*Applied Computer Science, vol. 16, no. 4, pp. 103–117*
*doi:10.23743/acs-2020-32*

*Submitted: 2020-11-30*
*Revised: 2020-12-07*
*Accepted: 2020-12-14*

*order violation, conflicts of resources, static analysis of the code*

*Damian GIEBAS*[*], *Rafał WOJSZCZYK* [0000-0003-4305-7253][*]

# ORDER VIOLATION IN MULTITHREADED APPLICATIONS AND ITS DETECTION IN STATIC CODE ANALYSIS PROCESS

**Abstract**

*The subject presented in the paper concerns resource conflicts, which are the cause of order violation in multithreaded applications. The work focuses on developing conditions that can be implemented as a tool for allowing to detect these conflicts in the process of static code analysis. The research is based on known errors reported to developers of large applications such as Mozilla Firefox browser and MySQL relational database system. These errors could have been avoided by appropriate monitoring of the source code.*

## 1. INTRODUCTION

The authors of some works concerning multithreading stress the need for diagnostic, monitoring or code optimization tools for developers, which will facilitate the so-called debugging process (Lu, Park, Seo & Zhou, 2008; Savage, Burrows, Nelson, Sobalvarro & Anderson, 1997). The basis for detecting such phenomena as race condition, deadlock, atomicity violation and order violation is the knowledge of resource conflicts which result in the mentioned phenomena. The conditions developed on the basis of resource conflicts research allow to carry out the process of static analysis of the source code to detect them (Giebas & Wojszczyk, 2020b; Lu et al., 2008; Park, Vuduc & Harrold, 2010). Phenomena such as race condition and deadlock are very well researched, and the literature contains many well documented methods allowing to locate the conflicts causing them (Bishop & Dilger, 1996; Cai, Wu & Chan, 2014; Giebas & Wojszczyk, 2018; Jin, Song, Zhang, Lu & Liblit, 2011; Netzer & Miller, 1992). Conflicts of

[*] Faculty of Electronics and Computer Science, Koszalin University of Technology, Śniadeckich 2,75-453 Koszalin, Poland, rafal.wojszczyk@tu.koszalin.pl

resources resulting in the phenomenon of atomicity violation are more complex than those concerning the previously mentioned phenomena, but there are also further successes in this field (Chew & Lie, 2010; Jin et al., 2011). The knowledge of resource conflicts causing a given phenomenon makes it possible to develop conditions allowing to analyse the code structure in order to detect them (Giebas & Wojszczyk, 2018, 2020a, 2020b, 2020c).

It turns out, however, that the atomicity violation, order violation and other undesirable phenomena can only occur in specific environments or on specific hardware configurations, as mentioned by Mozilla Firefox developers (Lu et al., 2008). Today, the multitude of combinations of settings, environments, and hardware configurations is so vast that it is impossible to perform enough tests in a real time to determine that the selected application is free of resource conflicts causing even one of the undesirable phenomena. As a result, applications are tested only on the most popular hardware platforms in environments based on the most popular operating systems. However, this process also has a number of disadvantages. Research conducted in 2017 showed that both developers and testers were usually unable to give the correct sequence of threads (Abbaspour Asadollah, Sundmark, Eldh & Hansson, 2017), i.e. knowledge of the scenario predicted by the architect or programmer implementing the indicated functionality is sometimes insignificant among other team members. In addition, the analysis of bug reports showed that the highest number of errors related to the phenomenon of order violation was classified in the Minor group, i.e. the fourth group on a scale from 1 to 5, where 5 are the least significant errors and 1 are the most significant ones. Therefore, the awareness of the threats posed by the phenomenon of order violation seems to be very low, which directly influences the amount of time spent on examining the causes of this phenomenon.

Data on the time needed to repair various types of errors were also analysed. The analysis shows that the repair of errors related to multithreading was 82 days on average, while the repair of errors not related to multithreading takes 66 days on average (Abbaspour Asadollah et al., 2017). This combined with the fact that very often the first modification of the code does not fix the error (Lu et al., 2008), it can be concluded that the average time spent by developers on fixing multithreaded errors is too short.

This work focuses on developing a condition for detecting resource conflicts that cause order violation. The element necessary for locating the searched conflicts turned out to be the sequential relations developed within the work (Giebas & Wojszczyk, 2020b).

A new definition of the phenomenon of order violation was developed as well. The own contribution should also include a review of actual errors in the open-source software and their analysis in order to develop conditions for locating resource conflicts causing the phenomenon of order violation. After the conditions have been developed, it is possible to implement the method as a computer program, used to code optimization.

