graph, Concurrent Process Systems, multithreaded applications

Damian GIEBAS[*], *Rafał WOJSZCZYK*[*]

GRAPHICAL REPRESENTATIONS OF MULTITHREADED APPLICATIONS

Abstract

This article contains a brief description of existing graphical methods for presenting multithreaded applications, i.e. Control Flow Graph and Petri nets. These methods will be discussed, and then a way to represent multithreaded applications using the concurrent process system model will be presented. All these methods will be used to present the idea of a multithreaded application that includes the race condition phenomenon. In the summary, all three methods will be compared and subjected to the evaluation, which will depend on whether the given representation will allow to find the mentioned phenomenon.

1. INTRODUCTION

Applications written for modern computers are characterized by diversity and are used in almost every area of life. Many of these applications are single-threaded programs that perform tasks one by one. Along with the development of computer hardware and the introduction of processors enabling concurrent performance of tasks, multithreaded applications began to appear. Some programming languages, as in C and C++, were not created for multithreading. To complement these gaps for C language, a pthreads library was created in line with the constantly evolving standard (ISO/IEC, 2003). The C++ language has received support for multithreading in the form of an extension of the standard library, with the introduction of the C++ 11 standard (Hinnant, Dawes, Crowl, Garland & Williams, 2007).

[*] Faculty of Electronics and Computer Science, Koszalin University of Technology, 75-453 Koszalin, Śniadeckich 2, Poland, +48 94 34 78 706, damian.giebas@gmail.com; +48 94 3478 710, rafal.wojszczyk@tu.koszalin.pl

As there is a multitude of software created in these languages on the market and a lot of such software is still created, the mentioned languages have been selected to present examples of multithreaded programs.

Compared to previously used single-threaded programming, multithreaded programming has a number of advantages and a number of disadvantages (Torp, 2001). The most important of them are presented below:

Advantages

- Responsiveness – in the case of long tasks in programs with a graphical user interface, single-threaded programs undergo the so-called freezing. This problem does not occur in multi-threaded applications, as such tasks can be delegated to separate threads.
- Resource sharing – threads that run as part of a single process share computer resources. Everything happens within one address space. In the case of single threaded programs, tasks had to be delegated to separate processes and communication was done by copying values from one address space to another.
- Savings – multithreaded programs consume less memory than solutions that use several single-threaded applications.
- Scalability – multithreaded applications make much better use of the hardware capabilities of multithreading processors than a set of single-threaded applications that perform the same task together. At the same time, machines for multithreaded application states are much less complicated than machines of an analogous state of solution composed of single-threaded applications.

Disadvantages

- Complex application code – each application start-up may look different and depends on the current state of other system components. The programmer never knows how much time the scheduler has allocated to a given thread and does not know the order of their work. This state of affairs also affects:
- Debugging of such applications is very difficult because the debugging process itself can affect the behavior of the application.
- Testing the application is very difficult, because it is extremely hard to predict all possible states in which applications will be found.
- Deadlock – this phenomenon is also called jamming or blocking. A situation in which a process or thread in case of multithreaded applications orders access to resources and goes into a waiting state. It is possible that the pending process or thread will never change its state, because the resources it procures are held by other waiting processes (Silberschatz, Galvin & Gagne, 2005).

Deadlock and race condition were known before, as they occur not only in multithreaded applications but also in solutions in which single-threaded applications use shared resources.

Other known phenomena occurring in multithreaded applications are described in chapter 4 on atomicity violation and order violation (Lu, Park, Seo & Zhou, 2016), and the last one is not analyzed in this work. This work focuses on graphical representation of multithreaded applications, which will allow, above all, to reveal places where race condition is present. Control Flow Graph discussed in chapter 1 is the most well-known graphical representation that allowed to develop methods and build tools for error detection of multithreaded applications. Petri nets discussed in chapter 2 are another popular graphic representation. The graphic representations used today have a number of limitations, which affect the developed methods and tools that use them. These tools include:

- Helgrind – a tool from the Valgrind's Tool Suite to securely debug multithreaded programs that can detect any kind of problems related to parallel access to resources. On the creators' site there is information that they do not guarantee the correct operation of the application. Despite all the advantages, Helgrind does not have the possibility of remote debugging, which is necessary to work in a very large number of environments where C and C++ languages are used, e.g. in embedded systems. For more about Valgrind's Tool Suite go to http://valgrind.org/info/tools.html#others
- ThreadSanitizer – Google's tool based on Helgrind and also having its limitations. Both tools use the algorithm described in the Helgrind documentation. ThreadSanitizer is a tool included in the LLVM / Clang and GCC compilers package for the x86 platform. This tool, like Helgrind, is in beta phase and its authors do not guarantee correct operation. For more about ThreadSanitizer go to https://clang.llvm.org/docs/ThreadSanitizer.html
  - RacerX – a tool to detect race condition and deadlock phenomena described in the paper (Engler & Ashcraft, 2003) using static code analysis. Detection is carried out by creating a Control Flow Graf for the analyzed application and enriching it with a list of function calls, global variables used, pointers to variables passed as a parameter, and optionally a list of all local variables. This tool is currently not publicly available to anyone according to https://goo.gl/DgYzt5
  - Relay – a tool created at the University of California, San Diego for static code analysis to detect race condition. This tool worked on a similar principle to RacerX as described in the work (Voung, Jhala & Lerner, 2007). This tool was used to analyze the Linux kernel code version 2.6.15. The analysis was carried out on the number of 4.5 million of the code line and showed the presence of 53 places where the race condition occurred. This is the only such a detailed report on Linux kernel code analysis by static code analysis for this phenomenon. Although the tool is publicly available, it has not been developed since 2010.

