IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 1, 2021, accepted March 29, 2021, date of publication April 16, 2021, date of current version April 28, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3073859

Detection of Concurrency Errors in Multithreaded Applications Based on Static Source Code Analysis

DAMIAN GIEBAS AND RAFAŁ WOJSZCZYK
Faculty of Electronics and Computer Science, Koszalin University of Technology, 75-453 Koszalin, Poland
Corresponding author: Rafał Wojszczyk (rafal.wojszczyk@tu.koszalin.pl)

ABSTRACT Computer systems that allow multithreaded execution of applications have become extremely common, even small portable devices operate in multithreaded mode. This is undoubtedly very convenient for users, but for programmers it is associated with many unwanted errors, which can occur after writing application code. These errors include race condition, deadlock, atomicity violation and order violation. The subject of this work is related to the detection of these errors in the process of static software analysis. The paper presents the author's model, which was then used to detect the above-mentioned occurrences, additionally each error has been discussed in detail. A tool supporting the detection of errors in multithreaded applications was also developed and the results of this tool were presented.

INDEX TERMS Parallel programming, parallel processing, multithreading.

I. INTRODUCTION

C is one of high-level programming languages and applications written in it characterize with low memory demand and high stability. It was designed to facilitate the creation of the kernel of Unix operating system, thanks to which C enables calling of low-level procedures and functions, even though it is a high-level language. Owing to the existence of GCC and clang compilers, programming in C is available for the vast majority of operating systems. In many areas, said language has been ousted by its successor, C++, which, in turn, is being outpaced in numerous fields by languages such as Java, Python and C#. Nevertheless, C is still highly popular and is commonly utilized to create e.a. device drivers, operating systems and software from the domain of the Internet of Things [75].

In the '90s, C was extended to the POSIX Threads library (pthreads), which allows programmers to create multi-threaded software using this programming language. However, pthreads library is not the only one bringing multi-threading to C and another and still very popular one is OpenMP, which is based upon compiler directives and environment variables. In contrast, pthreads uses functions and structures of C language, thanks to which code can be more transparent and clear than in the case of the usage of compiler directives.

Along with the emergence of the possibility of creation of multi-threaded software written in C, errors such as race condition, deadlock, atomicity violation and order violation came to existence. Errors of aforementioned kind had already been known in distributed systems, as evidenced by the number of dissertations on said subject published before the dissemination of multi-threaded programming.

However, if the comparison of race condition errors in distributed systems and the kernel of the operating system was to be performed, it would be easily discernible that the environment of these errors would be differing to the extent that would make the designing of a method facilitating detection of errors of this kind in all programs unfeasible.

In Linux systems, pthreads library is mainly utilized for creation of applications destined to run in user space. For the creation of threads in the kernel, a different library is used. Nevertheless, in case of real-time operating systems with so-called micro-kernel, e.g. in the QNX system, pthreads library is used in each program. It is done so because the architecture of the QNX micro-kernel was designed with the assumption that all programs should be run in user space.

The usage of pthreads library results in noticeable growth of performance and simultaneously causes insignificant rise in the utilization of remaining resources [76]. In this

The associate editor coordinating the review of this manuscript and approving it for publication was Theofanis P. Raptis.

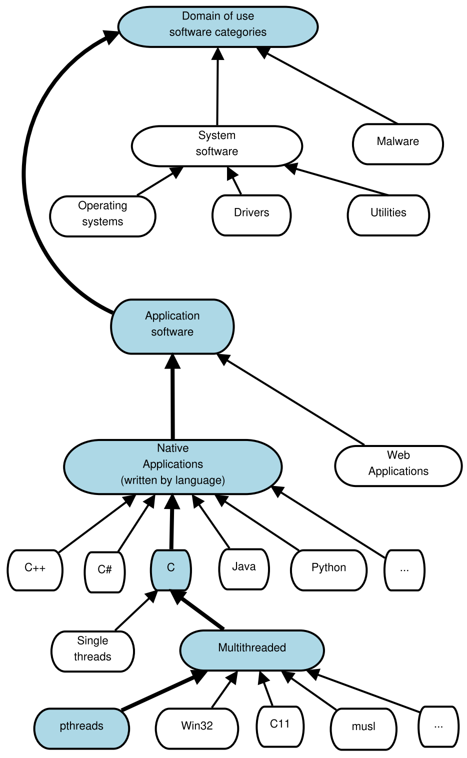

FIGURE 1. **Graph showing the classification of computer programs with important nodes marked in the field of embedded applications and IoT.**

approach, the class of computer programs considered in the paper (Fig. no. 1) applies to utility applications implemented on an energy-efficient device, based on languages supporting multithread programming (e.g. C and pthreads library). In this context, the problem addressed in this paper concerns the error detection in multithreaded applications. However, these errors can also be found in other classes of applications, where asynchronous operation occurs, e.g. in distributed systems.

This work is a continuation of [11], [19], [38], [39], where the theoretical basis for the detection of these errors is presented. This paper will present the final version of the model, theorems and evidence, as well as the unification of error detection conditions, resulting in the development of a tool to detect these error classes in the process of static code analysis.

The structure of the work is as follows: the chapter II discusses various models and methods known in the literature to detect or prevent the above-mentioned errors, then the chapter III presents the proposed model and the way of building a model instance, then a formal description of error detection in the IV model instance is presented. In the penultimate chapter, V, the results of verification of the proposed solution using the proprietary tool are presented, while the chapter VI is a summary of the work.

II. LITERATURE OVERVIEW

The commonly used POSIX Threads (Pthreads) standard enables the creation of threads within a single application and secures their synchronization using mechanisms of mutual exclusion, the so-called mutexes. Currently, the POSIX Threads standard is implemented in the form of libraries for C/C ++ languages (pthreads for POSIX-compatible systems and pthreads-win32 for Microsoft systems), PHP or the increasingly popular Go language.

C language (used in devices with low computing power - Internet of Things or Automotive), has several independent implementations of the POSIX Threads standard (pthreads-emb, pthreads-win32 _winec7, lib-pthread-embedded) as well as very similar libraries, e.g. Oracle Solaris libthread. These libraries allow to create multi-threaded applications, including for devices from Texas Instruments, Sony (PSP OS) or Oracle, which at the same time strongly supports the work on the POSIX Thread standard and its popularization, e.g. by publishing guides [3], [77].

In Automotive class solutions, multi-threading is ensured by, among others, Autosar software (Adaptive Platform package). The Autosar software package provides an operating system that complies with the POSIX standard, and with it a library that provides the POSIX Threads interface for multithreading [3].

On the other hand, in Internet of Things solutions, the official implementation of the POSIX Threads standard for C language is most often used as an element of the operating system of devices such as RaspberryPi or Odroid.

With multithreaded programming, a new category of errors has also appeared. In this case, an error should be understood as a fragment of the application code, the operation of which differs from the scenario assumed by the programmer, and the effect of such code may be unexpected termination of the application, incorrect algorithm result, data destruction or even destruction of the device.

Errors belonging to the new category are most often caused by:

- insufficient number of mutexes - the so-called race condition,
- too few operations in the critical section (not all key operations were covered by the critical section) or too many small critical sections - the so-called atomicity violation,
- placing operations in parallel threads that must be performed in a given order - the so-called order violation,
- the use of incorrect types of mutexes or the wrong arrangement of operations for setting and releasing mutexes - the so-called deadlock and livelock.

Based on the origin of error, errors specific to multithreaded applications can be divided into two categories, i.e. race-type errors and deadlock-type errors. Race-related errors include those that cause, among others, errors such as race condition, atomicity violation and order violation. The second category of errors includes those that cause, e.g. a deadlock. These errors most often arise as a result of an incorrect attempt to phase out race-related errors.

Despite the numerous advantages of multithreading, programmers often resign from using this solution due to the lack of effective mechanisms protecting applications against the occurrence of the above-mentioned errors. The reason for this is that multithreaded applications are seemingly non-deterministic, making them expensive and time consuming to test [16]. It is sometimes impossible to identify errors due to the fact that the operation of the application is affected by the state of the operating system in which the application is launched. In such cases, the analysis of all possible application operation scenarios (and thus the identification of errors) exceeds the time capacity of the available testing tools.

The methods currently used are mainly based on dynamic error identification (analysis of various application ''behavior'' scenarios). However, these methods are very invasive, as not only the pthreads library implementation or application implementation, but also the elements of the operating system are often changed. Additionally, some of the developed tools (using methods of dynamic error identification) need a database to operate, in order to store data, which will be processed only in the analysis after the end of the application life cycle. Moreover, due to the use of many different C compilers, the process of testing such an application must be repeated for different platforms and hardware configurations.

These limitations do not exist in tools for static code analysis, which allow the identification of errors based solely on its structure. However, the effective identification of errors is conditioned by the possibility of mapping all necessary relations between the threads and resources used. Building a model of the code structure of a multithreaded application program, enabling the identification of errors such as: race condition, atomicity violation, order violation, deadlock is the main purpose of the work.

A. CHARACTERISTICS OF SELECTED METHODS

The error detection issue discussed in this chapter is still open. This is because these errors can occur not only in a single application, but also between a set of communicating programs. Generally speaking, race condition, deadlock, atomicity violation, order violation, etc. are errors that occur in any system with asynchronicity, which is visualized in Fig. no. 2. Due to the differences between e.g. multithreaded programs and distributed systems, methods developed for distributed systems may not be used in multithreaded applications. As a result, after over 30 years of research on this issue, no methods that would allow a simple and unambiguous identification of errors in multithreaded applications are available yet. This is because research related to this subject is very challenging for many reasons. These include the multitude of programming languages and the programming paradigms supported by these languages, various libraries and tools that provide multithreading, as well as hardware properties, and all this effectively prevents the development of one comprehensive method. It should also be added that not all errors related to multithreading can be easily classified and then reproduced for repair [5]. Therefore, in practice, it is possible to locate a number of errors such as race condition, atomicity violation, order violation and deadlock, but also those that do not fit

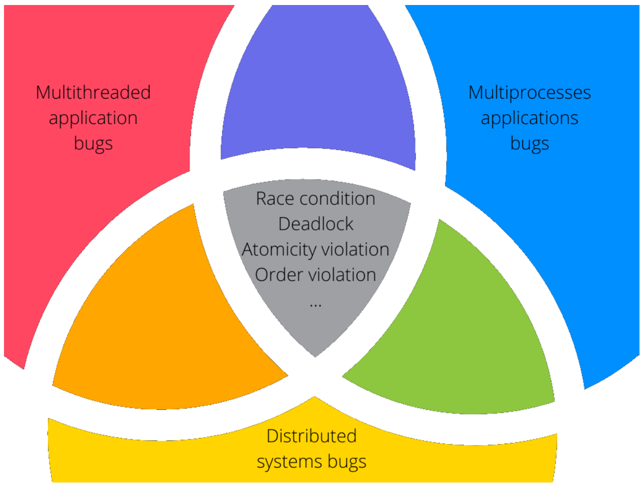

FIGURE 2. The errors in question are the common set of software where asynchronous calls occur.

into the category of race-related errors or the deadlock-type errors [1].

The literature divides error detection methods into 3 categories: static, dynamic and mixed. Static methods are methods that use the design or source code of the tested application [65] in the analysis process. Dynamic methods are based on the process of studying the behavior of the application in the runtime environment and the errors occurring inside it, which is carried out during the run of the application [65]. Mixed methods, on the other hand, are based on selecting parts of the application in the process of static analysis, and then observing the processes taking place in the application for which selected fragments of code are responsible.

The static methods include, among others the method discussed in this article, which is based on the proprietary Multithreaded Application Source Code Model (MASCM). At the same time, research on dynamic methods is also underway, for example on the method implemented in the Ewha COncurrency Detector (ECO) tool [4] or the SDRacer tool [8]. In the 1990s, it was believed that mixed methods were the best options for identifying race condition-type errors [69].

Each group of methods has some limitations, resulting from what is being studied: source code of the program or binary files (obtained after compilation). Within static methods, the identification of errors is based on the source code of the program. The disadvantages of these methods include the possibility of reporting the so-called false-positive error, i.e. indicating an error pattern in a piece of code where there is no such error [37], [70]. Static methods are very often limited to one language or a group of programming languages based on the same programming paradigms. On the other hand, dynamic methods, due to the fact that the application behavior is tested (e.g. by examining application binary files) are independent of the programming language. However, very often their application is limited to selected operating systems or hardware platforms. In addition, these methods may slow down the application's operation through their parallel operation or, in the worst cases, they may even generate new errors.

One of the biggest challenges in researching race-type errors (and probably not only this group) was indicated in [9]. Currently, there is no official database of known errors, as well as methods to assess the effectiveness of available tools and developed methods. The paper [9] proposes a DRInjector tool, which allows injecting the appropriate code with a known error into an existing C/C++ application in order to conduct research to identify it.

It should also be mentioned that a lot of research on the errors discussed in this work concerns their occurrence between different processes and even different systems working in different environments. The difference between threads and processes, e.g. in Linux, is insignificant because threads running within one process share the entire address space and other resources, while processes have their own resources and their own address spaces [20]. Sharing resources between processes is slower and more difficult to implement than sharing resources between threads, where it is enough to refer to a variable. In order to achieve the same effect between processes, it is necessary to use system mechanisms that have an internal mechanism of mutual exclusion, which prevents, for example, race condition.

One of the features of methods based on static code analysis is the close cooperation of tools implementing these methods with compilers. This is because compilers are very often a collection of different applications, which are responsible for compilation stages performed concurrently. Such an approach allows e.g. to transform the source code into a form of Abstract Syntax Tree, which can be subject to further analysis. For example, a tool that uses MASCM to convert source code to a MASCM instance uses the GCC compiler and the syntax trees generated by it. A similar procedure was used when building the DR-Frame tool, which uses the parser from the LLVM project and the clang compiler [32] to test network applications. The use of intermediates obtained during the compilation process (e.g. the aforementioned AST or debugging support files) guarantees that the data they contain is fully compatible with the application's boot environment. This reduces the occurrence of new errors that could occur during manual code parsing and interpreting.

Therefore, tools for static code analysis are often not independent, and their results are affected by tools included in compilers.

B. ERRORS IN MULTITHREADED APPLICATIONS

1) RACE CONDITION

As mentioned in the previous subsection, errors in multithreaded applications are divided into 2 groups: race-type, deadlock-type group. The most characteristic error of this group is race condition, an error resulting from incorrect application of synchronization mechanisms or from their complete omission. In other words, race condition is an error that occurs when several threads are simultaneously accessing and acting on the same data so that the result depends on the order in which the data [18] was accessed.