The section after the introduction is a review of the state of knowledge in the field of multithreaded applications and the phenomenon of order violation. Section no. 4 describes research on known and well documented disorderly errors from Mozilla Firefox and MySQL relational database system, which is used in many software and scientific research (Abdulhamid & Kinyua, 2020). Section 5 formulates the problem and section 6 presents a sufficient condition. Section 7 discusses, among other things, the assumptions and limitations of the method developed. The discussion also includes the topic of checking whether the claim is true not only for the examples in section 4, but also for the order violation occurring in applications written in languages other than C language. It is worth noting that the C language is still very popular, and thanks to good optimization it is used in well-known single-board computers, e.g. Raspberry Pi (Cygan, Borowik & Borowik, 2018). Section 8 contains a leading example, where it is checked whether a simple example written in C is true. In the last section includes a summary of this work.

## 2. THE CURRENT KNOWLEDGE

An order violation is caused by reversing the order of access to two (or more) memory areas (i.e. A should always be invoked before B, but the order is not maintained during execution) (Lu et al., 2008). Thus, the application may be free of race condition, deadlock and atomicity violations, and yet its operation may be affected by irregularities.

This phenomenon has been classified to the group of phenomena of race character, as well as race condition and atomicity violation (Chen, Jiang, Xu, Ma & Lu, 2018; Torres, Marr, Gonzalez & Mössenböck, 2018; Lu et al., 2008). The character of the race should be understood as including time as one of the most important variables.

An example of such an application can be found in the order_violation_examples repository on the GitHub portal[*] in the order_violation.c file. Running this code several times may bring incorrect output in the console. This example is very simple, but it shows the essence of the problem. In order to eliminate the phenomenon of atomicity violation, 5 strategies have been proposed in the literature (Lu et al., 2008): control instructions, changing the order of operations, changing the source code structure, changing the position of operations assuming and releasing locks, and other solutions that do not fit into any of the previous groups.

The order violation in this example can be removed in two ways. In the first one, the whole loop should be placed in the critical section in function *t1f*. The second solution is to run the second thread after the first thread has finished

[*] https://github.com/PKPhdDG/order_violation_examples

working, which will ensure that the operation is launched in the right order. This example illustrates how complicated is the phenomenon of order violation.

The literature says that in one version of the Apache server code, the time needed to restore a order violation took 22 hours of uninterrupted server operation with an eight-core processor (Park et al., 2009). However, rarely does a single restoration of the phenomenon allow to understand and eliminate it. This example shows how much tools are needed to search for phenomena in real time.

One of solution is to use a different type of memory (Andrew, Mcpherson, Nagarajan, Sarkar & Cintra, 2015). The research shows that even the use of software transactional memory (STM) provided by Convoider software is not able to protect against the phenomenon (Yu, Zuo & Xiong, 2019). The authors of Convoider estimate that the use of transactional memories will allow to avoid order violation with a probability equal to 0.5%.

The phenomenon of order violation is also mentioned in the research on a testing technique called fuzzing. The ConFuzz tool, developed for the analysis of multithreaded applications, has been classified as a static code analysis tool (Vinesh & Sethumadhavan, 2020), because it reduces the application code to bitcode using the *llvm* compiler tools. The bitcode is then analysed. The results of the work do not contain any information about the location of conflicts causing the order violation, but the innovative approach may prove to be effective.

In the presented literature, it was not possible to find any clues or conditions allowing to locate resource conflicts causing order violation phenomena.

## 3. MODEL

In the following sections, Mozilla Firefox and MySQL source code fragments are also presented in graphical form, according to the source code model representation of a multithreaded application, which is as follows (Giebas & Wojszczyk, 2020b):

$$C_P = (T_P, U_P, R_P, O_P, Q_P, F_P, B_P) \tag{1}$$

where: $P$ – the program index,

$T_P = \{t_i \mid i = 0 \ldots \alpha\}$, $(\alpha \in \mathrm{N})$ – a set of all threads of $t_i$ application $C_P$, where $t_0$ is the main thread, $|T_P| > 1$,

$U_P = (ub \mid b = 1 \ldots \beta)$, $(\beta \in \mathrm{N}^+)$ – is the sequence of sets of $ub$, which are subsets of $T_P$ containing threads working in the same period of time in the program $C_P$, whereas $|U_P| > 2$, $u_1 = \{t_0\}$ and $u_\beta = \{t_0\}$,