The first two tools for error detection use dynamic techniques, i.e. they work with the compiled application code, while the next two work tools require the source code of the application, because they use static techniques by analyzing the source code. The use of Concurrent Processing Systems (CPS) for the detection of race condition and deadlock phenomena is an example of a static approach. The main advantage of the methods that analyze the source code is the fact that they are independent of the platform on which the application code is written, but they are unable to take into account the phenomena caused by aggressive optimization of the compiler (resulting, for example, in changing the variable's contents by a numerical constant). The phenomena caused by aggressive optimization are detected thanks to dynamic techniques, however tools using dynamic techniques are strongly related to the platform, and, for example, all aspects of Helgrind can be used only on x86 and AMD64 platforms.

C and C++ languages have already been expanded to allow for parallel work. The Cilk extension ("A Brief History of Cilk", 2017) for C and C++ was created in 1990 at MIT and commercialized as Cilk ++, and then sold to Intel, which develops them as CilkPlus. This extension has not gained much popularity and will only be used until 2018. Intel proposes migrating from CilkPlus to OpenMP framework or Intel Threading Building Block ("Intel Threading Building Blocks Documentation", 2017).

The mentioned OpenMP framework (Bull, Reid & McDonnell, 2012) was created for Fortran and C languages and then expanded for C++ 98 and is supported by the largest companies in the IT sector. The program is parallelized with OpenMP by using the appropriate preprocessor directives, which increase the complexity of the code and do not cooperate with the latest versions of C++.

Competitive solution for OpenMP, i.e. the Intel TBB library ("Intel Threading Building Blocks Documentation", 2017) for C++ is much better adapted to work with the latest versions of this language. Unfortunately, when using Intel TBB, a lot of code must be rewritten using its elements.

Charm++ ("Introduction to Charm++ Concepts", 2017) is a dedicated C++ language framework for creating applications with parallel processing. It introduces a new paradigm, i.e. object-oriented asynchronous message passing parallel programming paradigm, which decomposes the program into chares that communicate using objects called messages. The disadvantages of this solution were presented in the presentation "Charm++" (Aiken, 2017). The biggest disadvantage concerns easy to omit synchronization of chares work, which is required to avoid race condition. Another big disadvantage of Charm++ is making the programmer manage message memory. Wrong management can lead to very serious memory leaks when resources are allocated and they are not released.

The above solutions for C and C++ languages have one undesirable feature, i.e. a high level of code complexity written with their use. In the case of using the pthread library or the C++ 11 standard, this code is much more readable.

The next part of the work concerns the location of the race condition phenomenon found in the program code shown in Listing No. 1 by means of graphical representation of multithreaded applications. The program was written in C language using the pthreads library. The aim of the program from the listing below is to perform a million incrementing operations of the *balance* variable by each of the application threads. The result of the action should be a two million value. Unfortunately, incrementing operations on a shared resource are not synchronized, resulting in a race condition in the program. Synchronization should be done by using synchronization mechanisms called the mutexes provided with the pthreads library. Mutexes are abstract structures that use the mechanism of mutual exclusion to synchronize work on selected resources. The word mutex is derived from the English words mutual exclusion.

```c
#include <stdio.h>
#include <pthread.h>
static volatile int balance = 0;
void* deposit(void *param) {
        // Block B
        char *who = (char*)param;
        int i;
        printf("%s: begin\n", who);
        for (i = 0; i < 1000000; i++)
        // Block C
        {
         balance = balance + 1; // Place with uncontrolled access. Race condition place.
        }
        // Endblock C
        printf("%s: done\n", who);
        return NULL;
        // Endblock B
}
int main() {
        // Block A
        pthread_t p1, p2;
        char a[] = "A";
        char b[] = "B";
        printf("main() starts depositing, balance = %d\n", balance);
        pthread_create(&p1, NULL, deposit, a);
        pthread_create(&p2, NULL, deposit, b);
        // Endblock A
        // Block D
        pthread_join(p1, NULL);
        pthread_join(p2, NULL);
        printf("main() A and B finished, balance = %d\n", balance);
        return 0;
        // Endblock D
}
```

Listing. 1. Multithreaded application code containing the race condition phenomenon

Although it is possible to write multithreaded programs in C language, the lack of native language support causes that these programs often include race condition or deadlock phenomena. The following application code containing the race condition phenomenon will be transformed into subsequent graphical representations in which this phenomenon should be exposed, because graphic representations do not have C language limitations and are better adapted to presenting high-level ideas.

The use of graphic methods results from the need for a universal tool that will allow for code analysis and detection of the occurrence of the discussed phenomena. As already mentioned, the graphic methods are better for presenting high-level ideas than the C programming language. In addition, converting the source code into a graphical representation is a platform-independent solution to which the code is to be compiled. This transformation of the source code for graphical representation is part of the static code analysis.