In 2007, Microsoft filed a patent for software using the dynamic method to detect a race condition error [6]. However, the method described in this patent proved to be ineffective and work on tools using it was abandoned.

A year later, in 2008, a document describing the method of static source code analysis for the detection of race condition [7] was sent to the American Patent Office. The method described in this patent is very similar to the method described in this article. The source code written using the object-oriented paradigm is converted to an indirect representation, in which the features indicating the presence of race condition in the application code are sought. The authors of the method indicate that it is dedicated for ANSI C and ANSI C++ languages, however, they indicate that it can be used with other languages as long as these can be reduced to an indirect representation.

The method described in [8], on the basis of which the SDRacer framework was created, was designed to detect and eliminate race condition errors in modern embedded systems. Modern embedded systems can also include IoT class systems (this framework was developed for the uClinux operating system currently integrated with the main branch of the Linux kernel). SDRacer was used to test 9 applications written in C language for uClinux. These applications, however, are not multithreaded applications, and their race condition errors are caused by the use of an interrupt service routine (ISR) in these applications, which enables asynchronous operations on selected memory areas.

One of the most common languages for investigating race condition errors is Java [14], [15], [78]. This is due to the fact that this language supports only one paradigm, i.e. an object oriented paradigm, which makes the grammar of this language very orderly, significantly facilitating the creation of tools for source code analysis of this language. These tools can be easily connected with IDE for Java. Example SWORD, static data race detector is implemented as a plugin in the Eclipse IDE [78].

Another tool for detecting race condition in Java programs is called Chord [81]. Chord uses Binary Decision Diagrams (BDDs) in scaling whole-program context sensitive analyses to detect concurrency errors. Results of research on Chord shows it is more efficient tool than RacerX described below. High efficiency of the Chord is a result of form of context sensitivity called k-object sensitivity that treats abstract contexts and abstract objects uniformly by defining the abstract contexts of an instance method as the abstract objects.

As it turns out, however, the grammar of programming languages is not a significant obstacle in order to develop tools for detection of race condition errors. To detect race condition, the RELAY [29] and RacerX [30] tools were developed, which were used to analyze source code written in C, e.g. the source code of the Linux system kernel. It is also worth mentioning that although a race condition error in almost all applications is an error that needs to be fixed, it does not always happen. As an example of the desired occurrence of a race condition error, let's use [10], where this error is used to develop a true random number generator (TRNG). In other words, the randomness associated with the occurrence of a race condition error can be used to replace commonly used pseudo-random number generators by generators with higher entropy, which will allow building better cryptographic tools.

The Race condition error can also be identified by graphical representations of multithreaded applications [11]. One of the graphical representations that allow locating race condition in the code of a multithreaded application is Petri Network (PN). The natural phenomenon in PN is the concurrency of performed actions; therefore, they are most often used for modeling concurrent systems [12] (including multithreaded). Despite many advantages, it is very difficult to use them to analyze the source code structure of multithreaded applications. This is due to the ambiguity of the received models, i.e. for a single multithreaded application one can build many network models [11], out of which not every application reproduces an error. The second graphic representation that allows locating the occurrence of race condition errors is the System of Concurrent Processes (CP). Although CP allows locating the race condition error, they were not developed with multithreaded applications in mind, and thus presenting complex multithreaded applications in them may be difficult or even impossible, because to present even a simple application, the CP notation had to be extended [11].

The [79] work proposes a method that uses Concurrent Control Flow Graph. Control Flow Graph as a model for searching concurrent errors was described in paper [80]. CFG is not sufficient to detect other concurrency errors, it cannot be used to model operations and resources as separate entities, and also did not include time intervals as one of the variables.

Another tool where CFG is used is LOCKSMITH [82]. LOCKSMITH is a static analysis tool for applications written in C. CFG used to capture operations in the program together with label flow graph is a core of for data race detection process. Authors of LOCKSMITH proposed two types of analysis context-insensitive and context-sensitive. Context-sensitive analysis reduces false positives but also limits LOCKSMITH's overall scalability.

In addition to the PN and the MASCM model discussed in this paper, attempts were made to build tools for graphical representation of race-type errors, e.g. an application created for this purpose called Bauhaus, which proved to be inefficient due to the time-consuming process of building graphic representation as well as a very large number of false-positive errors [31]. It is worth noting, however, that a good graphic representation is easier for a human being to perceive than long reports, and in the era of rapidly developing AI it may also turn out that such a form of representation will prove more effective over time.

The errors of the race condition class have been described in great detail in the [13] work for concurrency delivered through the fork/join mechanism by extending Cilc [17]. This mechanism was actually a precursor of the pthreads library, where the fork function is used to create a fork, i.e. a child process that simultaneously executes the specified code fragment. The result of the child process is attached to the parent process using the join function. Semaphores, which are similar but much simpler mechanisms to mutexes, have been used as a mechanism of process synchronization. Netzer in his work [13] used temporal ordering graph to detect race-condition errors, which he used to show relations between the child processes and the parent process, so they are not suitable to present the source code structure.

It is worth noting that in multithreaded applications written in C using the pthreads library, a race condition error occurs when at least one of the threads working in the selected period of time has writing to resource operation that is not in the critical section, or the critical section of one thread is created by a set of mutexes not used by all others. This property can be used to identify race condition errors by verifying that all critical sections have at least one common mutex - this property has been further used to develop a race condition error detection condition. It is also worth remembering that removing a race condition error does not guarantee the correctness of the application. Elimination of race condition may result in deadlock, atomicity violation or order violation errors [27], [28].

2) DEADLOCK

Deadlock-type error is manifested by the inability to gain control over the mechanism of mutual exclusion e.g. mutex, used to avoid race condition and other race-type errors [19]. The inability to gain control over the mutex most often results from the wrong order of occurrence of the operations performed within the thread, the lack of the operation to release the mutex, or placing the release operation in the wrong place in relation to the operation locking the mutex [19].

In the basics operating systems manual, a deadlock-type error is defined as a situation where several processes are waiting infinitely for an event that can only be started by one of them [18].

One of the models that allow to identification of deadlocks in multithreaded applications [71] is the previously mentioned PN. However, a directed graph [18] and Gadara Nets [21] were also used for this task, which are a variation of PN.

Deadlocks fall into two categories: resource deadlock and communication deadlock [24]. Resource deadlock occurs when threads need a certain group of shared resources to run, and each thread has reserved only a fraction of them, which results in indefinitely waiting for itself [25].

Communication deadlock errors are more abstract and more general than resource deadlock errors. They appear both in distributed systems, in interprocess communication, as well as in multithreaded applications, in which a wait cycle is created that causes a deadlock error [19]. Due to the high complexity of errors in the communication deadlock group ([24], [26]), further considerations are limited to the resource deadlock group only.

For a deadlock error to occur, four conditions must be met [18], [58]:

1) Mutual exclusion: At least one resource must be indivisible; that is, this resource can only be used by one thread at a time. If another thread requests access to a resource, it must be delayed until the resource is released.
2) Holding and waiting: There must be a thread to which at least one resource has been allocated, and which is waiting for an additional resource to be allocated, just held by another thread.
3) No expropriation: The resource is not subject to expropriation, which means that the resource can only be released at the initiative of the thread holding it, once the thread has been completed.
4) Recurring wait: There must be a set of $T_0, T_1, \ldots, T_n$ of awaiting threads such that T_0 is waiting for a resource being held by thread T_1. T_1 waits for the resource held by thread $T_2, \ldots, T_{n-1}$ waits for the resource held by thread T_n, and T_n waits for the resource held by thread T_0.

The above necessary condition is also annotated that for a deadlock class error to occur, all four conditions must be met. It was also noted that the cyclical waiting conditions implies the holding and waiting condition, so the four conditions mentioned are not completely independent.

There are four scenarios where a deadlock error occurs in a multithreaded application [19], [27]:

1) mutual exclusion of pairs of mutexes,
2) omitting the operation of releasing the mutex, e.g. as a result of a control instruction,
3) a renewed attempt to create the mutex as a result of:
    a) operation of the loop,
    b) recurrent call of the function.

To detect errors belonging to the deadlock class in applications from the cloud computing domain, a positive attempt was made to use UML activity diagrams [33]. However, the authors of the solution emphasize that the method they are developing is at an early stage of research, and its users must learn to use CSP algebra as the basic semantic domain for activity diagrams.

One of the solutions that allow eliminating the occurrence of deadlock class errors is a project called Gadara [21], [59], [60], which used the previously mentioned Gadara nets. These nets are used in the state machine, which is built during the compilation process based on the application's source code, and then built on the basis of this net, the supervisor is injected into the application. The way the supervisor works is based on Discrete Event Dynamic Systems and consists of forcing the correct state of the application when the application wants to go to a state unknown to the supervisor's state machine.

The disadvantages of the solution include the fact that the implementation of the supervisor may have its own errors, which may affect the operation of the supervised application. As a result, additional elements may affect the application in an unforeseen way, which may result in failures as serious in consequences as deadlock class errors.

An additional supervisor in a multithreaded application also means higher processor memory and time consumption, as additional operations must be performed. However, the presence of a supervisor in the program still does not eliminate the error present in the application code. This code can be compiled with a compiler other than the one from Gadara project and then used by an unaware user.

In 2014, a paper was published on a method for automatic fixing deadlock errors [22], but these methods have met with much criticism [23]. The most serious drawbacks of these solutions include the low quality of patches, which results from creating them based on a limited group of tests, not necessarily properly matched to the code to be fixed [23].

It was also undertaken to create general methods for detecting deadlocks. One of them included searches in C/C++ applications written using pthreads library or Qt framework and in Java applications [34]. In order to use it, one first needs to provide the modified pthreads library modified OpenJDK and modified Qt framework. Then we need to download and install the NoSQL Redis database, in which the information will be placed, and which also needs to be changed. The results of this database can be found in the Eclipse and QtCreator plugins. As a result, the developed method is limited to two IDEs, for which the appropriate plugins have been created. However, this method turns out to be highly useless in practice, as it requires constant development of patches with required changes and its application in environments without UI is impossible e.g. in CI\CD process.

DeadLock Analysis Models (lams) [35], [36] were developed and used to detect deadlocks in multithreaded applications. This model is used to detect errors caused by a thread that has already created a mutex, e.g. by using loop or recurrence. Developed on the basis of the lams model and the Petri Net, the method of static code analysis allows detecting only two out of four cases of deadlock error (a re-creation of mutex by the loop or recurrence).

Of course, apart from the methods listed in this section, there are also other methods that are used to avoid these errors, e.g. in operating systems or automation systems. These methods, however, very often require information specific to the fields in which they are used, so they cannot be used in the field of multithreaded applications.

3) ATOMICITY VIOLATION

The atomicity violation is one of the race errors and accounts for nearly 70% of all reported errors in this group [37]. This error consists in a disruption of the sequence relationship that connects two operations from two different critical sections of one thread that use a shared resource, and the disruption of which caused by another thread's operations on the same

TABLE 1. Scenarios of the atomicity violation. Source [40].

$read_{local}(A)$	$read_{local}(A)$
$write_{remote}(A)$	$write_{remote}(A)$
$read_{local}(A)$	$write_{local}(A)$
$write_{local}(A)$	$write_{local}(A)$
$write_{remote}(A)$	$read_{remote}(A)$
$read_{local}(A)$	$write_{local}(A)$

shared resource causes undefined operation of the algorithm, which includes these operations [38].

The four scenarios presented in table no. 1 come from [40] on Kivati, which was limited to detecting atomicity violations in Linux and x86 programs. This tool uses a mixed method by supervising the so-called atomic regions in the process of the application's operation, which are obtained by prior analysis of the application's code. The method used in the Kivati program has not received wider recognition and no further research is being done on it.

Two patents have been filed for atomicity violation error detection. The first is to analyze the processor instructions and simulate selected [41] scenarios. The main objections to this method are that it is limited to one type of synchronization mechanism and a large number of false-positive errors [42]. The authors of the patent do not indicate which synchronization mechanism is to be used, however, they force programmers to use specific, predefined structures, which require uniform synchronization mechanisms.

The second patent for the detection of atomicity violation uses a method called access interleaving invariants (AII) [43]. The method described there was originally implemented in AVIO [44] and SVD [45] tools, and a few years later it was also used in the CTrigger [37] test framework. AII is one of the dynamic methods, based on multiple analyses of the results of the application in search of the scenarios mentioned above. However, this method turned out to be ineffective, and work on these tools was suspended. Using CTrigger requires a lot of computational and time resources [37].

Java, due to its unflagging popularity, is still one of the most popular languages used in research on these errors. The paper [46] presents an algorithm called AeroDrome, which was implemented in the RAPID tool, which is used for dynamic analysis of applications written for JRE (Java Runtime Environment) to detect atomicity violation. Research on the algorithm is at an early stage of development and its authors indicate that it is not good enough and should be used as part of a hybrid method.

Solutions based on the analysis of the application during its operation, in order to locate the cause of atomicity violation, are used to develop methods independent of the language in which the program is written. With the growing popularity of JavaScript [47], this language has been applied in Node.js runtime environment for writing server applications. This environment creates applications with the help of the event-driven programming paradigm, in which many operations are performed in parallel in different threads [48]. However, this environment does not provide tools for marking operations as atomic [48]. Such a guarantee is also not provided by Rust [49], created for safe multithreading and published in 2010.

The works on the method of static analysis of Java code include a paper discussing the search for causes of atomicity violation based on the grammar of the language. The method described in the paper entitled "The method of static analysis of Java code" is based on locating in the definition of classes of methods having in the declaration the keyword *synchronized*, whose structure is then analyzed for the occurrence of patterns that guarantee the occurrence of atomicity violation [50]. The disadvantage of this method is that it is not possible to translate it into languages that differ in similarity to Java, including C.

The method called Hybrid Atomicity Violation Explorer (HAVE) [51], which is based on the use of Static Summary Tree, was also developed with Java in mind. Based on the created trees, the work of selected fragments of the application is simulated, and the effects of the simulation and the trees are the basis for building hybrid trees, which are then analyzed using the proprietary algorithm. The result of the algorithm work is information about potential conflicts causing atomicity violation and race condition. The authors of this method mention the occurrence of a false-positive error in the results of their method and further work on their elimination. They also indicate the further development of static code analysis as one of the most important elements of the developed method.

4) ORDER VIOLATION

Order violation is caused by a reordering of at least a pair of operations that have access to two (or more) memory areas (i.e., operation A should always be called before B, but the order is not preserved during execution) [1], [39]. This is another bug after race condition and atomicity violation of the race-type [1], [52], [53] the repair of which may result in a deadlock error.