$R_P = \{r_c \mid c = 1 \ldots \gamma\}$, $r_c = \{v_1, v_2, \ldots, v_\eta\}$, $(\gamma, \eta \in \mathrm{N}^+)$ – a collection of shared application resources $C_P$, and the following elements are sets of variable names referring to a single resource,

$O_P = \{o_{i,j} \mid i = 1...\delta,\ j = 1...\epsilon\}$, $(\delta,\epsilon, \in \mathrm{N}^+)$ – is a set of all application operations of $C_P$, which at a certain level of abstraction are atomic operations, i.e. they cannot be divided into smaller operations; an operation is understood as an instruction or function defined in the programming language; an index $i$ and indicates the number of the thread in which the operation is executed, and an index $j$ is an order number of operations working within the same thread,

$Q_P = \{q_s \mid s = 1...\kappa\}$, $q_s = (w_s,x_s)$, $(\kappa, \in \mathrm{N}^+)$ – a set of all mutexes available in the program, defined as a pair variable, mutex type, where the type is understood as one of the set values (PMN, PME, PMR, PMD), where values correspond to the lock types in the library *pthread*,

$F_P = \{f_n \mid n = 1...\iota\}$ and $F \subseteq (O_P \times O_P) \cup (O_P \times R_P) \cup (R_P \times O_P) \cup (O_P \times Q_P) \cup (Q_P \times O_P)$, $(\iota \in \mathrm{N}+)$ – a set of edges including:

1. *Transition edges* – defining the order of operations. These edges are pairs $f_n = (o_{i,j},o_{i,k})$, where the elements describe two consecutive operations $o_{i,j} \in O_P$,
2. *Usage edges* – indicating resources that change during the operation. These edges are pairs $f_n = (o_{i,j},r_c)$, in which one element is operation $o_{i,j} \in O_P$, and the other is resource $r_c \in R_P$,
3. *Dependency edges* – indicating operations depending on the current value of one of the resources. These edges are pairs $f_n = (r_c,o_{i,j})$, where the first element is the resource $r_c \in R_P$, and the second is the operation $o_{i,j} \in O_P$,
4. *Locking edges* – indicating the operation applying the selected lock. These edges are pair $f_n = (q_s,o_{i,j})$, in which one element is the lock, and the other is the locking operation.
5. *Unlocking edge* – indicating the operation releasing the selected lock. These edges are pairs $f_n = (o_{i,j},q_s)$, in which one element is the unlocking operation, and the other is the released lock.

$B_P = (B_P^{FWD}, B_P^{BWD}, B_P^{SYM})$ – set sequence:

$B_P^{FWD}$ – set of pairs of **forward**-relationship operations: $B_P^{FWD} = \{(o_{i,j},o_{a,b}); o_{i,j},o_{a,b} \in O_P\}$; the first operation from the pair forces the second operation, while the second operation does not force the first. In the further part of the work it will be marked with the symbol $o_{i,j} \rightarrow o_{a,b}$,

$B_P^{BWD}$ – a set of pairs of **backward** operations: $B_P^{BWD} = \{(o_{i,j},o_{a,b}); o_{i,j},o_{a,b} \in O_P\}$; the occurrence of the first operation from the pair does not force the second operation, while the occurrence of the second operation requires the first operation. In the further part of the work it will be marked with the symbol $o_{i,j} \leftarrow o_{a,b}$,

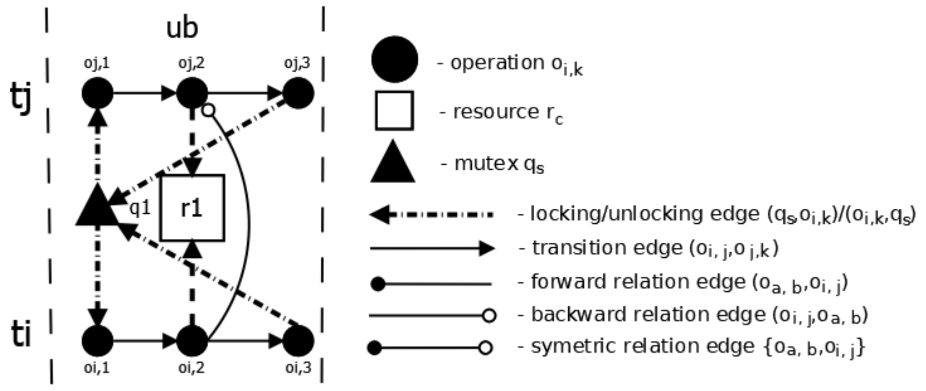
$B_P^{SYM}$ – a set of pairs of **symmetric** relationship operations: $B_P^{SYM} = \{(o_{i,j},o_{a,b}); o_{i,j},o_{a,b} \in O_P\}$; the occurrence of the first operation from the pair forces the second one and conversely, the occurrence of the second operation from the pair requires the first one to occur. In the further part of this work it will be marked with the symbol $o_{i,j} \leftrightarrow o_{a,b}$.