2. CONTROL FLOW GRAPH

Control Flow Graph (CFG) is nothing more than a directed graph, which is one of the possible graphical representation of a multi-threaded application. CFG presented in the work (Allen, 1970) consists of nodes and edges that correspond to the next blocks of code and determine the order in which they are executed. CFG assumes the existence of 3 types of nodes. The first type of node is the entry node, which is characterized by the fact that it does not have an ancestor, but it has descendants. The second type of node is the exit node, which, similarly to the entry node, does not have descendants, but it has ancestors. The third type of nodes are nodes having both ancestors and descendants. These nodes can have at least one ancestor and at least one descendant. Ancestors and descendants can be both direct and indirect nodes. In other words, CFG is a directed G graph being a pair *(B, E)* where B is a set of nodes $b_1, b_2, b_3, \ldots, b_n$ while E is a subset of the set of all possible edges $\{(b_1, b_2), (b_1, b_3), \ldots, (b_m, b_n)\}$ occurring between these nodes.

Figure 1 presents CFG for an application whose code is on Listing 1. The code is divided into 4 logical blocks that allow its easy conversion into CFG. Block A is a fragment of the code preparing the application for working on threads, while block D is a fragment of the code terminating work on threads and ending the work of the application. Blocks B and C are a fragment of the application executed in parallel, therefore to emphasize this aspect of the application on CFG for one thread they have been labeled as B and C and for the other as B' and C'. In addition, block C is contained in block B and its work is repeated a million times.

With the exception of the main function, every logical block, i.e. any other function, the body of a loop, the body of control instructions, or any other block enclosed in braces will have its reflection in the form of a node. The main function in the C and C++ languages is the place of beginning and end of the application work, therefore it has been broken into the above-mentioned blocks A and D.

Figure 1 shows the CFG of the application whose code is on Listing 1. The diagram starts with node A in the main function. It precedes the creation of two application threads, which are represented as nodes B and B'.

Nodes C and C' are the nodes corresponding to the body of the for loop, so until the loop condition is always true, the block will still be executed, as indicated by the presence of edges *(C, C)* and edges *(C', C')*. After completing the loop operation, the control returns to the main body of the function, i.e. to blocks B and B'. The program already has only the D block responsible for the end of work and the corresponding node D finishes the graph.

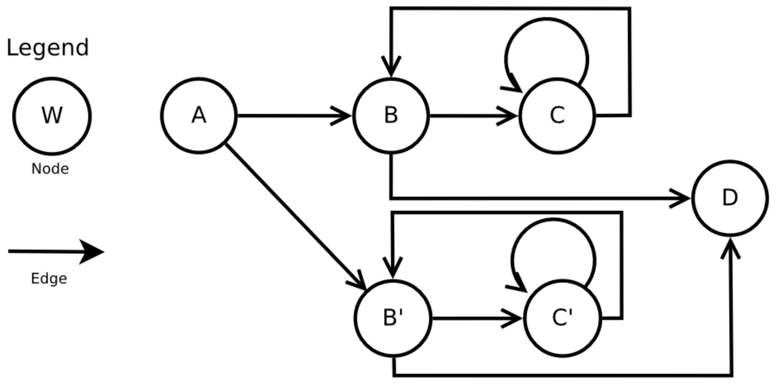

Fig. 1. Control Flow Graph of application from Listing 1

The created CFG reflects exactly the order of the blocks of code executed, however, the information about operations on shared resources cannot be read out of it. These operations occurring in block C of the application do not show their graphical representation, therefore the CFG diagram will be identical for both a properly functioning application and the one in which the race condition phenomenon is located. Hence, CFG is not a good enough notation to detect race condition and deadlock phenomena.

Another disadvantage is the fact that CFG does not allow showing blocks' nesting. Without an exact description, one can get the impression that after the block C exit and returning to block B, it is possible to return to block C again, which is not possible in the application.

Information that a given block is executed in two independent threads allows for the presumption that this is the place where the race condition may occur. In the case of systems where threads do not share any resources, it would also be necessary to check all such places. In a situation where the resource is shared by two threads that do not have common blocks, this mechanism is insufficient to locate the race condition phenomenon.

Control Flow Graf is used in tools to detect these phenomena, e.g. in the RacerX tool, each of the CFG nodes created by this tool is additionally enriched with lists of function calls, global variables used, pointers to variables passed as a parameter, and optionally a list of all local variables. Only when there is a set of all this information it is possible to detect the phenomenon of race condition.

3. PETRI NET

Petri Net (PN) is a formal information flow model designed to describe asynchronous systems in which tasks are carried out in parallel. Petri nets consist of places and transitions connected by directed edges (Peterson, 1977). The flow of information is demonstrated by moving tokens between places by passing through the edges. On the edges there are transitions, which are responsible for the permission to make the transition, and this happens when there are tokens at all the entry points of the transition. The simplest example of PN is shown in Figure 2. There are two places on it, p1 and p2 and one transition t1. The token located in place p1 will be moved along the edges to the place p2, because the transition condition is met, i.e. place p1 is the only entry point of the t1 transition and it has a token.