Out of all the errors discussed in this paper, the order violation error is the least studied one, which is why it is also the most frequently misclassified error (often mistaken for atomicity violation) [39]. Research conducted in 2017 shows that both programmers and testers are usually unable to provide the correct sequence of thread execution [5], i.e. knowledge of the scenario predicted by the architect or programmer implementing the indicated functionality is sometimes negligible among other team members. The analysis of error reports showed that the largest number of reported order violations was classified into 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 errors. The data on the working time needed to fix various types of bugs were also analyzed and it shows that repairing errors in multithreaded applications took an average of 82 days, while repairing errors in single-threaded applications takes an average of 66 days [5]. Combined with the fact that very often the first code modification does not resolve the [5] error, it can be concluded that the average time spent by developers fixing bugs in multithreaded applications is too short.

The literature proposed 5 strategies to eliminate the order violation error [1]: using control instructions, changing the sequence of operations, changing the structure of the source code, changing the location of operations that create and release mutexes, as well as other solutions not matching any of the previous groups.

Order violation errors are also mentioned in research on the testing technique called fuzzing. ConFuzz using this technique, and other techniques designed to analyze multithreaded applications, including static code analysis [55]. ConFuzz reduces the application code to bytecode using the llvm compiler, and then using the ThreadSanitizer tool implementation for llvm, this code is subjected to a fuzzing process, in which ConFuzz collects information about the program's operation, as well as information from the tool ThreadSanitizer. Consequently, the ConFuzz results consist of 3 data sets, i.e. the application input data, the ThreadSanitizer report, and the application progress report. This process, performed for a pre-set period of time, ends with a report containing information on localized errors.

Three applications were tested with ConFuzzer: pbzip2, pigz, pixz. The analysis time for each of these applications is over 12 hours of operation. ConFuzz'er work was compared to another DumbFuzzer implementation of the fuzzing technique. These tools showed similar results when analyzing pbzip2 and pigz in favor of ConFuzzer. Pixz analysis showed no errors for both tools. Unfortunately, the results of the work do not include division into individual types of errors.

5) CONVOIDER - UNIFIED ELIMINATION OF THE DESCRIBED ERRORS

A solution for detecting all the above-mentioned errors is a tool called Convoider [54], [56]. It uses a mechanism called software transactional memory (STM). According to the creators of Convoider, the use of their tool guarantees the elimination of all deadlock, race condition, and atomicity violation errors from the application.

Software transactional memory is a software concurrency control mechanism that programmers can use to break down code into transactions and ensure that each transaction runs atomically and is isolated from the rest of the [56]. A software concurrency control mechanism should be understood as a mechanism that turns threads into separate processes, eliminates the mechanisms of mutual exclusion, and completely changes the mechanism of allocation and deallocation of resources. This solution, although very attractive, has three limitations that make its common application difficult. The first limitation is the high cost of introducing the transaction mechanism by developers, as this requires the introduction of low-level API STM calls [56]. These are then used by Convoider in the automatic transactionalization of binary files. The second is the poor compatibility with operating system kernel I\O calls [56]. The third limitation is very low compatibility with condition variables, which can lead to e.g. lost signal errors [56]. The authors of the tool also mention only the implementation of this tool for applications written in C and C ++ for Linux.

The biggest problem for Convoider is preventing an order violation error, because despite extensive research on this issue, the probability of preventing it is still only 0.5% [54], [56].

Convoider is a tool extending the method used in the abandoned tool Grace [57]. As a result, Convoider, like Grace, affects the speed of the application and increased memory consumption, and at the moment it is possible to use it only in systems with Linux kernel.

Convoider is tested on applications collected from projects STAMP, PARSEC, SPLASH2 and Phoenix [56]. Example applications from the last one were described in V-D as a part of the experiment on MASCM model.

C. SUMMARY OF SELECTED SOLUTIONS

The 2 table summarizes selected methods related to the research issue. The above-mentioned methods have been compiled in relation to the method of performed analysis (static, dynamic, mixed) and the errors detected. A direct comparison of the presented methods, e.g. on a benchmark basis, is not possible because:

- methods differ in the language of destination - despite the similarities between C and Java, the results returned by the methods could be incorrect,
- there are no tools to implement the methods or only proposals / ideas have been presented in the literature,
- in the case of direct comparison, an objective evaluation criterion should be adopted, e.g. the accuracy of the results or the cost of using the method, unfortunately, in the case of many methods, the above-mentioned criteria will be opposite to each other, e.g. methods that are expensive to use may not be accurate,
- most methods detect only one error.

Considering the above, this paper compares the available methods in relation to the characteristics that make it impossible to meet the requirements set out in the introduction to the work:

- the restriction of use only to high-level object oriented languages, which often requires high-performance devices and reduces battery life: [30], [48], [50], [66], [69], [74], [78], [81].
- limited to a specific operating system only, which excludes the application of the method to applications based on microcontrollers: [4], [30], [40], [54], [56], [57], [69], [72].
- interference into the compiler, object code, or application execution, which may interfere with the previously selected IoT hardware: [21], [54], [56].
- the need to perform simulations or, in extreme cases, a complete review, which does not allow the result to be obtained within a few minutes: [37], [44], [45], [51].

TABLE 2. List of selected methods.

Method/Tool	Method Type	Phenomena	Intended use
Detection			
AVIO [44]	dynamic	3	C language
Ewha COncurrency Detector [4]	dynamic	2, 3, 4	Applications for Linux
CTrigger [37]	dynamic	3	**C and C++ languages
Sasturkar [66]	dynamic	*1, 3, 4	Java language
Trace-based symbolic analysis [67]	dynamic	*1, 3, 4	C language
AeroDrome [46]	dynamic	3	Application written for JRE
Eraser [69]	dynamic	*1, 3, 4	Digital Unix operating system, AltaVista Web search engine, and **other unspecified software
RacerX [30]	static	1, 2	Linux, FreeBSD and other **C programs
DeepRace [73]	static	*1, 3, 4	OpenMP and POSIX programs
ERIGONE [83]	static	2	**C language but also can be used with other
Automatically detecting atomicity violations using atomic blocks [50]	static	3	Java language
Bauhaus [31]	static	*1	C language
DR-Frame [32]	static	*1	Programs compiled by clang compiler
RELAY [29]	static	*1	Linux kernel
Communicating Sequential Processes semantics for activity diagrams [33]	static	2	Applications from cloud computing domain
DeadLock Analysis Models [36]	static	2	**Java and Java-like languages
SWORD [78]	static	1	Java programs written in Eclipse IDE
Concurrent CFG [79]	static	*1	C language
Chord [81]	static	1	Java language
LOCKSMITH [82]	static	*1	C language
Atomizer [74]	mixed	3	Java language
ThreadSanitizer [72]	mixed	*1, 3, 4	C++ language
SVD [45]	mixed	1	**C language
Have [51]	mixed	1, 3	Java language
ConFuzz [55]	mixed	*1, 3, 4	Programs compiled compiler from the LLVM project
HistLock+ [68]	mixed	*1	C and C++ languages
Chang et al. [48]	mixed	3	Event-driven Node.js applications
Prevention			
Gadara [21]	dynamic	2	**Deadlock detection general model
Grace [57]	dynamic	1, 2, 3	C and C++ applications for Linux based operating system
Convoider [54], [56]	dynamic	*1, 2, 3	C and C++ applications for Linux based operating system
Automatic bug fixing for both deadlock and livelock [22]	static	2	C programs using POSIX threads
Kivati [40]	mixed	3	C programs for Linux/x86 platforms
SDRacer [8]	mixed	1	C programs for uLinux

1 race condition, 2 deadlock, 3 atomicity violation, 4 order violation
* The authors of the methods use the general term *data race*, which can mean either a single error or a whole category of errors.
** Intended use is not directly expressed in cited paper.

- the requirement to learn new, unpopular programming languages, which increases the costs of using the method: [33], [74], [83].
- the ambiguity in the transformation of the program code to other formal representations, e.g. multiple PN structures correspond to one code fragment: [31], [51], [73].

Most of the methods do not meet at least one of the requirements, and the use of several methods simultaneously (to ensure the detection of four errors) increases the cost of use.

None of the methods or tools described in the previous subsection can be used to verify the results of the *rdao detector* tool described later. Methods and tools for errors prevention do not inform about places where errors occur. Part of detection methods have to be excluded because they cannot be used with C language or POSIX Thread library. Also, some part of methods and tools are abandoned because of low quality, some part is no available anymore and some part is on the early phase of development. In other words, none of tools and methods described meet the requirements to make a comparison with *rdao detector*.

III. MASCM MODEL

A. MOTIVATION

The conclusion from the previous section prompted the authors of this paper to search for new solutions enabling the identification / detection of a given class of errors (race-condition, deadlocks, atomicity violation, and order violation) in multithreaded applications. The methods based on dynamic software analysis are limited to selected operating systems and / or programming language cases. This disadvantage is devoid of methods based on static software analysis, which, in turn, are time-ineffective (in many cases code analysis requires a complete review). In addition, the analysis of the source code equivalent expressed using known representations, e.g. CFG or Petri net does not provide information about many characteristics important for detecting errors in a multithreaded application (e.g. execution time intervals of individual threads [11]). Knowing the characteristics of various types of software analysis, work was undertaken to develop a new model dedicated to multithreaded applications. During the development, key assumptions were made for the model to have the following characteristics:

- explicitness of the code to model transformation,
- emphasizing the elements that are important for identifying a given error class,
- simplification of selected code elements,
- applicable to different programming languages.

Previous research shows [11], [19], [38], [39] that the implementation of these characteristics will significantly improve the detection of errors in multithreaded applications while reducing the time of analysis. Additionally, static analysis of the code using the proposed model may be one of the stages of the hybrid method, in which other solutions will also be used.

It is important to support C programmers who write a lot of C applications for medical, automotive, or energy industries. The huge number of devices used there do not contains GPU's but do contain CPU's with multithreading support. A lot of software cannot be tested on the target devices, so programmers use emulators and static analysis tools to improve the quality of software. The testing process can be long and expensive even with emulators, so to reduce costs programmers use static analysis tools to find bugs in a short time during development. As an example for QNX operating system software is mostly create on Linux and cross compiled for target devices. QNX is used on embedded devices where there are no GPU's but CPU's supported multithreading.

B. FORMAL DESCRIPTION OF THE MODEL

The structure of the model allows for the mapping of a multithreaded computer program in the form of a tuple:

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

where:

1) P is the program index,
2) $T_P = \{t_i | i = 0 \ldots \alpha\}$, $(\alpha \in \mathbb{N})$ is the set of all threads t_i of C_P, where t_0 is the main thread, $|T_P| > 1$,
3) $U_P = (u_b | b = 1 \ldots \beta)$, $(\beta \in \mathbb{N}^+)$ is a sequence of sets $u_b \subseteq T_P$ containing threads working in the same time frame in the program C_P, whereas $\beta > 1$, $u_1 = \{t_0\}$ and $u_\beta = \{t_0\}$,
4) $R_P = \{r_c | c = 1 \ldots \gamma\}$, $(\gamma \in \mathbb{N}^+)$ is a family of shared resources $r_c = \{v_1, v_2, \ldots, v_\eta\}$ of the program C_P, where "v" are the names of variables,
5) $O_P = \{o_{i,j} | i = 1 \ldots \delta, j = 1 \ldots \epsilon\}$, $(\delta, \epsilon, \in \mathbb{N}^+)$ is a set of all atomic operations of the program C_P. An atomic operation is an instruction or function defined in a programming language that cannot be divided. The i index indicates the number of the thread in which the operation is executed and the j index is the order number of operations working within the thread t_i,
6) $Q_P = \{q_s | s = 1 \ldots \kappa\}$, $(\kappa \in \mathbb{N}^+)$ - the set of mutexes $q_s = (w_s, x_s)$, available in the program, defined as the pair *variable, mutex type*, where the type is one of the set values (PMN, PME, PMR, PMD), where the values correspond to the mutex types from the library *pthread*,
7) $F_P = \{f_n | n = 1 \ldots \iota\}$ and $F_P \subseteq (O_P \times O_P) \cup (O_P \times R_P) \cup (R_P \times O_P) \cup (O_P \times Q_P) \cup (Q_P \times O_P)$, $(\iota \in \mathbb{N}^+)$ - the set of edges:
   a) transition edges defining the order of operations: $f_n = (o_{i,j}, o_{i,k})$, where the operation $o_{i,j}$ is performed prior to the operation $o_{i,k}$, $o_{i,j}, o_{i,k} \in O_P$,
   b) resource-indicating edges r_c, that change during the operation $o_{i,j}$: $f_n = (o_{i,j}, r_c)$, $o_{i,j} \in O_P$, $r_c \in R_P$,
   c) relationship edges indicating operations $o_{i,j}$ depending on the current value of one of the resources r_c: $f_n = (r_c, o_{i,j})$, $r_c \in R_P$, $o_{i,j} \in O_P$,
   d) the edges of the mutex creation indicating the operation $o_{i,j}$ creation selected mutex q_s: $f_n = (q_s, o_{i,j})$, $q_s \in Q_P$, $o_{i,j} \in O_P$,
   e) mutex release edges indicating the operation $o_{i,j}$ releasing selected mutex q_s: $f_n = (o_{i,j}, q_s)$, $q_s \in Q_P$, $o_{i,j} \in O_P$,
8) $B_P = (B_P^{FWD}, B_P^{BWD}, B_P^{SYM})$ - sequence of sets:
   - B_P^{FWD} set of pairs of operations related by a forward relationship: $B_P^{FWD} = \{(o_{i,j}, o_{a,b}); o_{i,j}, o_{a,b} \in O_P\}$; operation execution $o_{i,j}$ forces the execution of the operation $o_{a,b}$, while $o_{a,b}$ of operation is not conditioned by the earlier implementation of the operation $o_{i,j}$. Later in the work, this relation will be marked with the symbol $o_{i,j} \rightarrow o_{a,b}$;
   - B_P^{BWD} set of pairs of operations related by a forward relationship: $B_P^{BWD} = \{(o_{i,j}, o_{a,b}); o_{i,j}, o_{a,b} \in O_P\}$; operation execution $o_{i,j}$ from the pair does not force the operation $o_{a,b}$, while the execution of the operation $o_{a,b}$ is conditioned by the earlier execution of the operation $o_{i,j}$. Later in the work, this relation will be marked with the symbol $o_{i,j} \leftarrow o_{a,b}$;
   - B_P^{SYM} a set of pairs of operations linked by a symmetrical relation: $B_P^{SYM} = \{(o_{i,j}, o_{a,b}); o_{i,j}, o_{a,b} \in O_P\}$; execution of the operation $o_{i,j}$ forces the execution of $o_{a,b}$ and conversely, the execution of $o_{a,b}$ requires prior execution of $o_{i,j}$. Later in the work, this relation will be marked with the symbol $o_{i,j} \leftrightarrow o_{a,b}$.