An extension was introduced to the model consisting in changing the definition of a **symmetrical** relation. All symmetrical relations are a set of pairs of operations, because both operations must be performed in a given order, however, these operations can occur in two different threads. As a result, a two-element set consisting of operations of two different threads does not have information which of the operations should logically be performed first.

## 4. STUDIES ON THE ORDER VIOLATION

The review of the literature on the phenomenon of order violation did not bring the expected results in the form of conditions that the source code must meet in order for a resource conflict resulting in order violation to occur. The development of such conditions has already made it possible to locate the phenomena of race condition, deadlock and atomicity violation (Giebas & Wojszczyk, 2020a, 2020b, 2020c). The resource conflicts causing the order violation phenomenon should also have a number of common characteristics, which will enable locating them. In order to find these characteristics, it is necessary to analyse several fragments of the source code, the activation of which results in the phenomenon of order violation. Therefore, based on the literature, Mozilla Firefox and MySQL source code fragments will be reviewed, in which the resource conflicts bringing order violation will be analysed. All of these code fragments have been discussed in a paper (Lu et al., 2008), which generally discusses multithreaded application errors.

The file figure_2_mozilla_firefox.c, which is located in the **order_violation_examples** repository, contains an extract from Mozilla Firefox, the execution of which will result in the order violation. The application allows for this to happen when a thread using the *mMain* function will be run first and perform a dereference operation on the *mThread* resource, resulting in an unexpected termination of the application as a result of the order violation.



ub
tj
ti
oj,1
oj,2
oj,3
oi,1
oi,2
oi,3
q1
r1
- operation $o_{i,k}$
- resource $r_c$
- mutex $q_s$
- locking/unlocking edge $(q_s, o_{i,k})/(o_{i,k}, q_s)$
- transition edge $(o_{i,j}, o_{j,k})$
- forward relation edge $(o_{a,b}, o_{i,j})$
- backward relation edge $(o_{i,j}, o_{a,b})$
- symetric relation edge $\{o_{a,b}, o_{i,j}\}$

**Fig. 1. File code figure_2_mozilla_firefox.c. as a graph**

Thus, it will be true to say that there is a **backward** relationship (Giebas & Wojszczyk, 2020b) between the dereferencing operation and the initialization operation. This example shows that in Firefox application there are **backward** relationships between two operations of two different threads, and the reversed order of these operations with shared resource results in the phenomenon of order violation.

Another file from the **order_violation_examples** repository named figure_4_mozilla_firefox.c similarly to the previous one contains a piece of Mozilla Firefox browser code. The comment in the code shows that the second thread (and thus the *DoneWaiting* function) is launched at the end of the *PBReadAsync* function. As a result, one of the operations of the first thread is the reason for starting the second thread, with both operations changing the content of the *io_pending* resource in the same interval.

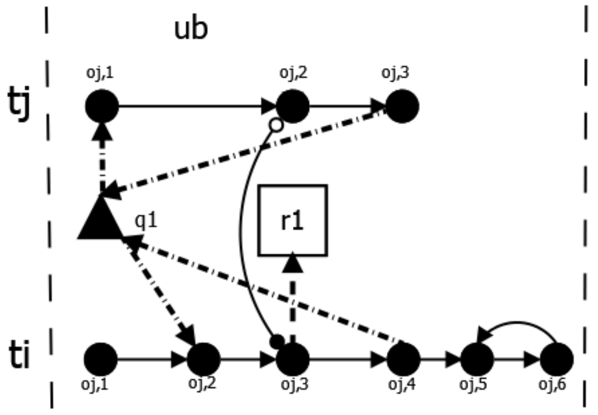


**Fig. 2. File code figure_4_mozilla_firefox.c as a graph**

From the description of the function contained in the article (Lu et al., 2008) it follows that first the resource should store the *TRUE* value and then *FALSE*. Therefore, it can be concluded that both value assignment operations are bound by a **symmetric** relation (Giebas & Wojszczyk, 2020b). The conflict has been resolved by moving the operation of assigning *TRUE* value to the resource *io_pending* over *PBReadAsync* operation. In the context of the proposed source code model of multithreaded applications, the repair was made by moving the operation to the previous time frame, so that it is certain that the *TRUE* value assignment operation will always be performed before the *FALSE* value assignment operation. Thus, as in the previous case, the resource conflict causing the order violation was the reversed order of execution of a pair of operations on a shared resource.