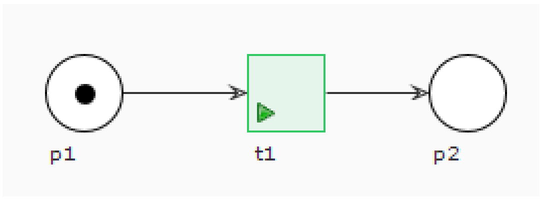

Fig. 2. Example of Petri net

Unlike CFG, PN is not built only on logical blocks of code. The construction should take into account such things as the initial state of some elements, i.e. the place reflecting the loop counter should have as many tokens as the iterations will be made by the loop, or information about the order of execution of individual tasks in case they can be done in parallel. Net from figure 3 showing the application of the mutual exclusion mechanism, which imposes the order of tokens shifting from place p1 through the transitions t1 and t2 is the example of the above.

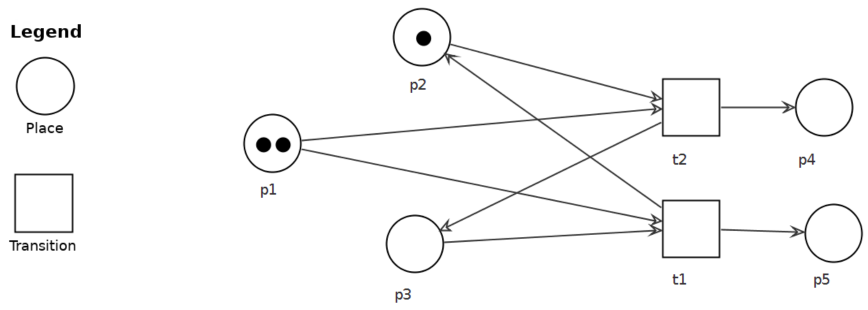

Fig. 3. Example Petri net with the mechanism of mutual exclusion

The possibility of using mutual exclusion mechanisms (it consists of the transitions t1, t2 and places p2, p3 in the above example) allows to control token shifts in the net and simulate application multithreading. However, it is not a realistic representation. In the case of multithreaded applications whose main purpose is the speed of the operation, the programmer does not impose their execution order. It is the scheduler to decide which thread is currently working while the situation of alternating thread work, as shown in the figure above, is unlikely.

In Figure 4, the Petri net is presented for the application under consideration. This net is built from 6 places and 4 transitions. Place p1 corresponds to block A of the selected application and means its start. Place p2 is the equivalent of the moment of starting both threads of the application. In the case of the Petri net, we can simulate the operation of the for loop, so block C in this case will consist of places (p3, p5, p8) and transitions (t2, t3) for the first thread, as well as places (p3, p4, p8) and transitions (t2, t4) for the second thread. Place p3 is a loop count, which should have a million tokens, because so many iterations are executed by each loop in the threads. This will enable every branch of the net to perform a million times as in every thread a million operations are performed on a shared resource.

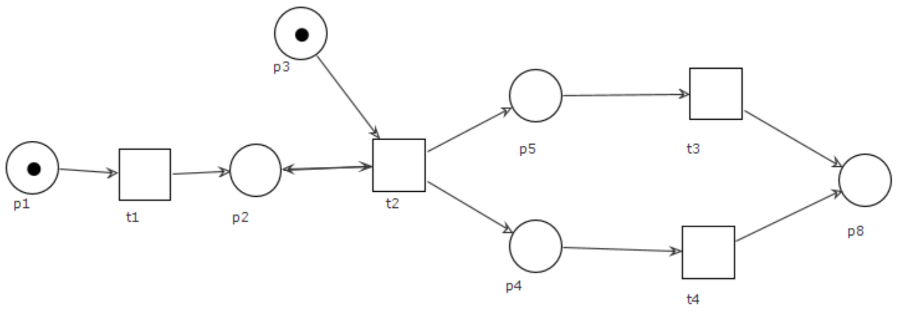

Fig. 4. Petri net of multithreaded application from Listing No. 1

Place p8 is the equivalent of application block D and ends the whole network, and the number of tokens corresponds to the value of the balance variable. If there were one million tokens in place p3, i.e. the maximum number of iterations of the loop in block C, then after the simulation there will be 2 million in the place p8.

The structure of the net does not allow the occurrence of a situation in which the race condition occurs, therefore the result of the net operation will be consistent with the expected result of the application but not with its real operation.

An additional disadvantage of such representation is that many net models can be built into one and the same application code. This situation causes that when a net model for the application is created, one can never be sure that all necessary information can be read from it to locate the information one is looking for.

In the case of using the mutual exclusion mechanisms in the net, it should always be determined which transition will have a priority resulting in a pre-determined order of operation of the transition. This is not the case in applications. The programmer is never sure which thread will be given access to the resource first, because the work of the threads is set to execute tasks as quickly as possible and they are executed immediately when the scheduler assigns the processor time to the thread. Unlike nets, the mechanisms of mutual exclusion provided with the C language do not enforce the order of threads.

4. FORMULATING THE PROBLEM

The presented analysis of two widely known graphical representations of multithreaded applications allows to conclude that using them to find the location of the race condition phenomenon is very complex and in many cases requires the use of additional (redundant) control mechanisms.

A multithreaded application code written in C language using the pthreads library is available.

The limitations result from the syntax of the C language, its grammar and the fact that the calculations must be performed in parallel.

Therefore, the question is as follows: is the application code correct, i.e. are there no phenomena like:

– deadlock,
– race condition?

The two previously discussed graphical representations did not allow to determine whether the code on Listing 1 is free from these phenomena. In item 4, a representation using models of concurrent processes for this purpose will be presented. Two representations will be shown, where the phenomenon of race condition will be visible on the first one and the second one will present a solution to eliminate this phenomenon.