C. MODEL INSTANCE BUILDING METHOD

The source code model for multithreaded applications was developed for the C language and pthreads library (the official implementation of the POSIX Threads standard for C language). An example of an application

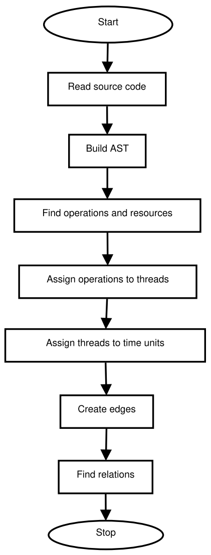
Start
Read source code
Build AST
Find operations and resources
Assign operations to threads
Assign threads to time units
Create edges
Find relations
Stop

FIGURE 3. The algorithm for building a model instance by the application mascm_generator.

mascm_generator transforming the source code of any C multithreaded application to be represented according to the developed C_P (1) can be found in the repository *rdao detector* (https://github.com/PKPhdDG/rdao_detector).

It was assumed that the C_P (1) representation is built indirectly from AST syntax trees generated by GCC compiler. The advantage of using syntax trees is their widespread use in compilers and parsers, which potentially enables easier integration of the developed tool with existing compilers. The disadvantages include the fact that two different compilers can generate different member trees for the same source code, which can result in keeping two different C_P representations. The idea of *mascm_generator* application is illustrated in Fig. 3. Determining the C_P representation for a given P application requires 6 stages:

1) *Build AST* - build AST from the source code and validate the code performed by the compiler.
2) *Find operations and resources* - analysis of AST nodes to extract operations and resources.
3) *Assign operations to threads* - assign the operations you have previously extracted to the corresponding threads.
4) *Assign thread to time units* - create time intervals based on AST nodes and add threads to those time intervals.
5) *Create edges* - analyze a set of operations and create relations/edges of the model.
6) *Find relations* - analyze the operation set and AST tree to detect related operation pairs.

The most difficult stage of determining the C_P representation is stage no. 4. To detect every time unit source code has to be analyzed to find every call of functions pthread_create and pthread_join. Calling the first one is equivalent to creating a new time unit, calling the second one with the end of the time unit. When few calls of the pthread_create follow one another then all of these calls are treaded as one. The same rule applies to the pthread_join function. If between two calls of functions pthread_create or pthread_join exist call of other function then a new time unit is created for this function.

D. SAMPLE MODEL INSTANCES

The process shown in Fig. 3 was used to derive the representation of C_{EG1} (1) of the following program EG1:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *thread_start(void *args)
{
    int *i = (int*)args;  // o1,1
    *i = 100;  // o1,2
    return (void*)NULL; // o1,3
}


int main()
{
    int i = 0; // o0,1
    pthread_t thandler; // o0,2
    pthread_create(&thandler, NULL, thread_start,
        (void*)&i); // o0,3
    pthread_join(thandler, NULL); // o0,4
    printf(``I value is equal %d'', i); // o0,5
    return 0; // o0,6
}
```

The collection of EG1 threads includes main thread t_0 and thread t_1 created with *pthreads*. All operations of the *main* function belong to the t_0 thread, and operations in the *thread_start* function belong to the t_1 thread. The shared resource set includes the i variable, to which the indicator is passed as a parameter to the t_1 thread function. The resource set is therefore $R = \{r_1 = \{args, i\}\}$. Projection operations, address download, and extraction operations are skipped, for example, "$(void*)\&and$". There are three time intervals for thread execution. The thread t_0 is executed in the first u_1 and the last u_3 of the time interval, while the thread t_1 is executed in the interval u_2. The sequence of time intervals is as follows $U = (\{t_0\}, \{t_1\}, \{t_0\})$.

The set of edges includes transition edges that indicate the order of operations and usage edges that indicate resource utilization. There are no mutexes in the application, so Q is an empty set, just like B, because there are no relations between operations in the application.

Ultimately, the C_{EG1} model for the $EG1$ application looks as follows:

$$\begin{aligned}
T_{EG1} &= \{t_0, t_1\}\\
U_{EG1} &= (\{t_0\}, \{t_1\}, \{t_0\})\\
R_{EG1} &= \{r_1 = \{args, i\}\}\\
O_{EG1} &= \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{0,6}, o_{1,1}, o_{1,2},\\
&\qquad o_{1,3}\}\\
Q_{EG1} &= \{\emptyset\}\\
F_{EG1} &= \{(o_{0,1}, o_{0,2}), (o_{0,2}, o_{0,3}), (o_{0,3}, o_{0,4}),\\
&\qquad (o_{0,4}, o_{0,5}), (r_1, o_{0,5}), (o_{0,5}, o_{0,6}), (o_{1,1}, o_{1,2}),\\
&\qquad (o_{1,2}, r_1), (o_{1,2}, o_{1,3})\}\\
B_{EG1} &= \{\emptyset\}
\end{aligned}$$

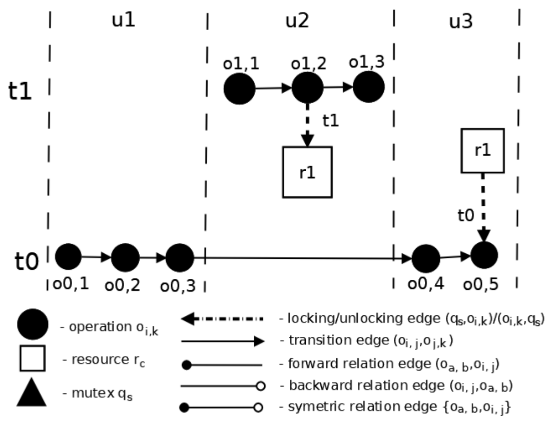
u1
u2
u3
t1
t0
o1,1
o1,2
o1,3
o0,1
o0,2
o0,3
o0,4
o0,5
r1
- operation $o_{i,k}$
- resource r_c
- mutex q_s
- locking/unlocking edge $(q_s,o_{i,k})/(o_{i,k},q_s)$
- transition edge $(o_{i,j},o_{j,k})$
- forward relation edge $(o_{a,b},o_{i,j})$
- backward relation edge $(o_{i,j},o_{a,b})$
- symetric relation edge $\{o_{a,b},o_{i,j}\}$

FIGURE 4. Graphic representation of the EG1 application model instance.

A graphic representation of the above C_{EG1} model is shown in Fig. 4. As it can be seen, the operations of both threads are performed in different mutually exclusive time intervals. This means that the resource r_1 is used by at most one thread at any time. The program is therefore free from errors such as race condition, deadlock, atomicity violation, order violation. The presented model (as well as its graphic representation) allows for the development of conditions meeting which allows for the detection of the considered errors. The construction of such conditions is the subject of the next chapter.

IV. ERROR DETECTION

A. TEST APPLICATION DESCRIPTION

For the purposes of this article, the homesoftserver application (https://github.com/PKPhdDG/homesoftserver) was developed, enabling the collection of data from IoT devices such as cameras, motion sensors, water sensors, smoke detectors, and heat sensors. It is a multithreaded application in which one of the modules simulates the work of a device receiving commands from an external environment, then executed in separate threads. Depending on the command, calculation results are saved to shared resources. Delegating work to separate threads allows the listening thread to continuously accept new commands. In addition to the task threads and the command listening thread, there is also a log handling thread that places the application logs in a file that other threads can also access by properly designed API. The homesoftserver application has been designed so that its architecture corresponds to the architecture of applications such as Apache or Nginx. The application repository on the GitHub portal, in the main branch, contains the application code free from the errors discussed below. These errors can be put into the application by applying patches in the bug_patches directory.

B. RACE CONDITION

A race condition error occurs when several threads concurrently access and act on the same resources so that the result depends on the order in which the resources were accessed [18]. In general, the race condition detection problem can be formulated as follows:

Problem 1: There is a P multithreaded application whose code is written in C using the pthreads library. I am looking for an answer to the question: Is there a race condition error in the P application?

In the context of the introduced C_P model, the answer to the above question comes down to the search for conditions, the fulfillment of which will enable the identification of operations (and related threads), the implementation of which may cause a race condition.

An example of a very simple application with a race condition (RC1) is available at https://bit.ly/2YKk4vr. The presented application is a textbook example in which two threads working in parallel change the value of a global variable in an uncontrolled manner. The C_P representation for the RC1 application is as follows:

$$\begin{aligned}
T_{RC1} &= \{t_0, t_1, t_2\}\\
U_{RC1} &= (\{t_0\}, \{t_1, t_2\}, \{t_0\})\\
R_{RC1} &= \{\{r1\}\}\\
O_{RC1} &= \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{0,6}, o_{0,7}, o_{0,8},\\
&\quad o_{0,9}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{1,5}, o_{1,6}, o_{2,1}, o_{2,2}, o_{2,3},\\
&\quad o_{2,4}, o_{2,5}, o_{2,6}\}\\
Q_{RC1} &= \{\emptyset\}\\
F_{RC1} &= \{(o_{0,1}, o_{0,2}), (o_{0,2}, o_{0,3}), (r_1, o_{0,3}),\\
&\quad (o_{0,3}, o_{0,4}), (o_{0,4}, o_{0,5}), (o_{0,5}, o_{0,6}), (o_{0,6}, o_{0,7}),\\
&\quad (o_{0,7}, o_{0,8}), (r_1, o_{0,8}), (o_{0,8}, o_{0,9}), (o_{1,1}, o_{1,2}),\\
&\quad (o_{1,2}, o_{1,6}), (o_{1,2}, o_{1,3}), (o_{1,3}, o_{1,4}), (o_{1,4}, r_1),\\
&\quad (o_{1,4}, o_{1,5}), (o_{1,5}, o_{1,2}), (o_{1,5}, o_{1,6}), (o_{2,1}, o_{2,2}),\\
&\quad (o_{2,2}, o_{2,6}), (o_{2,2}, o_{2,3}), (o_{2,3}, o_{2,4}), (o_{2,4}, r_1),\\
&\quad (o_{2,4}, o_{2,5}), (o_{2,5}, o_{2,2}), (o_{2,5}, o_{2,6})\}\\
B_{RC1} &= \{\emptyset\}
\end{aligned}$$

It is worth noting that the race condition can be caused by operations of threads working in a common time interval - in the RC1 application, it is the u_2 range, containing the threads: t_1 and t_2. Within the thread t_1, the operation $o_{1,4}$ is performed to change the shared resource r_1. Similarly, in the thread t_2 the resource r_1 is changed by the operation $o_{2,4}$. None of these operations is preceded by the operation of creating the q_w mutex, ensuring their mutual exclusion when accessing the r_1 resource. This means that both operations are not in critical sections which leads to a race condition. In the graphic representation of C_P, this is manifested by the acyclic nature of threads using a common resource.

This observation leads to the claim, the definition of which requires entering the concept of thread path t_i.

Definition 1: Let $\lambda^{P,i}$ be the set of all paths that can be created from the operations and mutexes used by the t_i thread starting from the $o_{i,1}$ operation. Formally, the a-th path of $\lambda^{P,i}$

is defined as a sequence:

$$\lambda_a^{P,i} = (\lambda_{a,1}^{P,i}, \lambda_{a,2}^{P,i}, \ldots, \lambda_{a,b}^{P,i}, \ldots, \lambda_{a,q}^{P,i}),$$
where
$$\lambda_{a,1}^{P,i} = o_{i,1}, \lambda_{a,b}^{P,i} \in O_P \cup Q_P,$$
$$(\lambda_{a,k}^{P,i}, \lambda_{a,k+1}^{P,i}) \in F_P, \lambda_{a,k}^{P,i} \neq \lambda_{a,l}^{P,i} \text{ for } k \neq l,\ l,k = 1 \ldots q$$

In this section, the following form was adopted:

- $\lambda_{a,b}^{P,i} \lhd \lambda_a^{P,i}$ if the element $\lambda_{a,b}^{P,i}$ is part of the path $\lambda_a^{P,i}$
- $\lambda_{a,b}^{P,i} \ntriangleleft \lambda_a^{P,i}$ if the element $\lambda_{a,b}^{P,i}$ is not part of the path $\lambda_a^{P,i}$.

In addition, the path $\lambda_a^{P,i}$ is assumed to be cyclical if the condition: $\exists_{\lambda_{a,b}^{P,i} \lhd \lambda_a^{P,i}} \lambda_{a,b}^{P,i} = \lambda_{a,q}^{P,i}, b \neq q$. The set of cyclic paths of the t_i thread is denoted by $C\lambda^{P,i} \subseteq \lambda^{P,i}$

Theorem 1: Let $O_P = \{o_{m,j}, \ldots, o_{i,j}, \ldots, o_{a,b}\}$ denote the set of operations performed within the threads of the set $u_b \in U_P$, which use a common resource: $r_c \in R_P$ (i.e. there are $(o_{m,j}, r_c) \in F_P$ or $(r_c, o_{m,j}) \in F_P$, ..., or $(o_{a,b}, r_c) \in F_P$ or $(r_c, o_{a,b}) \in F_P$).

If there is an operation $o_{i,j} \in O_P$ using the resource r_c that:

- is not an element of the cyclic path $\lambda_a^{P,i} \in C\lambda^{P,i}$ (i.e. $o_{i,j} \ntriangleleft \lambda_a^{P,i}$) or
- is an element of the cyclic path $\lambda_a^{P,i} \in C\lambda^{P,i}$ (i.e. $o_{i,j} \lhd \lambda_a^{P,i}$) but not preceded by the mutex locking operation q_n

the operation causes race condition.

Proof: The proof of the theorem follows directly from the definition of race condition. Operation access $O_P = \{o_{m,j}, \ldots, o_{i,j}, \ldots, o_{a,b}\}$ to resource r_c inR_P in the mutual exclusion mode is conditional on their implementation within the critical section of the previously assumed (for each operation) q_n mutex. The operations of setting and releasing the q_n mutex always make up a cyclic path: $\lambda_a^{P,i} = (\ldots, q_n, \ldots, o_{i,j}, \ldots, q_n) \in C\lambda^{P,i}$, where q_n precedes $o_{i,j}$. The lack of such paths in the $C\lambda^{P,i}$ set means the existence of the $o_{i,j}$ operation which has "free" access to the r_c resource, which leads to a race condition.

c.k.d.

The listing below shows the **homesoftserver** application code with another example of race condition. This fragment is part of the mechanism responsible for recording events from the application (so-called logs). The *enable_logger* function runs in a separate thread and causes an asynchronous display of logs downloaded from the buffer using the *add_log* function. The operation of the logging mechanism in the *enable_logger* function is secured by the *log_m* and *buf_index_m* mutexes. The first one protects the *logger.buffer* component, while the second one protects the *logger.index* component. However, in the *add_log* function, none of the operations involving the *logger.buffer* and *logger.index* variables were secured with a mutex, which leads to race condition.