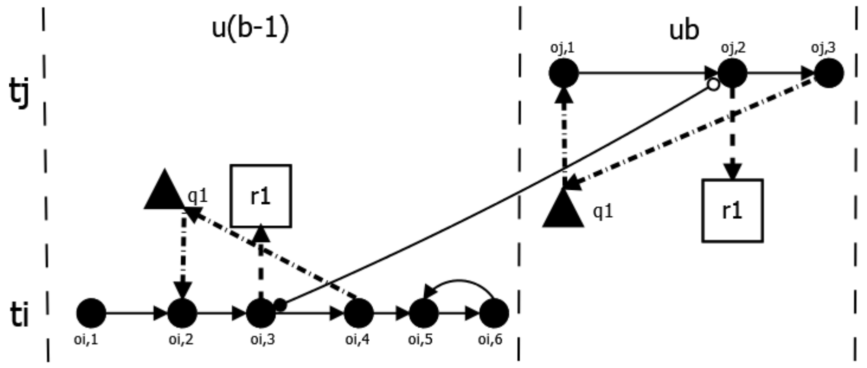


**Fig. 3. Graph of the source code from the file figure_4_mozilla_firefox.c after taking into account modifications eliminating the resource conflict**

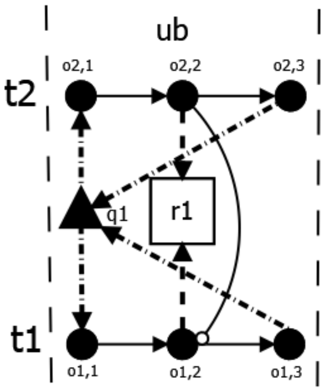


**Fig. 4. Source code from the file figure_5_mozilla_firefox.c as a graph**

The next piece of Mozilla Firefox browser code is in the file figure_5_mozilla_firefox.c of the aforementioned repository. In this case, it is the second thread operation that must be performed first. Every time *js_DestroyContext* is called, operations are performed on the shared *atoms* resource. The last time this function is executed by the first thread, the *js_UnpinPinnedAtom* function is performed, which executes the operation of freeing resources of the *atoms* variable. The result of this operation is unexpected termination of the browser operation, because in the second thread the *js_MarkAtom* function is called, whose parameter is the atoms variable with the value *nullptr*. This example is very similar to the previous two. The phenomenon of order violation occurs when the order of operations on the shared resource is reversed, which is the atoms variable. In this situation calling the *js_UnpinPinnedAtom* function cannot precede the *js_MarkAtom* function, so there is a **backward** relationship between them. The last piece of code comes from the MySQL database system and is in the figure_7_mysql.c file. In the first thread, the *dynamicId* variable is initialized, which is a shared resource. The handle for this resource is stored in the *dynamicId* variable of the *m_state* component of the node structural variable. Thus, if the second thread is run faster than the first thread, the uninitialized variable will be attempted to dereferencing, which in this case will lead to indefinite application behavior. As in the first example from Mozilla Firefox, there is a backward relationship between the two operations. The operations are performed in reverse order, with the result that a dereference is performed on an indicator variable for which memory has not been allocated, resulting in the order violation phenomenon.

The analysis of four resource conflicts resulting in the order violation, coming from large applications such as undoubtedly Mozilla Firefox browser and MySQL database system, has led to the following conclusions. The pairs of operations to which the definition of a violation of order refers should, according to the programmer's assumptions, be performed in the order specified by a certain algorithm. It is from the algorithm that a logical order is derived, on the basis of which one of the three types of relations that may occur between the operations (Giebas & Wojszczyk, 2020b) is determined. The algorithm assumes that these operations will be performed in a specific order, so the relation connecting the two operations is a sequence relation and performing the operations contrary to this order results in a violation of the order.