5. SYSTEM OF CONCURRENT PROCESSES (Polish: SWP) FOR MULTITHREADED APPLICATIONS

The process system is a set of processes $P=\{P_i \mid i=1, ..., ln\}$ performing operations based on a set of shared resources $R=\{R_k \mid k=1, ..., lm\}$. Concurrent execution of processes means that each successive operation of one process begins before the end of another process operation and is associated with limited access to shared resources (Banaszak, Majdzik & Wójcik, 2008). A specific case refers to the systems in which processes are carried out cyclically (i.e. process operations are repeated many times over fixed time periods). In this approach, the System of Concurrent Cyclic Processes (Polish: SWPC) is understood as a set of concurrent cyclic processes that are related to each other through the use of shared resources (Bocewicz, Banaszak & Wójcik, 2006; Bocewicz, 2013).

When talking about SWPC one should mention the conflicts of resource demands, which are a consequence of the occurrence of, among others, such phenomena as starvation and blockade. Similar phenomena can be found in multi-threaded applications. Starvation occurs when one of the threads of the application over its entire duration does not release the specified resource and thus prevents access to other threads. Deadlock, on the other hand, occurs when two threads (or more) attempt to gain access to the resources they occupy, and so-called resource request cycle occurs. This situation causes that each thread waits for the remaining ones to release their resources, which never happens.

Another specific phenomenon of multithreaded applications refers to the race condition - a situation in which the status of a shared resource (e.g. the value of a variable represented by this resource) is changed by one of the threads when other threads perform operations with an already obsolete resource value. The consequence of such a phenomenon is the possibility of obtaining various results of applications (often difficult to predict) depending on the order of access of threads to shared resources.

Similarly to the CFG and PN models discussed in the previous sections, systems of concurrent cyclic processes can also be used to represent multithreaded applications. A set of graphic elements is used for its purpose (Figure 5), consisting of:

- shared resources representing an instance of any type that is shared among threads, e.g. by means of a pointer or as a global variable,
- internal resources of threads, which like resources are shared by instances of any type, and their period of life lasts as long as the life span of threads,
- cyclical processes representing the threads of the application,
- synchronization mechanism (mutex) ensuring mutual exclusion of processes on resources. In C language, a mutex is an algorithm implemented in the form of an object on which blocking and releasing operations can be performed.

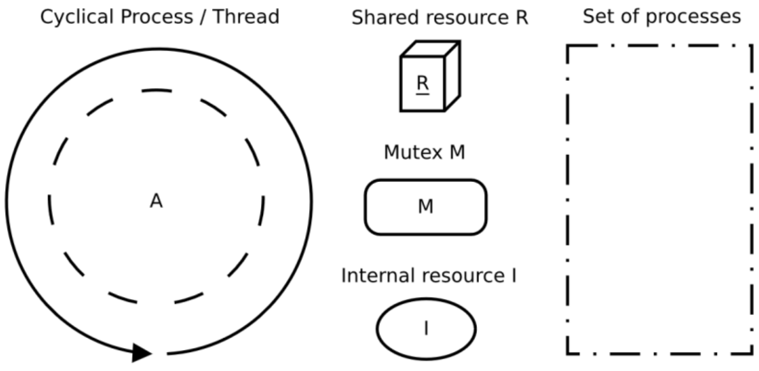

Fig. 5. SWPC elements used for multithreaded application modeling

Both resources and mutexes can be a base for cyclic processes performed (the names of these operations are given inside the resource).
These are, e.g.:

- Inc – resource increment operation,
- Lock – operation of placing a lock on a mutex object,
- Unlock – the operation of releasing the lock from the mutex object.

The proposed SWPC model (using the proposed set of elements), unlike the Petri net and CFG, hides many implementation details. It will highlight only those features of the application that are important to assess its correctness (in terms of the occurrence of phenomena leading to conflicts of resource demands). This approach should allow for accurate reproduction of the application from the model and at the same time indicate the places where race condition or deadlock can occur.

Fig no. 6 presents SWPC for the application from Listing No. 1. It differs significantly from the Petri net and CFG. The system includes a pair of processes (A, B) corresponding to both threads of the application under consideration. The A and B processes are within one set, as both threads work within one application. Both processes perform the operation of increasing the value of a shared resource called balance or increase the value of their internal resources in a similar way to the threads of the sample application. The remaining elements of the application, i.e. displaying information on the standard output, initializing variables or terminating the work of threads are hidden, because they are unnecessary in the process of detecting the phenomenon of race condition.

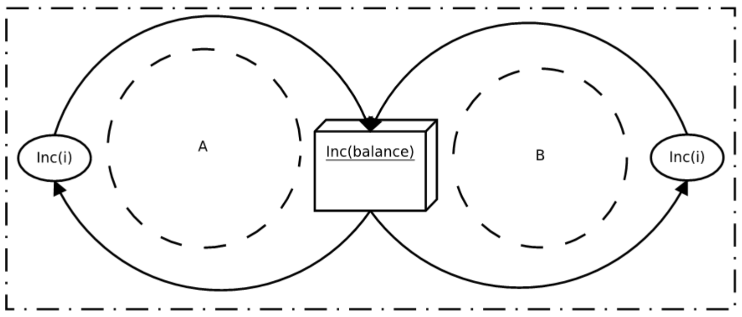

Fig. 6. SWPC model of a multi-threaded application from Listing No. 1

Although very general. the presented SWPC has the necessary information regarding the reconstruction of the application under consideration. The programmer receives a set of information that allows to recreate its code. It is easy to see in the figure that work on a shared resource is not synchronized, i.e. there is no mutex that ensures mutual exclusion of processes on a shared resource. This means that race condition on this resource may occur.