```
void* enable_logger(void *args)
{
    ...
            pthread_mutex_lock(&log_m);
            pthread_mutex_lock(&buf_index_m);
            for (int i = 0; i <= logger.index; ++i
                )
            {
                if (logger.buffer[i] != (char*)
                    NULL)
                {
                    const size_t log_lvl_str_index
                        = (size_t)logger.
                        log_levels[i] - 1;
                    printf(``LOG:%s:%s\n'',
                        log_lvl_str[
                        log_lvl_str_index], logger
                        .buffer[i]);
                    free(logger.buffer[i]);
                    logger.buffer[i] = (char*)NULL
                        ;
                }
            }
            logger.index = -1;
            pthread_mutex_unlock(&buf_index_m);
            pthread_mutex_unlock(&log_m);
    ...
}

void add_log(const char *log, enum LogLevel level)
{
    char *llog = (char*)malloc(sizeof(char)*strlen
        (log));
    strcpy(llog, log);
    if (logger.index >= BUFFER_SIZE || log_level >
        level) return;

    ++logger.index;
    logger.buffer[logger.index % BUFFER_SIZE] =
        llog;
    logger.log_levels[logger.index % BUFFER_SIZE]
        = level;
    logger.index = logger.index % BUFFER_SIZE;
}
```

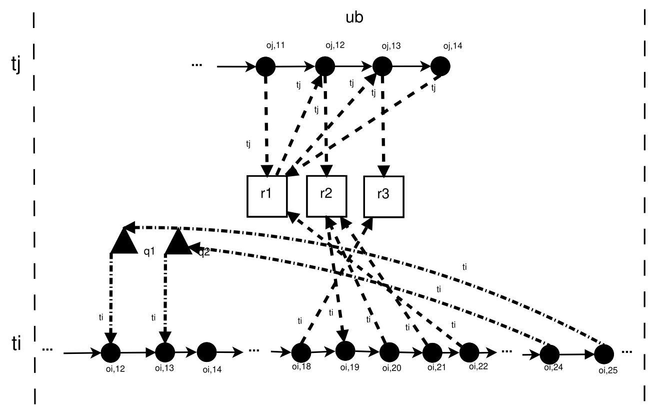

FIGURE 5. Fragment of the homesoftserver application model instance with race condition.

The picture no. 5 includes a representation of C_P of the application **homesoftserver** in a graphic form that corresponds to the presented fragment of the program code. Operations from the t_j thread correspond to the code fragment from the *add_log* function, while operations from the t_i thread correspond to the *enable_logger* function. According to the adopted notation, the C_P representation has the following form:

$$T_{HSSRC} = \{t_0, \ldots, t_i, t_j, \ldots\}$$
$$U_{HSSRC} = (\{t_0\}, \ldots, \{t_i, t_j, \ldots\}, \ldots, \{t_0\})$$
$$R_{HSSRC} = \{\{logger.index\}, \{logger.buffer\},$$

$$
\begin{aligned}
& \{logger.levels\}, \ldots\} \\
O_{HSSRC} &= \{o_{i,12}, o_{i,13}, o_{i,14}, \ldots, o_{i,18}, o_{i,19}, \\
& o_{i,20}, o_{i,21}, o_{i,22}, \ldots, o_{i,24}, o_{i,25}, \ldots, o_{j,11}, o_{j,12}, \\
& o_{j,13}, o_{j,14}, \ldots\} \\
Q_{HSSRC} &= \{(log\_m, PMD), \\
& (buf\_index\_m, PMD), \ldots\} \\
F_{HSSRC} &= \{\ldots, (q_1, o_{i,12}), (o_{i,12}, o_{i,13}), \\
& (q_2, o_{i,13}), (o_{i,13}, o_{i,14}), \ldots, (o_{i,18}, r_3), (o_{i,18}, o_{i,19}), \\
& (r_2, o_{i,19}), (o_{i,19}, o_{i,20}), (o_{i,20}, r_2), (o_{i,20}, o_{i,21}), \\
& (o_{i,21}, r_2), (o_{i,21}, o_{i,22}), (o_{i,22}, r_1), \ldots, (o_{i,24}, q_2), \\
& (o_{i,24}, o_{i,25}), (o_{i,25}, q_1), \ldots, (o_{j,11}, r_1), (o_{j,11}, o_{j,12}), \\
& (r_1, o_{j,12}), (o_{j,12}, r_2), (o_{j,12}, o_{j,13}), (o_{j,13}, r_3), \\
& (o_{j,13}, o_{j,14}), (o_{j,14}, r_1), \ldots\} \\
B_{HSSRC} &= \{\ldots\}
\end{aligned}
$$

As it is easy to notice, in the program under consideration there are no operations assuming mutexes in the thread t_j. This means that the operations of the t_j thread have free access to the resources r_1, r_2, r_3. According to the introduced theorem 1, the highlighted operations of the t_j thread are not an element of cyclic paths (containing mutex creation and release operations). Therefore, the application will have a race condition error.

In order to eliminate the detected race condition, the code should be supplemented with operations that create and release mutexes. One of the popular practices to avoid race condition is to assign each of the shared resources one mutex, whose task will be to watch over all operations performed on that resource. It is possible for the application code to have two or more resources that are closely related and changing one of them changes the others. For groups of closely related resources, a single mutex is used instead of multiple mutexes. However, the improper use of mutexes can lead to a deadlock or atomicity violation.

C. DEADLOCK

The first error that may occur as a result of the incorrect operation of mechanisms aimed at eliminating the race condition error is deadlock. Deadlock has been defined as a situation where several threads wait indefinitely for an event that can only be started by one of the waiting threads [18]. This error is strongly related to mutex creation and release operations, which can be one of four types:

- PTHREAD_MUTEX_NORMAL (PMN)
- PTHREAD_MUTEX_ERRORCHECK (PME)
- PTHREAD_MUTEX_RECURSIVE (PMR)
- PTHREAD_MUTEX_DEFAULT (PMD)

For a deadlock to occur, four conditions must be met [18], [58]:

- mutual exclusion,
- holding and waiting,
- no expropriation,
- cyclical waiting.

The above conditions are met in the following scenarios [19]:

- mutex pairs mutually exclusive (scenario S1),
- no operation of removing the mutex as a result of control instructions (scenario S2),
- re-calling the mutex of the loop result (scenario S3),
- re-calling of the creation of a mutex of a non-PMR type as a result of calling a recurrent function (scenario S4).

The problem of locating deadlocks in multithreaded applications can be defined as follows.

Problem 2: There is a P multithreaded application written in C using the pthreads library. We are looking for an answer to the question: Is there a deadlock in the P application?

In the context of the introduced C_P model, the answer to the above question comes down to determining the conditions enabling the identification of program elements responsible for the occurrence of one of the above-defined S1-S4 scenarios.

Mutually exclusive pairs of mutexes (S1). For example, code for an application with a deadlock error due to mutually exclusive pairs of mutexes can be found at https://bit.ly/2D7ANBk (DL0). The C_P representation for a DL0 application is as follows:

$$
\begin{aligned}
T_{DL0} &= \{t_0, t_1, t_2\} \\
U_{DL0} &= (\{t_0\}, \{t_1, t_2\}, \{t_0\}) \\
R_{DL0} &= \{\{counter\}\} \\
O_{DL0} &= \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{0,6}, o_{0,7}, o_{0,8}, \\
& o_{0,9}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{1,5}, o_{1,6}, o_{2,1}, o_{2,2}, o_{2,3}, \\
& o_{2,4}, o_{2,5}, o_{2,6}\} \\
Q_{DL0} &= \{(m, PMN), (n, PMN)\} \\
F_{DL0} &= \{(o_{0,1}, o_{0,2}), (o_{0,2}, o_{0,3}), (r_1, o_{0,3}), \\
& (o_{0,3}, o_{0,4}), (o_{0,4}, o_{0,5}), (o_{0,5}, o_{0,6}), (o_{0,6}, o_{0,7}), \\
& (o_{0,7}, o_{0,8}), (r_1, o_{0,8}), (o_{0,8}, o_{0,9}), (q_1, o_{1,1}), \\
& (o_{1,1}, o_{1,2}), (q_2, o_{1,2}), (o_{1,2}, o_{1,3}), (o_{1,3}, r_1), \\
& (o_{1,3}, o_{1,4}), (o_{1,4}, q_2), (o_{1,4}, o_{1,5}), (o_{1,5}, q_1), \\
& (o_{1,5}, o_{1,6}), (q_2, o_{2,1}), (o_{2,1}, o_{2,2}), (q_1, o_{2,2}), \\
& (o_{2,2}, o_{2,3}), (o_{2,3}, r_1), (o_{2,3}, o_{2,4}), (o_{2,4}, q_1), \\
& (o_{2,4}, o_{2,5}), (o_{2,5}, q_2), (o_{2,5}, o_{2,6})\} \\
B_{DL0} &= \{\emptyset\}
\end{aligned}
$$

From the above representation it is easy to read that in the thread t_1 mutexes are created in the order (q_1, q_2), the reverse of that in the thread t_2: (q_2, q_1). Both these threads run in the same u_2 time frame. The reverse order of the mutex setting causes the threads to wait for each other - a deadlock error occurs.

This observation was used to develop a condition enabling the identification of deadlock S1 errors in the program code.

Definition 2: Let the path $\lambda_l^{P,i}$ have two such operations: $o_{i,a} \lhd \lambda_l^{P,i}$ and $o_{i,b} \lhd \lambda_l^{P,i}$ within which the q_c and q_d mutexes are created respectively (i.e. in F_P there are $(o_{i,a}, q_c)$ and $(o_{i,b}, q_d)$) edges.

The q_c mutex is said to precede the q_d mutex on $\lambda_l^{P,i}$ if $o_{i,a}$ precedes $o_{i,b}$ ($o_{i,a} \prec o_{i,b}$). This relationship is denoted by $q_c \prec_i^l q_d$.

The above definition leads to the following lemma:

Lemma 1: Two threads t_i and t_j of application P are given, and the sets of paths $\lambda^{P,i}$ are known. In both threads there are operations setting up q_c and q_d mutexes.

If there is such a pair of paths: $\lambda_l^{P,i}, \lambda_k^{P,i}$ for which the condition is met: $(q_c \prec_i^l q_d) \bigwedge (q_d \prec_j^k q_c)$ or $(q_c \prec_j^k q_d) \bigwedge (q_d \prec_i^l q_c)$ then in the source code of the P application there will be a deadlock (S1) with threads t_i and t_j.

Proof: Based on the instance of the DL0 application model, the necessary condition for a deadlock error of type S1 (mutex pair exclusion) between two threads t_i and t_j there are sequences of mutex setting up q_c and q_d in reverse order each of these threads. This condition is therefore fulfilled when in the thread t_i the q_c mutex precedes the q_b mutex (according to def. 2) and the thread t_j the q_b mutex precedes the q_c mutex $(q_c \prec_i^l q_d) \bigwedge (q_d \prec_j^k q_c)$ or reversely $(q_c \prec_j^k q_d) \bigwedge (q_d \prec_i^l q_c)$. c.k.d.

No mutex release operation (S2) / re-creation of mutex operation (S3). For a sample application code with a deadlock based on S2 scenario, see https://bit.ly/32tiBKY (DL1). The C_P representation for a DL1 application is as follows:

$$
\begin{aligned}
T_{DL1} &= \{t_0, t_1, t_2\}\\
U_{DL1} &= (\{t_0\}, \{t_1, t_2\}, \{t_0\})\\
R_{DL1} &= \{\{counter\}, \{args, i, t2arg\}\}\\
O_{DL1} &= \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{0,6}, o_{0,7}, o_{0,8},\\
&\quad o_{0,9}, o_{0,10}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{2,1}, o_{2,2}, o_{2,3}, o_{2,4},\\
&\quad o_{2,5}, o_{2,6}, o_{2,7}, o_{2,8}, o_{2,9}\}\\
Q_{DL1} &= \{(m, PMN)\}\\
F_{DL1} &= \{(o_{0,1}, o_{0,2}), (o_{0,2}, o_{0,3}), (r_1, o_{0,3}),\\
&\quad (o_{0,3}, o_{0,4}), (o_{0,4}, o_{0,5}), (o_{0,5}, o_{0,6}), (o_{0,6}, o_{0,7}),\\
&\quad (o_{0,7}, o_{0,8}), (o_{0,8}, o_{0,9}), (r_1, o_{0,9}), (o_{0,9}, o_{0,10}),\\
&\quad (q_1, o_{1,1}), (o_{1,1}, o_{1,2}), (o_{1,2}, r_1), (o_{1,2}, o_{1,3}),\\
&\quad (o_{1,3}, q_1), (o_{1,3}, o_{1,4}), (q_1, o_{2,1}), (o_{2,1}, o_{2,2}),\\
&\quad (o_{2,2}, o_{2,3}), (r_2, o_{2,3}), (o_{2,3}, o_{2,4}), (o_{2,4}, o_{2,7}),\\
&\quad (o_{2,4}, o_{2,5}), (o_{2,5,r_1}), (o_{2,5}, o_{2,6}), (o_{2,6}, q_1),\\
&\quad (o_{2,6}, o_{2,9}), (o_{2,7}, o_{2,8}), (o_{2,8}, r_1)\}\\
B_{DL1} &= \{\emptyset\}
\end{aligned}
$$

Sample application code with S3 deadlock can be found at https://bit.ly/3lAFAwL (DL2). The C_P representation for DL1 application is as follows:

$$
\begin{aligned}
T_{DL2} &= \{t_0, t_1\}\\
U_{DL2} &= (\{t_0\}, \{t_1\}, \{t_0\})\\
R_{DL2} &= \{\{counter\}, \{args, i, t1arg\}\}\\
O_{DL2} &= \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{0,6}, o_{0,7}, o_{1,1},\\
&\quad o_{1,2}, o_{1,3}, o_{1,4}, o_{1,5}, o_{1,6}\}\\
Q_{DL2} &= \{(m, PMN)\}\\
F_{DL2} &= \{(o_{0,1}, o_{0,2}), (r_1, o_{0,2}), (o_{0,2}, o_{0,3}),\\
&\quad (o_{0,3}, o_{0,4}), (o_{0,4}, o_{0,5}), (o_{0,5}, o_{0,6}), (r_1, o_{0,6}),\\
&\quad (o_{0,6}, o_{0,7}), (o_{1,1}, o_{1,2}), (o_{1,2}, o_{1,5}), (o_{1,2}, o_{1,3}),\\
&\quad (q_1, o_{1,3}), (o_{1,3}, o_{1,4}), (o_{1,4}, o_{1,2}), (o_{1,4}, r_1),\\
&\quad (o_{1,4}, o_{1,5}), (o_{1,5}, q_1), (o_{1,5}, o_{1,6})\}\\
B_{DL2} &= \{\emptyset\}
\end{aligned}
$$

When analyzing variants of DL1 application execution, it is easy to find a scenario in which the operation (starting from the $(q_1, o_{2,1})$ edge and then going through the $(o_{2,4}, o_{2,7})$ edge) omits the $o_{2,6}$ operation responsible for releasing the previously created q_1 mutex - a deadlock (S2) occurs. A similar situation is present in the DL2 application. A scenario is acceptable where the execution of the operation (from the $(q_1, o_{1,3})$ edge, then through the $(o_{1,3}, o_{1,4}), (o_{1,4}, o_{1,2})$ edge to the $(o_{1,2}, o_{1,3})$ edge), leads to the operation of re-setting up the q_1 mutex - a deadlock (S3) occurs. The above observations were used to develop a condition to identify S2 and S3 deadlock errors in the program code.

Lemma 2: If in the set of $\lambda^{P,i}$ of application P there is a path that contains the mutex $q_s \lhd \lambda^{P,i}$ and it is not a cyclic path with the element $\lambda_{a,q}^{P,i} = q_s$, then an error occurs in the application P of deadlock class, type S2 and S3.

Proof: The necessary condition for an error of the S2 and S3 deadlock class in the thread t_i is the existence of a sequence of operations which result in the lack of release of the previously created mutex q_s. This condition is therefore fulfilled when there is a path in the $\lambda^{P,i}$ of application P that starts with q_s (mutex setting) and does not return to q_s (mutex release) - a path that is not cycle. c.k.d.

Re-calling the operation of setting up a mutex type other than PMR (S4).

For an example of an application with an S4-compatible deadlock, see https://bit.ly/34J63lu (DL3). The C_P representation for a DL3 application is as follows:

$$
\begin{aligned}
T_{DL3} &= \{t_0, t_1\}\\
U_{DL3} &= (\{t_0\}, \{t_1\}, \{t_0\})\\
R_{DL3} &= \{\{counter\}, \{args, i, t1arg\}\}\\
O_{DL3} &= \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{0,6}, o_{0,7}, o_{1,1},\\
&\quad o_{1,2}, o_{1,3}, o_{1,4}, o_{1,5}, o_{1,6}, o_{1,7}, o_{1,8}, o_{1,9}, o_{1,10}, o_{1,11},\\
&\quad o_{1,12}, o_{1,13}, o_{1,14}, o_{1,15}, o_{1,16}, o_{1,17}, o_{1,18}\}\\
Q_{DL3} &= \{(m, PMN)\}\\
F_{DL3} &= \{(o_{0,1}, o_{0,2}), (r_1, o_{0,2}), (o_{0,2}, o_{0,3}),\\
&\quad (o_{0,3}, o_{0,4}), (o_{0,4}, o_{0,5}), (o_{0,5}, o_{0,6}), (r_1, o_{0,6}),\\
&\quad (o_{0,6}, o_{0,7}), (o_{1,1}, o_{1,2}), (q_1, o_{1,2}), (o_{1,2}, o_{1,3}),\\
&\quad (o_{1,3}, o_{1,10}), (o_{1,3}, o_{1,4}), (o_{1,4}, r_2), (o_{1,4}, o_{1,5}),\\
&\quad (o_{1,5}, r_2), (o_{1,5}, o_{1,6}), (o_{1,6}, r_1), (o_{1,6}, o_{1,7}),\\
&\quad (o_{1,7}, o_{1,8}), (o_{1,8}, o_{1,9}), (q_1, o_{1,9}), (o_{1,9}, o_{1,17}),\\
&\quad (o_{1,10}, o_{1,11}), (o_{1,11}, r_2), (o_{1,11}, o_{1,12}), (o_{1,12}, r_2),\\
&\quad (o_{1,12}, o_{1,13}), (o_{1,13}, r_1), (o_{1,13}, o_{1,14}), (o_{1,14}, o_{1,15}),
\end{aligned}
$$

$$(o_{1,15,q_1}), (o_{1,15}, o_{1,16}),(o_{1,16}, o_{1,1}), (o_{1,17,q_1}),$$
$$(o_{1,17}, o_{1,18}), (o_{1,18}, o_{1,1})\}$$
$$B_{DL3} = \{\emptyset\}$$

In the adopted model, recursive functions are represented by edges, leading again to the operation calling the recursive function. In the case of DL3 applications, these are the edges of $(o_{1.16}, o_{1.1})$ and $(o_{1.18}, o_{1.1})$. Such edges appear in the model if the return keyword is used in the program code. The allowed mutex type for recursive functions is PTHREAD _MUTEX _RECURSIVE. Another type of mutex results in an S4 deadlock.

The above assumption leads to the following lemma.

Lemma 3: If the thread t_i of the P application calls a recursive function that does not use PMR mutex, a deadlock error (S4) occurs.

Proof: The lemma is a direct consequence of the assumptions made to call recursive functions.

General condition. The examples presented above allow us to make the following theorem:

Theorem 2: Let $T_P = \{t_0, \ldots, t_\alpha\}$ denote a set of threads using mutexes $Q_P = (q_1, \ldots, q_\kappa)$ of application P. The S1-S4 deadlock error will occur if there is a pair of threads t_i, t_j inT_P for which the lemma no. 1 is met or there is a thread $t_k \in T_P$ for which any of the lemmas no. 2 and 3 is met.

As mentioned earlier, deadlock errors most often occur when trying to eliminate race condition, and the case for the **homesoftserver** application is included in the listing below.