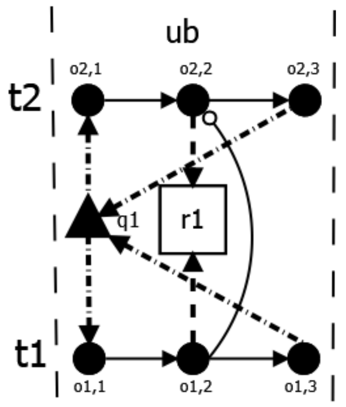


**Fig. 5. Source code from the file figure_7_mysql.c as a graph**

According to the current knowledge about resource conflicts causing the order violation phenomenon, the definition of this phenomenon is:

*Definition 1. An order violation is a phenomenon where, between two operations of two different threads (or groups of operations), there is a sequential relationship whose reversal causes the algorithm to malfunction and an undefined state of the shared resources that have been used by the algorithm.*

## 5. PROBLEM FORMULATING

The source code of the multithread application P is given, written in C using the *pthread* library. In this application there are sequential relations between operations of two threads and at least one pair of operations connected with the sequential relation is executed in the same time interval. This application is also free of race condition, deadlock and atomicity violation.

Therefore, is it possible to locate conflicts causing the phenomenon of order violation?

## 6. SUFFICIENT CONDITION

The source code model for multithreaded applications presented in section 3 will be used to develop a sufficient condition. Based on the examples presented in section 4, the statement of order violation is as follows:

*Theorem 1. Let P be a multithreaded application free of race condition, deadlock and atomicity violation. So let $B_P = (B_P^{FWD}, B_P^{BWD}, B_P^{SYM})$ will be a set of pairs of operations which are in sequential relationship with each other, and ${}^{i,j}B_P^{\xi} \subseteq B_P^{\xi}$ will be a subset containing such pairs of operations $(o_{i,\alpha}, o_{j,\beta})$, the first of which is done in a thread $ti$ and the second in the thread $t_j$.*

*If* $\{t_i,t_j\} \subseteq u_b$ *then there will be a violation of order in the implementation of the operation* $(o_{i,\alpha},o_{j,\beta})$.

Proof. Proof is a direct consequence of the definition of a violation of order. If the threads $\{ti,tj\}$ are performed in a common interval of time, i.e. $\{t_i,t_j\} \subseteq u_b$ it is therefore acceptable to implement the concurrent operation $(o_{i,\alpha},o_{j,\beta})$. This means at the same time that any order of execution of the operation is possible, i.e.: $o_{i,\alpha} \rightarrow o_{j,\beta}$, $o_{i,\alpha} \leftarrow o_{j,\beta}$, $o_{i,\alpha} \leftrightarrow o_{j,\beta}$. It is therefore permissible to violate the set order of operations $(o_{i,\alpha},o_{j,\beta})$.

## 7. DISCUSSION

The definition of order violation from section no. 2 did not give any premises as to how to search for resource conflicts causing the discussed phenomenon in the source code of the application. Only the analysis of fragments of applications containing resource conflicts causing the phenomenon of order violation, taking into account the relations described in the paper (Giebas & Wojszczyk, 2020b), allowed for redefinition of the phenomenon and development of conditions allowing for detection of these conflicts, using the source code model of multi-threaded applications.

It can be stated with certainty that the detection of conflicts causing the phenomenon of order violation will be excessive, similarly as it is the case with the detection of conflicts causing race condition and atomicity violations (Giebas & Wojszczyk, 2020a, 2020b). In other words, the results will include the so-called false-positive error. It can also be stated that, despite the redundancy, it will be possible to ignore some conflicts with poorly defined relationships between the two operations.

It is also worthwhile to verify in the future the no. 1 definition based on source code of applications other than Mozilla Firefox and MySQL, and in which there is also a violation. The applications under study do not necessarily have to be written in C language. As soon as the application code manages to determine whether functions (or methods for languages supporting only object-oriented paradigm) are in one of the three developed relationships (Giebas & Wojszczyk, 2020b), and any shared resource is involved in the whole process, an attempt can be made to confirm this definition.

The statement of order violation from section 6 allows to locate the violation in all four cases described in section 4. In each of the described examples this phenomenon occurs because the structure allows to perform the operation contrary to the programmer’s assumptions. According to the source code model of multithreaded applications, for two operations to be performed in a given order, the operations must belong to one thread. In a situation where both operations are in different threads, the order of execution can be forced only by placing operations in two different intervals. This type of solution has been used to eliminate the conflict causing atomicity violation in the second of the discussed examples in section no. 4. The graph presenting this solution can be found in figure no. 3.