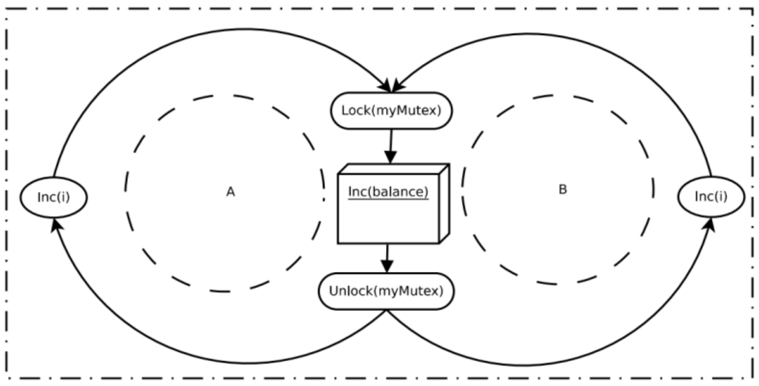

Fig. 7. The SWPC model of Listing 1 application without race condition error

Hiding unnecessary details about threads implemented in the application makes the model very clear. Omitting the implementation details does not affect the assessment of the correctness of the application. In contrast to PN and CFG, the SWPC model enhances the sensitive elements of the application, which translates into a better presentation of how the application works and allows to locate places where potential errors may occur.

Eliminating the error resulting from the occurrence of race condition is possible as a result of adding synchronization elements. Figure 7 shows the SWPC model with mutexes that eliminate race condition. In the presented solution, processes before sharing a shared resource block access to it (Lock (myMutex)) and then release it (Unlock (myMutex)) after completing operations on this resource.

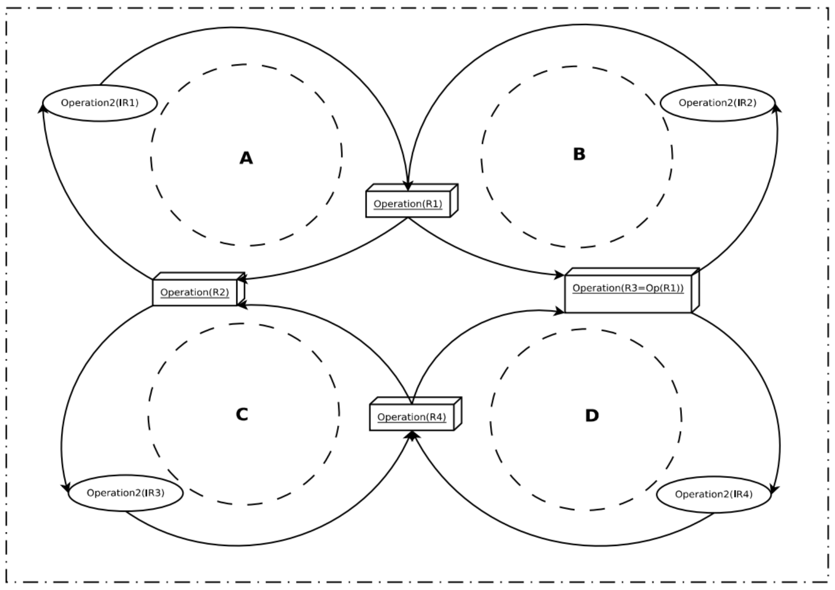

Fig. 8. Sample SWPC model of the exemplary application without synchronization elements

The application from Listing No. 1 is an example for which SWPC is not complicated. Figure 8 includes SWPC model for an exemplary application with four threads. The application has four shared resources R1, R2, R3 and R4, and each thread works with two of them and with its own internal resource. Additionally, in the B-thread, the operation on the R3 resource is dependent on the new R1 resource value (this relationship is expressed by the R3 = Op (R1) equation included in the graphic element). The figure clearly shows that the operations carried out on the shared resources are not synchronized, therefore, a race condition may occur. In addition to race condition, atomic violation is also present in the application. This phenomenon is a consequence of the relationship between R1 and R3. The state of the resource R1 affects the state of the resource R3. Before process B performs an operation on resource R3, the state of resource R1 can be changed by process A.

The elimination of the race condition comes down to placing 4 mutexes: mutex1, mutex2, mutex3, and mutex4 in application in order to ensure mutual exclusion of processes on shared resources – the appropriate SWPC is shown in Figure 9. Before each operation, a lock action on the corresponding mutex is performed on the shared resource, and after its execution, this mutex is released.