```
void* enable_logger(void *args)
{
    ...
            pthread_mutex_lock(&log_m);
            pthread_mutex_lock(&buf_index_m);
            for (int i = 0; i <= logger.index; ++i
            )
            {
                if (logger.buffer[i] != (char*)
                    NULL)
                {
                    const size_t log_lvl_str_index
                         = (size_t)logger.
                        log_levels[i] - 1;
                    printf(``LOG:%s:%s\n'',
                        log_lvl_str[
                        log_lvl_str_index], logger
                        .buffer[i]);
                    free(logger.buffer[i]);
                    logger.buffer[i] = (char*)NULL
                        ;
                }
            }
            logger.index = -1;
            pthread_mutex_unlock(&buf_index_m);
            pthread_mutex_unlock(&log_m);
    ...
}

void add_log(const char *log, enum LogLevel level)
{
    ...
    pthread_mutex_lock(&buf_index_m);
    pthread_mutex_lock(&log_m);
    ++logger.index;
    logger.buffer[logger.index % BUFFER_SIZE] =
        llog;
```

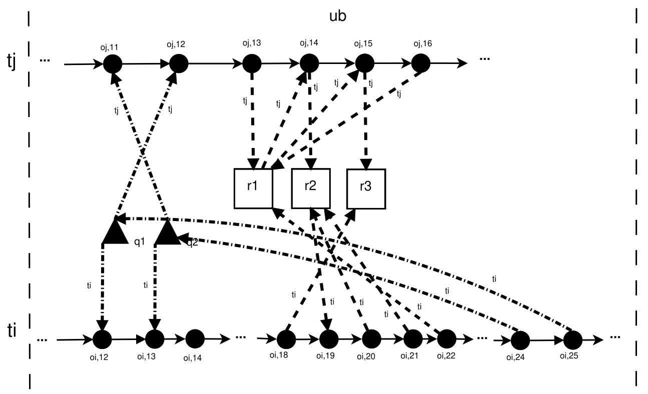

FIGURE 6. Fragment of the homesoftserver application model instance with a deadlock error.

```
    logger.log_levels[logger.index % BUFFER_SIZE]
         = level;
    logger.index = logger.index % BUFFER_SIZE;
    pthread_mutex_unlock(&buf_index_m);
    pthread_mutex_unlock(&log_m);
}
```

The following is a representation of C_P of a selected fragment of the homesoftserver application (it has a deadlock error), which is shown in Fig. no. 6.

$$T_{HSSDL} = \{t_0, \ldots, t_i, t_j, \ldots\}$$
$$U_{HSSDL} = (\{t_0\}, \ldots, \{t_i, t_j, \ldots\}, \ldots, \{t_0\})$$
$$R_{HSSDL} = \{\{logger.index\}, \{logger.buffer\}, \{logger.levels\}, \ldots\}$$
$$O_{HSSDL} = \{o_{i,12}, o_{i,13}, o_{i,14}, \ldots, o_{i,18}, o_{i,19}, o_{i,20}, o_{i,21}, o_{i,22}, \ldots, o_{i,24}, o_{i,25}, \ldots, o_{j,11}, o_{j,12}, o_{j,13}, o_{j,14}, o_{j,15}, o_{j,16}, \ldots\}$$
$$Q_{HSSDL} = \{(log\_m, PMD), (buf\_index\_m, PMD), \ldots\}$$
$$F_{HSSDL} = \{\ldots, (q_1, o_{i,12}), (o_{i,12}, o_{i,13}), (q_2,o_{i,13}),(o_{i,13},o_{i,14}), \ldots,(o_{i,18},r_3),(o_{i,18}, o_{i,19}), (r_2, o_{i,19}), (o_{i,19}, o_{i,20}), (o_{i,20}, r_2), (o_{i,20}, o_{i,21}), (o_{i,21}, r_2), (o_{i,21}, o_{i,22}), (o_{i,22}, r_1), \ldots, (o_{i,24}, q_2), (o_{i,24},o_{i,25}),(o_{i25}, q_1),\ldots,(q_2,o_{j,11}),(o_{j,11}, o_{j,12}), (q_1, o_{j,12}), (o_{j,12}, o_{j,13}), (o_{j,13}, r_1), (o_{j,13}, o_{j,14}), (r_1, o_{j,14}), (o_{j,14}, r_2), (o_{j,14}, o_{j,15}), (o_{j,15}, r_3), (o_{j,15}, o_{j,16}), (o_{j,16}, r_1), \ldots\}$$
$$B_{HSSDL} = \{\ldots\}$$

The error identification in the discussed fragment of the homesoftserver application requires checking the fulfillment of Theorem no. 2. In the considered application, Lemma no. 1 is fulfilled: the order of setting mutex pairs in the threads t_i and t_j is different. In the thread t_i mutexes are created in the order (q_1, q_2), while in the thread t_j the order is reversed, i.e. (q_2, q_1) which indicates a deadlock error (S1).

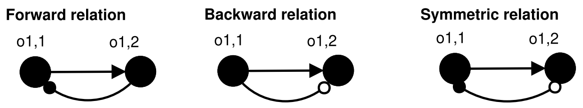

FIGURE 7. Edges reflecting the relationship between operations.

In summary, the modification of the code that introduces a fix that eliminates the race condition type error may, among other things, lead to a deadlock class error. Preventing deadlock failures is as complicated as preventing race condition failures. It should be noted that too many mutexes can lead to a significant drop in the speed of the application. When setting up groups of mutexes, one of the practices used is the use of dedicated locking and unlocking functions for entire groups of mutexes or dedicated macros.

D. ATOMICITY VIOLATION

Atomicity violation is the second error that could be introduced as a result of changes to race condition elimination. This error is the result of an inconsistent order in which the data was accessed [61]. It consists of placing linked operations (logically together) in at least two different critical sections of one thread, which allows code to be executed in parallel, which may change the state of the shared resources used in the previously mentioned critical sections. Operations can be linked with one of the following relations, which has been additionally visualized in the figure 7:

- **forward** - relation in which after the execution of the operation a, the execution of the operation b must always take place,
- **backward** - relation in which the execution of the operation a must always precede the execution of the operation b,
- **symmetric** - a relation in which the execution of the operation a must always precede the execution of the operation b, and the execution of the operation b must always follow the execution of the operation a.

In the review of the literature and technical documentation, no mechanism has been found so far in the programming languages by means of which it is possible to declare the relations presented above. The solution to this problem is to use encapsulation. Two operations that are connected to each other are closed in one function so that an operation that cannot be performed by itself is not possible to be called from outside. However, in order to properly encapsulate the operations logically connected to each other, it is necessary to anticipate this already at the application design stage. Unfortunately, it is not possible that the whole process of application design and development is supervised by a specialist in multithreading.

Taking into account the mentioned gap in the literature, it is worth providing the following definition of atomicity violation:

Definition 3: Atomicity violation is an error where between two operations of a single $o_{i,\alpha}, o_{i,\beta}$ thread using a common r_c resource, there is a sequential relationship belonging to one of the sets of sequences B_P, whose disruption caused by the $o_{j,\epsilon}$ operation of another thread on the same resource (an unexpected change of the resource) causes undefined actions of the algorithm which these operations are part of.

Taking into account the above definition, the problem of atomicity violation detection can be formulated as follows:

Problem 3: There is a multithreaded P application, written in C using the pthreads library, in which pairs of operations are known to be in one of the contractual relationships with each other. An answer is being sought to the question: Is it possible to detect an error of the atomicity violation type?

For an example of an application with an atomicity violation error, see https://bit.ly/2QBlVOC (AV1). The C_P representation for the AV1 application is as follows:

$$
\begin{aligned}
T_{AV1} &= \{t_0, t_1, t_2\} \\
U_{AV1} &= (\{t_0\}, \{t_1, t_2\}, \{t_0\}) \\
R_{AV1} &= \{\{r1\}\} \\
O_{AV1} &= \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{0,6}, o_{0,7}, o_{0,8}, \\
&\quad o_{0,9}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{1,5}, o_{1,6}, o_{1,7}, o_{1,8}, o_{1,9}, \\
&\quad o_{1,10}, o_{1,11}, o_{1,12}, o_{2,1}, o_{2,2}, o_{2,3}, o_{2,4}, o_{2,5}, o_{2,6}, \\
&\quad o_{2,7}, o_{2,8}, o_{2,9}, o_{2,10}, o_{2,11}, o_{2,12}\} \\
Q_{AV1} &= \{(m, PMN)\} \\
F_{AV1} &= \{(o_{0,1}, o_{0,2}), (o_{0,2}, o_{0,3}), (r_1, o_{0,3}), \\
&\quad (o_{0,3}, o_{0,4}), (o_{0,4}, o_{0,5}), (o_{0,5}, o_{0,6}), (o_{0,6}, o_{0,7}), \\
&\quad (o_{0,7}, o_{0,8}), (r_1, o_{0,8}), (o_{0,8}, o_{0,9}), (o_{1,1}, o_{1,2}), \\
&\quad (o_{1,2}, o_{1,12}), (o_{1,2}, o_{1,3}), (o_{1,3}, o_{1,4}), (q_1, o_{1,4}), \\
&\quad (o_{1,4}, o_{1,5}), (o_{1,5}, r_1), (o_{1,5}, o_{1,6}), (o_{1,6}, q_1), \\
&\quad (o_{1,6}, o_{1,7}), (o_{1,7}, o_{1,8}), (q_1, o_{1,8}), (o_{1,8}, o_{1,9}), \\
&\quad (r_1, o_{1,9}), (o_{1,9}, o_{1,10}), (o_{1,10}, q_1), (o_{1,10}, o_{1,11}), \\
&\quad (o_{1,11}, o_{1,2}), (o_{1,11}, o_{1,12}), (o_{2,1}, o_{2,2}), (o_{2,2}, o_{2,12}), \\
&\quad (o_{2,2}, o_{2,3}), (o_{2,3}, o_{2,4}), (q_1, o_{2,4}), (o_{2,4}, o_{2,5}), \\
&\quad (o_{2,5}, r_1), (o_{2,5}, o_{2,6}), (o_{2,6}, q_1), (o_{2,6}, o_{2,7}), \\
&\quad (o_{2,7}, o_{2,8}), (q_1, o_{2,8}), (o_{2,8}, o_{2,9}), (r_1, o_{2,9}), \\
&\quad (o_{2,9}, o_{2,10}), (o_{2,10}, q_1), (o_{2,10}, o_{2,11}), (o_{2,11}, o_{2,2}), \\
&\quad (o_{2,11}, o_{2,12})\} \\
B^{BWD}_{AV1} &= \{(o_{1,5}, o_{1,9}), (o_{2,5}, o_{2,9})\}
\end{aligned}
$$

The presented AV1 application has two backward relations, which is a reason for possible atomicity violation error. To locate the atomicity violation error, the first step is to check if the operations of each pair from the B set are located in two different critical sections. The $o_{1.5}$ operation using the r_1 resource is located in the first critical section between the $o1.4$ mutex creation operation and the $o_{1.6}$ mutex release operation. The second operation, that is $o_{1.9}$, is between the operations $o_{1.8}$ and $o_{1.10}$. This means that both of these operations are in two different critical sections. However, this is not yet the same as atomicity violation errors. In the next step, it should be checked whether there is an operation in the parallel thread that can use the r_1 resource. However, in the second thread there are two such operations, i.e. $o_{2.5}$ and $o_{2.9}$, which proves that atomicity occurs between the following three operations $(o_{1,5}, o_{1,9}, o_{2,5})$ and $(o_{1,5}, o_{1,9}, o_{2,9})$.

The same should be done in the case of the second pair of operations from the B set. The search results are the following threes $(o_{2.5}, o_{2.9}, o_{1.5})$ and $(o_{2.5}, o_{2.9}, o_{1,9})$.

The approach presented above can be reduced to the verification of the following statement:

Theorem 3: There are operations $o_{i,\alpha}$ and$o_{i,\beta}$, related to each other by any of the contractual sequential relations. If there is a cyclic path in $\lambda^{P,i}$ of the P application that includes the mutex $q_s \lhd \lambda^{P,i}$ and only one operation from $\{o_{i,\alpha}, o_{i,\beta}\}$ then there is an atomicity violation error between these operations.

Proof: The proof results directly from the definition of atomicity violation. If there is a cyclic path in $\lambda^{P,i}$ of the P application that includes the mutex $q_s \lhd \lambda^{P,i}$ and only one operation from $\{o_{i,\alpha}, o_{i,\beta}\}$ it means that the operations $o_{i,\alpha}, o_{i,\beta}$ do not belong to one critical section. This means that it is allowed to perform the operation $o_{i,\epsilon}$ of another thread (t) on the same resource between operations $o_{i,\alpha}, o_{i,\beta}$ i.e.: $o_{i,\alpha} \prec o_{j,\epsilon} \prec o_{i,\beta}$. Thus, it is permissible to violate the atomicity of $o_{i,\alpha}, o_{i,\beta}$ operations by performing the $o_{j,\epsilon}$ operation.

The code of the homesoftserver application including the modification to eliminate the race condition, which also introduces the atomicity violation is presented below.