## 8. LEADING EXAMPLE

Half of the examples described in sec. 4 concern the execution of an action on a resource before any memory resources are allocated to that resource. A common mistake in applications written in C by inexperienced programmers is to use indicator variables without checking the state of such variable first. In multithreaded applications it is additionally necessary to synchronize threads, so that the thread using indicator variable does not cause application failure. Such synchronization does not occur in OV1 application code located in motivation_example.c file in order_violation_examples repository. The first thread of this application is responsible for allocating space on the heap and returning the indicator to the indicator variable, and the second thread is responsible for copying to the address indicated by this indicator variable. The result of incorrect order of execution of the operation is unexpected termination of the application.

A common practice in writing multithreaded applications is to allocate memory in a different thread than other operations performed on it. In the *t2f* function of the leading example, just checking if the indicator variable does not indicate the NULL value and taking action only if this value is correct and it does not solve the problem. The programmer should ensure that the *memcpy* function receives an indicator to the allocated memory. This problem can be solved in several ways. The first way belongs to the group of naive solutions, i.e. the thread waits for the indicator to change its state by cyclic checking it in a loop, which can lead to waiting indefinitely. The second naive solution seems to be to sleep the thread for a given time by using the *sleep* function. In practice, this solution is worse than the previous one, because the time operation of the first thread is unknown, so the waiting time can be either overestimated or underestimated, and whether this value is overestimated or underestimated is strongly dependent on the hardware configuration on which the application will run. The only correct solution to this type of problem is to move the memory allocation operation with the thread to the previous time interval, as Mozilla developers have done by fixing one of the errors in Firefox.

The source code of the leading example in the model is as follows:

$T_{\mathrm{OV1}} = (t_0, \mathrm{t}_1, \mathrm{t}_2)$

$U_{\mathrm{OV1}} = (\{t_0\}, \{\mathrm{t}_1, \mathrm{t}_2\}, \{t_0\})$

$R_{\mathrm{OV1}} = \{(string)\}$

$O_{\mathrm{OV1}} = \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{0,6}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{2,1}, o_{2,2}, o_{2,3}, o_{2,4}, o_{2,5}, o_{2,6}\}$
$Q_{\mathrm{OV1}} = \{(n, PMD)\}$
$F_{\mathrm{OV1}} = \{(o_{0,1}, o_{0,2}), (o_{0,2}, o_{0,3}), (o_{0,3}, o_{0,4}), (o_{0,4}, o_{0,5}), (o_{0,5}, o_{0,6}), (q_1, o_{1,1}), (o_{1,1}, o_{1,2}), (o_{1,2}, r_1), (o_{1,2}, o_{1,3}), (o_{1,3}, q_1), (o_{1,3}, o_{1,4}), (o_{2,1}, o_{2,2}), (o_{2,2}, o_{2,3}), (q_1, o_{2,3}), (o_{2,3}, o_{2,4}), (o_{2,4}, r_1), (o_{2,4}, o_{2,5}), (o_{2,5}, q_1), (o_{2,4}, o_{2,6})\}$
$\mathrm{B}_{\mathrm{OV1}}^{\mathrm{SYM}} = \{(o_{1,2}, o_{0,5})\}$
$\mathrm{B}_{\mathrm{OV1}}^{\mathrm{BWD}} = \{(o_{1,2}, o_{2,4})\}$

Therefore, in order to locate the order violation phenomenon in the OV1 application, we must follow the theorem in section 6. Which means that the OV1 application includes a pair of operations $(o_{1,2}, o_{2,4})$, which is connected by a **backward** relationship and these operations belong to two different threads performed in the same time interval $u_2$. Both operations use a shared resource which is a *string* indicator variable. This means that the theorem is fulfilled, so there is a resource conflict in the application, which consists in reversing the order relationship resulting in the phenomenon of order violation.

## 9. SUMMARY

Based on actual errors and the current state of knowledge, a criterion has been developed in this work that can be implemented as an algorithm to locate resource conflicts in the process of static code analysis. However, the developed criterion is imprecise and may not include all real cases. On the other hand, the results obtained may be redundant, i.e. they may contain the so-called *false-positive error*. To a large extent, the location of resource conflicts that cause order violation is influenced by the correct definition of relations that may occur between operations.

Despite the disadvantages of static code analysis. it is worth to develop it, because its biggest advantage is speed. This process should not take more time than the process of compiling the program, which makes it very attractive compared to the 22 hours mentioned in the literature (Park et al., 2009). As a result, it can be used as one of the functionalities of the IDE (e.g. real-time monitoring), because in a very short period of time the programmer will receive information about, for example, the phenomenon of order violation.