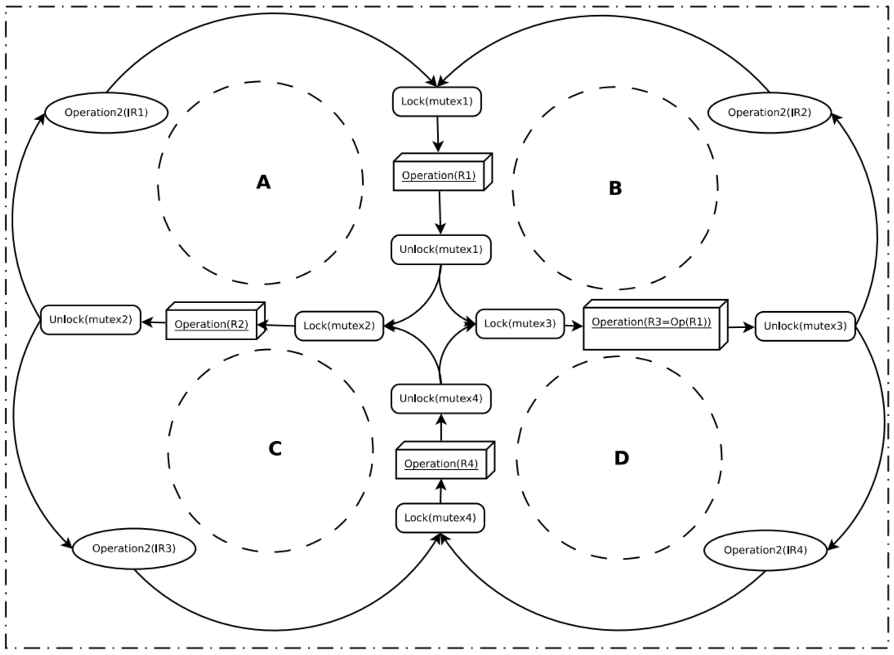

Fig. 9. An example of the SWPC model of an exemplary atomic violation application

Unfortunately, this approach does not eliminate the phenomenon of atomicity violation. This phenomenon is still present because thread B, after releasing mutex1, goes to the mutex3 blocking operation and unsecured by thread B R1 resource can be changed by thread A. One of the acceptable ways to eliminate this phenomenon is to introduce an additional mutex, which in the B-thread will control working on both resources, and in the A-thread it will control only operations on the R1 resource. A model with such a mutex is presented in Figure 10. Eliminating the phenomenon by adding another mutex increases the risk of blocking but there is a better solution, i.e. without adding mutex5.

The solution that will eliminate the phenomenon of atomicity violation without adding mutex5 is shown in Figure 11. The role of the element synchronizing the work of the B thread was received by mutex1 so that the excess mutex could be removed. The moment thread B starts work, it blocks the possibility of working on shared resources for threads A and D until it finishes the work.

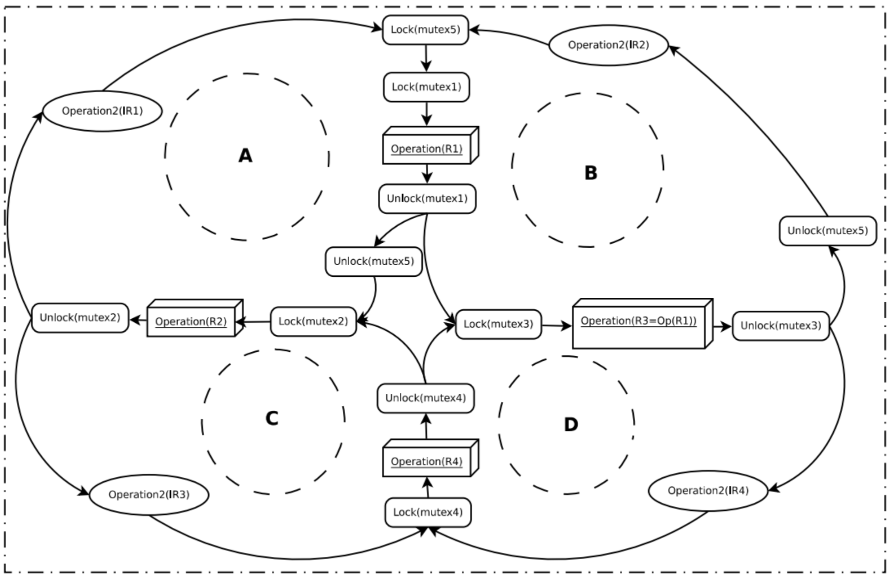

Fig. 10. An example of the SWPC model of an exemplary application with atomic violation over redundant mutex

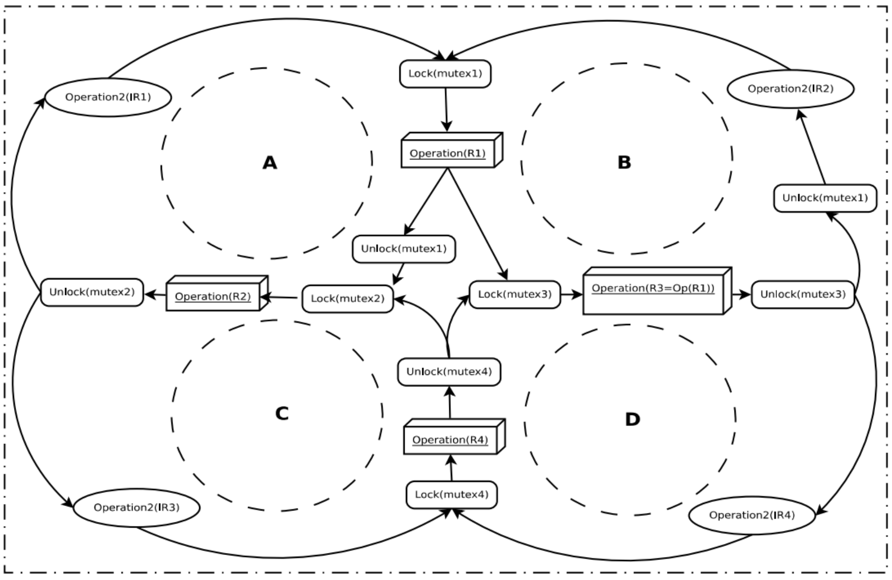

Fig. 11. An example of the SWPC model of an exemplary application with a solution for atomicity violation with a minimum number of mutexes