As mentioned in section 7, in order to reduce the amount of *false-positive error*, further research should be conducted into the relationships between operations. Another branch of research that can be conducted is the use of the criterion developed in this work, allowing to locate the phenomenon of the violation of order, together with the source code model of multithreaded applications to develop a method based on artificial neural networks.

## REFERENCES

Abbaspour Asadollah, S., Sundmark, D., Eldh, S., & Hansson, H. (2017). Concurrency bugs in open source software: a case study. *Journal of Internet Services and Applications*, *8*, 4. https://doi.org/10.1186/s13174-017-0055-2

Abdulhamid, M., & Kinyua, N. (2020). Software for recognition of car number plate. *Applied Computer Science, 16*(1), 73–84. https://doi.org/10.23743/acs-2020-06

Andrew, J., Mcpherson, A. J., Nagarajan, V., Sarkar, S., & Cintra, M. (2015). Fence Placement for Legacy Data-Race-Free Programs via Synchronization Read Detection. *ACM Trans. Archit. Code Optim.*, *12*(4), 46. https://doi.org/10.1145/2835179

Bishop, M., & Dilger, M. (1996). Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2), 131–152.

Cai, Y., Wu, S., & Chan, W. K. (2014). ConLock: a constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *Proceedings of the 36th International Conference on Software Engineering ICSE 2014* (pp. 491–502). https://doi.org/10.1145/2568225.2568312

Chen, D., Jiang, Y., Xu, C., Ma, C., & Lu, J. (2018). Testing multithreaded programs via thread speed control. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (*ESEC/FSE 2018*) (pp. 15–25). https://doi.org/10.1145/3236024.3236077

Chew, L., & Lie, D. (2010). Kivati: fast detection and prevention of atomicity violations. In *Proceedings of the 5th European conference on Computer systems* (*EuroSys '10*) (pp. 307–320). Association for Computing Machinery. https://doi.org/10.1145/1755913.1755945

Cygan, S., Borowik, B., & Borowik, B. (2018). Street lights intelligent system, based on the Internet of Things koncept. *Applied Computer Science, 14*(1), 5–15. https://doi.org/10.23743/acs-2018-01

Giebas, D., & Wojszczyk, R. (2018). Graphical representations of multithreaded applications. *Applied Computer Science*, *14*(2), 20–37. https://doi.org/10.23743/acs-2018-10

Giebas, D., & Wojszczyk, R. (2020a). Multithreaded Application Model. *Advances in Intelligent Systems and Computing*, *1004*, 93–103. https://doi.org/10.1007/978-3-030-23946-6_11

Giebas, D., & Wojszczyk, R. (2020b). Atomicity Violation in Multithreaded Applications and Its Detection in Static Code Analysis Process. *Applied Sciences*, *10*(22), 8005. https://doi.org/10.3390/app10228005

Giebas, D., & Wojszczyk, R. (2020c). Deadlocks Detection in Multithreaded Applications Based on Source Code Analysis. *Applied Sciences*, *10*(2), 532. https://doi.org/10.3390/app10020532

Jin, G., Song, L., Zhang, W., Lu, S., & Liblit, B. (2011). Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '11*) (pp. 389–400). https://doi.org/10.1145/1993498.1993544

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). https://doi.org/10.1145/1346281.1346323

Netzer, R., & Miller, B. P. (1992). What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, *1*(1), 74–88. https://doi.org/10.1145/130616.130623

Park, S., Vuduc, R. W., & Harrold, M. J. (2010). Falcon: fault localization in concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (pp. 245–254). https://doi.org/10.1145/1806799.1806838

Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K. H., & Lu, S. (2009). PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (*SOSP '09*) (pp. 177–192). https://doi.org/10.1145/1629575.1629593

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., & Anderson, T. (1997). Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, *15*(4), 391–411. https://doi.org/10.1145/265924.265927

Torres, L. C., Marr, S., Gonzalez, B. E., & Mössenböck, H. (2018). A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs. *Lecture Notes in Computer Science*, *10789*, 155-185. https://doi.org/10.1007/978-3-030-00302-9

Vinesh, N., Sethumadhavan, M. (2020). ConFuzz—A Concurrency Fuzzer. *Advances in Intelligent Systems and Computing*, *1045*, 667-691. https://doi.org/10.1007/978-981-15-0029-9_53

Yu, Z., Zuo, Y., & Xiong, W. C. (2019). Concurrency Bug Avoiding Based on Optimized Software Transactional Memory. *Scientific Programming*, *2019*, 9404323. https://doi.org/10.1155/2019/9404323.