The models presented for the example application show that thanks to SWP, it is very easy to locate not only the phenomenon of race condition but also the phenomenon of atomicity violation. An additional advantage of SWP lies its readability, which allowed for the optimization consisting in the removal of the excess mutex. This operation will affect the speed of the application, as there are less blocking and unblocking operations, which can be very expensive.

6. CONCLUSION

All three presented representations have their advantages and disadvantages. In terms of multithreaded applications, CFG should be used when the objects of interest include the number of logical blocks and the order in which they are executed. Unfortunately, CFG is a very general graphical representation and is not suitable for analyzing relationships between threads without additional information about individual code blocks that are presented as nodes.

Petri nets are a much more sophisticated tool. They show the mechanism of mutual exclusion and the flow of information. However, the complexity of the net will increase with the complexity of the application, and an attempt to optimize it may hide important details. For each multithreaded application, it is also possible to create many different PNs. Each of the nets can work exactly as assumed by the multi-threaded application concept, while none of them will work as a real application when the application has race condition.

The method using SWP models seems to be a much better solution than the two previous methods. It hides most of the implementation details, highlighting those places where race condition, atomicity violation or deadlock may occur, which like atomicity violation is a phenomenon resulting from incorrect setting of mutexes. Interpretation of the SWP model is much simpler than in the case of PN or CFG and the extension of the notation allowed to locate the place of the race condition error in the example application. In addition, a small change in the SWP model showed how to solve the race condition in the example application or atomic violation in the example application in chapter 4. The method using SWP models is disadvantageous because they were not created for multithreaded applications. For the purposes of this article, it was necessary to extend the standard notation so that it could express all the necessary elements of a multithreaded application.

REFERENCES

A Brief History of Cilk. (n.d.). Retrieved September 16, 2017, from https://www.cilkplus.org/cilk-history

Aiken, A. (October 28, 2017). *Charm++*. Retrieved from https://web.stanford.edu/class/cs315b/lectures/lecture11.pdf

Allen, F. E. (1970). *Control Flow Analisys*. Retrieved July 5, 2017, from http://sumanj.info/secure_sw_devel/p1-allen.pdf

Banaszak, Z., Majdzik, P., & Wójcik, R. (2008). *Procesy współbieżne. Modele efektywności funkcjonowania.* Koszalin: Wydawnictwo Uczelniane Politechniki Koszalińskiej.

Bocewicz, G. (2013). *Modele multimodalnych procesów cyklicznych.* Koszalin: Wydawnictwo Uczelniane Politechniki Koszalińskiej.

Bocewicz, G., Wójcik, R., & Banaszak, Z. (2006). Harmonogramowane pracy wózków samojezdnych w warunkach ograniczonego dostępu do współdzielonych zasobów ESW (Model logiczno-algebraiczny). In *Postępy robotyki: Systemy i współdziałanie robotów*. Warszawa: WKiŁ.

Bull, J. M., Reid, F., & McDonnell, N. (2012). A Microbenchmark Suite for OpenMP Tasks. In: B. M. Chapman, F. Massaioli, M.S. Müller, M. Rorro (Eds), *OpenMP in a Heterogeneous World. IWOMP 2012. Lecture Notes in Computer Science* (271–274). Berlin, Heidelberg: Springer.

Engler, D., & Ashcraft, K. (2003). RacerX: effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review, 37*(5), 237–252. doi:10.1145/1165389.945468

Hinnant, H. E., Dawes, B., Crowl, L., Garland, J., & Williams, A. (June 24, 2007). *Multi-threading Library for Standard C++*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2320.html

Intel Threading Building Blocks Documentation. (n.d.). Retrieved September 16, 2017, from https://software.intel.com/en-us/tbb-documentation

Introduction to Charm++ Concepts (n.d.). Retrieved September 16, 2017, from http://charmplusplus.org/tutorial/CharmConcepts.html

ISO/IEC. (2003). Information technology - Portable Operating System Interface (POSIX) – Part 1: Base Definitions (9945-1:2003).

Lu, S., Park, S., Seo, E., & Zhou, Y. (2008). Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)* (pp. 329–339). New York: ACM.

Peterson, J. L. (1977). Petrie Nets. *ACM Computing Surveys (CSUR), 9*(3), 223-252.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2005). *Operating System Concepts.* USA: John Wiley & Sons

Torp, K. (November 19, 2001). *Multithreading*. Retrieved from http://people.cs.aau.dk/~torp/Teaching/E02/OOP/handouts/multithreading.pdf

Voung, J. W., Jhala, R., & Lerner, S. (2007). RELAY: static race detection on millions of lines of code. In Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (pp. 205–214). New York: ACM.