```
void* enable_logger(void *args)
{
   ...
            pthread_mutex_lock(&log_m);
            pthread_mutex_lock(&buf_index_m);
            for (int i = 0; i <= logger.index; ++i
                )
            {
                if (logger.buffer[i] != (char*)
                    NULL)
                {
                    const size_t log_lvl_str_index
                         = (size_t)logger.
                        log_levels[i] - 1;
                    printf(``LOG:%s:%s\n'',
                        log_lvl_str[
                        log_lvl_str_index], logger
                        .buffer[i]);
                    free(logger.buffer[i]);
                    logger.buffer[i] = (char*)NULL
                        ;
                }
            }
            logger.index = -1;
            pthread_mutex_unlock(&buf_index_m);
            pthread_mutex_unlock(&log_m);
   ...
}

void add_log(const char *log, enum LogLevel level)
{
   ...
    pthread_mutex_lock(&log_m);
    pthread_mutex_lock(&buf_index_m);
    ++logger.index;
    logger.buffer[logger.index % BUFFER_SIZE] =
        llog;
    logger.log_levels[logger.index % BUFFER_SIZE]
        = level;
    pthread_mutex_unlock(&buf_index_m);
    pthread_mutex_unlock(&log_m);


    pthread_mutex_lock(&log_m);
    pthread_mutex_lock(&buf_index_m);
    logger.index = logger.index % BUFFER_SIZE;
    pthread_mutex_unlock(&buf_index_m);
    pthread_mutex_unlock(&log_m);
}
```

In this example, there may be a situation where the incorrect state of the *logger.index* variable will cause the *enable_logger* function to malfunction, which will lead to errors in the application's operation.

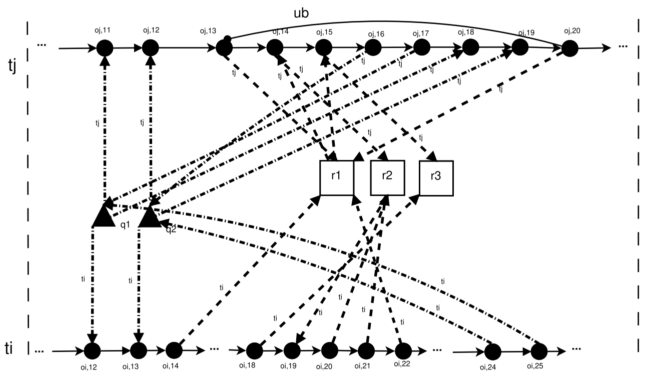

FIGURE 8. Fragment of the homesoftserver application model instance with atomicity violation.

The representation of C_P of the considered fragment of the **homesoftserver** application is presented below, the graphic form of which is shown in the figure 8.

$$
\begin{aligned}
T_{HSSAV} &= \{t_0, \ldots, t_i, t_j, \ldots\} \\
U_{HSSAV} &= (\{t_0\}, \ldots, \{t_i, t_j, \ldots\}, \ldots, \{t_0\}) \\
R_{HSSAV} &= \{\{logger.index\}, \{logger.buffer\}, \\
&\quad \{logger.levels\}, \ldots\} \\
O_{HSSAV} &= \{o_{i,12}, o_{i,13}, o_{i,14}, \ldots, o_{i,18}, o_{i,19}, o_{i,20}, \\
&\quad o_{i,21}, o_{i,22}, \ldots, o_{i,24}, o_{i,25}, \ldots, o_{j,11}, o_{j,12}, o_{j,13}, \\
&\quad o_{j,14}, o_{j,15}, o_{j,16}, o_{j,17}, o_{j,18}, o_{j,19}, o_{j,20}, \ldots\} \\
Q_{HSSAV} &= \{(log\_m, PMD), \\
&\quad (buf\_index\_m, PMD), \ldots\} \\
F_{HSSAV} &= \{\ldots, (q_1, o_{i,12}), (o_{i,12}, o_{i,13}), \\
&\quad (q_2, o_{i,13}), (o_{i,13}, o_{i,14}), \ldots, (o_{i,18}, r_3), (o_{i,18}, o_{i,19}), \\
&\quad (r_2, o_{i,19}), (o_{i,19}, o_{i,20}), (o_{i,20}, r_2), (o_{i,20}, o_{i,21}), \\
&\quad (o_{i,21}, r_2), (o_{i,21}, o_{i,22}), (o_{i,22}, r_1), \ldots, (o_{i,24}, q_2), \\
&\quad (o_{i,24}, o_{i,25}), (o_{i,25}, q_1), \ldots, (q_1, o_{j,11}), (o_{j,11}, o_{j,12}), \\
&\quad (q_2, o_{j,12}), (o_{j,12}, o_{j,13}), (o_{j,13}, r_1), (o_{j,13}, o_{j,14}), \\
&\quad (r_1, o_{j,14}), (o_{j,14}, r_2), (o_{j,14}, o_{j,15}), (o_{j,15}, r_3), \\
&\quad (o_{j,15}, o_{j,16}), (o_{j,16}, q_2), (o_{j,16}, o_{j,17}), (o_{j,17}, q_1),
\end{aligned}
$$

$$(o_{j,17}, o_{j,18}), (q_1, o_{j,18}), (o_{j,18}, o_{j,19}), (q_2, o_{j,19}),$$
$$(o_{j,19}, o_{j,20}), (o_{j,20}, r_1), \ldots\}$$
$$B^{FWD}_{HSSAV} = \{(o_{j,13}, o_{j,20})\}$$

This is a more extensive example than the two previous fragments as there are more operations that need to be considered for atomicity violation, and there is also a forward relationship edge. According to the introduced theorem, this error occurs because there are two operations connected with each other by the edge of the forward relationship, and these operations are located in two different critical sections [38]. In other words, between a pair of operations $(o_{j,13}, o_{j,20})$ there is an atomicity violation, because these are not in the same critical section, and the resource r_1 that is used by these operations is also shared by the $o_{i,22}$ operation.

In summary, code modifications aimed at elimination of e.g. race condition, can lead to atomicity violation. Correct identification of the atomicity violation error is conditioned by the information about relations between the operations. This knowledge is usually beyond the reach of software designers and programmers. This is due to the property of the C language, which has no mechanisms to include this information in the source code, and the programmers repeatedly omit it when developing documentation and commenting on the code. Some good practices can be mentioned for preventing atomicity violation:

- properly designed application architecture in such a way that pairs of operations that are connected by any of the contractual relations should be encapsulated,
- development of full documentation in which information about these relations will be included,
- transferring information about these relationships to all developers who work on the code for such an application.

E. ORDER VIOLATION

The order violation is the least researched of the errors described in this paper, it is confirmed by a small number of publications that address this problem. Additionally, the order violation is repeatedly confused with the atomicity violation. An order violation is caused by reversing the order of access to two (or more) memory areas (i.e. "A" should always be called before "B", but the order is not maintained during execution) [1].

The reversal of the order of operations most often results from placing both operations in two different threads and allowing these threads to work in the same time frame. Detection of the order violation is therefore conditional on the knowledge of contractual relations of the order of operations (similarly as in the case of atomicity violation). Taking into account these relations is connected with the new definition of order violation, which is as follows:

Definition 4: An order violation is an error where, between two operations of two different threads (or groups of operations), there is a sequence relation whose reversal causes the algorithm to malfunction and an undefined state of shared resources that have been used by this algorithm.

Taking into account the above definition, the problem of order violation detection can be formulated as follows:

Problem 4: The source code of the *P* multithreaded application is given, written in C using the pthreads library. In this application there are sequential relations between the operations of two threads and at least one pair of operations connected by a sequential relation is executed in the same interval. The answer to the question is sought: Is it possible to detect an order violation?

The described order violation cases can be found in the paper [1], which is a study of errors occurring in open-source multithreaded applications. The simplest example of an order violation is an attempt to execute operations on a resource by using an indicator to that resource, which has not yet been properly initialized. As a result, the resource has no memory space assigned to it (due to an uninitialized indicator), and any operations performed on it result in the unexpected closing of the application.

An example of an order violation application can be found at https://bit.ly/31DpGJU. The representation of the C_P of OV1 application is as follows:

$$T_{OV1} = (t_0, t_1, t_2)$$
$$U_{OV1} = (\{t_0\}, \{t_1, t_2\}, \{t_0\})$$
$$R_{OV1} = \{(string)\}$$
$$O_{OV1} = \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5},$$
$$o_{0,6}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{2,1}, o_{2,2},$$
$$o_{2,3}, o_{2,4}, o_{2,5}, o_{2,6}\}$$
$$Q_{OV1} = \{(n, PMD)\}$$
$$F_{OV1} = \{(o_{0,1}, o_{0,2}), (o_{0,2}, o_{0,3}), (o_{0,3}, o_{0,4}),$$
$$(o_{0,4}, o_{0,5}), (o_{0,5}, o_{0,6}), (q_1, o_{1,1}), (o_{1,1}, o_{1,2}),$$
$$(o_{1,2}, r_1), (o_{1,2}, o_{1,3}), (o_{1,3,q_1}), (o_{1,3}, o_{1,4}),$$
$$(o_{2,1}, o_{2,2}), (o_{2,2}, o_{2,3}), (q_1, o_{2,3}), (o_{2,3}, o_{2,4}),$$
$$(o_{2,4}, r_1), (o_{2,4}, o_{2,5}), (o_{2,5}, q_1), (o_{2,5}, o_{2,6})\}$$
$$B^{SYM}_{OV1} = \{(o_{1,2}, o_{0,5})\}$$
$$B^{BWD}_{OV1} = \{(o_{1,2}, o_{2,4})\}$$

In the OV1 application, there is a pair of operations $(o_{1,2}, o_{2,4})$, which is connected by the **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 the *string* indicator variable. This means that, in the application, it is allowed to perform these operations in any order - an order violation error occurs.

In order to identify the order violation, the following theorem was developed:

Theorem 4: Let P be a multithreaded application free of errors such as race condition, deadlock and atomicity violation. Let $B^{\xi}_P = B^{FWD}_P \cup B^{BWD}_P \cup B^{SYM}_P$ be a set of pairs of operations that are in sequence with each other, and ${}^{i,j}B^{\xi}_P \subseteq B^{\xi}_P$ will be a subset containing such pairs of operations $(o_{i,\alpha}, o_{j,\beta})$, the first of which is executed in the t_i thread and the second in the t_j thread. If $\{t_i, t_j\} \subseteq u_b$ then the order of execution of operations $(o_{i,\alpha}, o_{j,\beta})$ will be violated.

Proof: Proof is a direct consequence of the order violation definition. If the $\{t_i, t_j\}$ threads are executed in a common time frame, i.e., $\{t_i, t_j\} \subseteq u_b$ it is thus allowed to carry out the operation $(o_{i,\alpha}, o_{j,\beta})$ concurrently. It also means 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 acceptable to violate the set order of operations $(o_{i,\alpha}, o_{j,\beta})$.

The considered example of the *main* function code from the listing below:

```
int main() {
    pthread_t logger_thread,
        command_listener_thread,
        initialize_devices_thread;
    log_level = INFO;

    initilize_logger();
    pthread_create(&initialize_devices_thread,
        NULL, initialize_devices, NULL);
    pthread_create(&logger_thread, NULL,
        enable_logger, NULL);
    pthread_create(&command_listener_thread, NULL,
         command_listener, NULL);

    pthread_join(initialize_devices_thread, NULL);
    pthread_join(command_listener_thread, NULL);
    pthread_join(logger_thread, NULL);

    return 0;
}

void initialize_camera(camera_t* camera, const
    char* id, int x, int y, int z)
{
   ...
    camera->x_angle = x;
    camera->y_angle = y;
    camera->z_angle = z;
   ...
}

void change_camera_angles(camera_t* camera, int x,
     int y, int z)
{
   ...
    camera->x_angle = x;
    camera->y_angle = y;
    camera->z_angle = z;
}
```

The **homesoftserver** application assumes that device initialization in the *initialize_devices* function can be delegated to a separate thread. It was expected that the thread would be executed in its entirety always before the thread listening to commands using *command_listener*. However, it is possible that the device initialization thread will be delayed. During this time, the thread listening to the commands may receive, for example, a command with a change of camera angle. However, before the cameras are properly set, the delayed initialization thread will overwrite the submitted settings and the camera will return to the original settings.

The following is a representation of C_P of the source code under consideration whose graphic form is illustrated by the figure 9 showing the graphic representation.

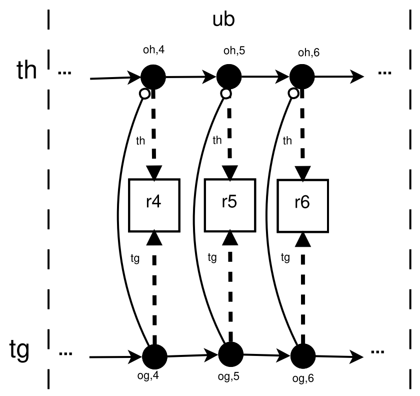

FIGURE 9. Fragment of the homesoftserver application model instance with the order violation.

$$
\begin{aligned}
T_{HSSOV} &= \{t_0, \ldots, t_g, t_h, \ldots\}\\
U_{HSSOV} &= (\{t_0\}, \ldots, \{t_g, t_h, ..\}, \ldots, \{t_0\})\\
R_{HSSOV} &= \{\ldots, \{camera.x\_angle\},\\
&\quad \{camera.y\_angle\}, \{camera.z\_angle\}, \ldots\}\\
O_{HSSOV} &= \{\ldots, o_{g,4}, o_{g,5}, o_{g,6}, \ldots, o_{h,4}, o_{h,5},\\
&\quad o_{h,6}, \ldots\}\\
Q_{HSSOV} &= \{\ldots\}\\
F_{HSSOV} &= \{\ldots, (o_{g,4}, r_4), (o_{g,4}, o_{g,5}), (o_{g,5}, r_5),\\
&\quad (o_{g,5}, o_{g,6}), (o_{g,6}, r_6), \ldots, (o_{h,4}, r_4), (o_{h,4}, o_{h,5}),\\
&\quad (o_{h,5}, r_5), (o_{h,5}, o_{h,6}), (o_{h,6}, r_6), \ldots\}\\
B^{BWD}_{HSSOV} &= \{\ldots, (o_{g,4}, o_{h,4}), (o_{g,5}, o_{h,5}),\\
&\quad (o_{g,6}, o_{h,6}), \ldots\}
\end{aligned}
$$

For the case under consideration, Theorem no. 4 is fulfilled. There are three pairs of operations causing the order violation in the application - each of the pairs is connected backwards, and the operations of these pairs are performed in one time interval. As a result, delegating initialization to a separate thread resulted in not one, but three order violations. Prevention of order violation is similar in difficulty to atomicity violation prevention. It is best practice to always put logically linked operations in one thread. However, as with atomicity violation, this is not always possible. Sometimes more complicated mechanisms such as conditional variables are used. However, their use may lead to an error called "lost signal", the symptoms of which may be similar to deadlock [62].

V. VERIFICATION OF METHOD BASED ON MASCM

A. TOOL SUPPORTING THE METHOD

The *rdao detector* application was used to detect the errors described above, which is the first implementation of the error detection method discussed in this paper. The application has 105 unit tests that allow to verify of the correctness of the generated models, locate specific errors and verify the correct operation of specific application mechanisms. All the tests of

TABLE 3. **Results of homeoftserver application analysis obtained with *rdao detector* application.**

Error	Analysis time*	Result
Without known errors	1m 14,797s	-
Race condition	56,938s	Found
Deadlock	1m 13,784s	Found
Atomicity violation	1m 35,244s	Not Found
Order violation	1m 19,856s	Not Found

* The given analysis time is the highest time obtained from 3 measurements

the generated models and the tests of the location of errors use a set of files with the source code of many different applications, which is designed to prevent regression errors in the *rdao detector* tool. Tests are also used to verify the correctness of previously made theorems.

B. OCCURRENCE OF FALSE POSITIVE ALARMS

The *rdao detector* tool reports false-positive errors [37], [70] due to the method used.

For the application version that is free from the errors described above, the following false-positive errors are reported:

- 3 race condition,
- 4 deadlock,
- 1 atomicity violation,
- 3 order violation.

However, the report is much larger, with far more entries than the 11 listed above. After analysis, it turns out that many of them are copies of the same notification. Their number results from two reasons. First of all, apart from the two threads and the main thread, the remaining threads are created in a loop, so each of the so created threads is considered as a pair of two identical threads for each of the 4 possible tasks, which gives a total of 8 dynamically created threads working in the same time unit plus 2 threads created by the programmer responsible for displaying logs and creating threads dynamically. As a result, as many as 10 threads are considered to work in one time interval. Secondly, all threads created dynamically use the mechanism of passing a variable number of arguments. Thanks to this approach the application code is consistent with "*don't repeat yourself*" principle. Unfortunately, the *rdao detector* does not have any variable tracking mechanism implemented at the moment, taking into account the variable number of arguments. As a result, some errors reported by the application are false-positive errors.

C. ANALYSIS OF THE HOMEOFTSERVER APPLICATION CODE

Table 3 shows the results of detecting previously presented errors from the homesoftserver application using the *rdao detector*. Based on the results from the table it can be concluded that the implementation of the method based on the Multithreaded Application Source Code Model in the *rdao detector* application has only managed to find two of the four errors.

The search time for race condition is the lowest and almost 20 seconds shorter than the analysis of the application without these errors. This is because the searched error is due to the lack of mutexes creation and release, which is connected with the reduced number of operations that the *rdao detector* application must perform. The application correctly reported race condition for all operations of the *add_log* function, which should be protected with mutexes.

The second error that was found is the previously described deadlock. The time of searching for it was very similar to the time of analyzing the application without these errors because the number of elements and operations was the same, however, the two operations had changed the order. The reported error has all the necessary information to locate the cause of the deadlock.

Locating both the race condition and deadlock errors was successful because both of these errors result directly from the C-language code architecture and all necessary elements were in the generated model instance.

Locating atomicity violation and order violation requires additional information about the relationship between operations. Additional information was provided, but as a result of an error noticed during research in *rdao detector*, it was not possible to locate atomicity and order violation in the code of the homesoftserver application. The error in the *rdao detector* application occurs in the model instance generation algorithm, which is due to the lack of support for the variable number of arguments in the current, first version of the *rdao detector*, which has been further explained.

The time of application analysis to detect atomicity violation is higher than the time of application analysis without these errors. The increased analysis time is influenced by the fact that the number of operations of creating and removing mutexes in the application with atomicity violation is higher than in the correct version.

In case of order violation, the analysis time is only 5 seconds longer than the analysis time of the application without the mentioned errors. Here, however, the comparison is not as simple as in the case of other errors, because the introduction of the order violation required a change in the application architecture of the homesoftserver.

To sum up, the *rdao detector* application in its current version as a tool to support the work of programmers contains several imperfections that prevent it from being used in commercial software development. First of all, it failed to locate atomicity violation and order violation, however, solving this problem is one of the most important plans. The time of application analysis is also unsatisfactory, as it oscillates around one minute and is strongly linked to the number of mutexes and operations related to them. As a result, the application requires further work on improving model instance generation algorithms and accelerating the error detection algorithm. However, it is worth emphasizing 100% effectiveness in detecting race condition and deadlock errors. All errors of this type that occurred in the homesoftserver application were located.

D. ANALYSIS OF SAMPLE APPLICATIONS OF THE PHOENIX PROJECT

The Phoenix project described in [63] is based on Google's MapReduce model, and the sample applications provided in the project have become a kind of benchmark for methods that allow eliminating these errors through the transactional memory mechanism [54], [56], [57], [64]. Unfortunately, among the static methods presented earlier, none were used to analyze sample applications of the Phoenix project. However, they will be used to check the effectiveness of *rdao detector* applications. A computer with the following parameters was used for the experiment:

- Processor: AMD Ryzen 5 1500X, 3.5 GHz,
- RAM: Corsair Vengeance LPX, DDR3, 32GB, 3000MHz, CL15,
- Disc: SSD Samsung 970 Evo 1 TB M.2 2280, Sequential Read 3400 MB/s.

The computer worked under Windows 10 and the *rdao detector* application was run with the Python 3.8.0 parser and the results are presented on the following pages. In the code of the histogram program, using the *rdao detector* application, 81 places where race condition could occur were located. A manual analysis showed that 15 reports were false and 45 of the remaining reports were duplicates. The remaining 21 reports concern parallel operations on one shared resource (or its components) by 4 threads working in the same time interval, and none of these operations is protected with a mutex.

The analysis of the kmeans source code resulted in locating 141 places where race condition may occur. After deducting duplicates and false-positive errors from this number, there are 21 places where race condition errors occur. Similar results were given for the reverse_index analysis results showed 147 places, 19 of which are actually places where race condition occurs. In results for both applications is easy to see near 50% of reported issues are false positives.

The kmeans application also reported 26 places where the order violation would occur. All these reports were related to the malloc and free functions and turned out to be false.

This result shows how hard is locate order violation in the static analysis process. Also, it was hard to verify this result manually because there was a need to completely understand what application do.

Analysis of the pca application source code showed as many as 226 sites with a potential race condition. Only 58 of all entries are places where race condition occurs. The remaining 168 reports were false-positive errors and duplicates. Two places where an order violation occurs were also reported in the pca application. However, these reports turned out to be false.

Word_count analysis resulted in reporting only one order violation, which turned out to be false, whereas the analysis of the linear_regression, matrix_multiply, and string_match applications showed no errors. It does not mean these applications is bugs free, however, it is possible to write multithread applications using Phoenix project which contains concurrent bugs. Phoenix project is probably written in the best way it is possible however programmers still can use it in an incorrect way which leads to concurrent errors.

Programs from Phoenix projects analyzed in this section consist of at least 1500 lines of code. However, program source code length has no as much impact as a number of different threads. In the future the rdao_detector will be used to detect errors in programs with thousands of lines of code, and for such cases, a linear increase in the running time of the program is expected. Before this can happen, however, significant optimizations and improvements need to be made. Most important is to reduce false positives and to speed up of algorithm, especially part responsible for deadlocks localization.

E. CONCLUSION FROM VERIFICATION

The table no. 4 (4) shows the results obtained from the analysis of the Phoenix project. Unfortunately, based on these results and the literature, the effectiveness of the proposed solution cannot be clearly determined. It can be assumed that in the case of race condition and deadlock error detection the effectiveness is close to 100%. This is due to a review of the Phoenix project code, where no errors were found that *rdao detector* application would not show. However, it is not clear whether there are more errors in the Phoenix project. A special case here is the atomicity violation, as it cannot be deduced from the application code that the two functions defined by the programmer are in relation to each other. The programs described are unreported and poorly commented, which effectively prevents such information from being extracted even if the applications are subjected to very thorough analysis.

The situation is similar to locating an order violation, although in this case, the *rdao detector* reported several potential locations where an order violation may occur. However, after analysis, these reports turned out to be false alarms. False-positive errors in such cases are very common. This is because some of the functions of the standard C library are in relation to each other, which by default are checked by the *rdao detector* and in which case it is very easy to get a structure that resembles the one expected by the *rdao detector* algorithm responsible for locating the order violation.

Summarizing the analysis of sample applications from the Phoenix project we can say that the current implementation of MASCM detection in *rdao detector* application needs to be refined. However, there are indications that with the development of *rdao detector* or other MASCM implementation it will be possible to get an efficient and reliable tool for static code analysis to detect these errors.

In order to increase the effectiveness of the application, the number of test scenarios in which the application should work based on the results of the tests should be increased. It is also necessary to analyze the existing application code to make sure that no errors were made anywhere during the implementation of error finding algorithms.

TABLE 4. Results of the analysis of sample applications of the Phoenix project obtained using the application *rdao detector*.

1. Reported errors 2. False positives alarms 3. Duplicated alarms 4. Real errors

Program	Analysis time	Highest memory usage	Result															
			Race condition				Deadlock				Atomicity violation				Order violation			
			1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
histogram	3m 12.02s	10.46 MB	**81**	**15**	**45**	**21**	0	0	0	0	0	0	0	0	0	0	0	0
kmeans	3m 36.89s	10.68 MB	**141**	**73**	**47**	**21**	0	0	0	0	0	0	0	0	**26**	**26**	0	0
linear_regression	1.34s	2.43 MB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
matrix_multiply	1.74s	2.90 MB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
pca	13m 2.54s	16.59 MB	**226**	**60**	**108**	**58**	0	0	0	0	0	0	0	0	**2**	**2**	0	0
reverse_index	3m 33.11s	10.84 MB	**147**	**85**	**43**	**19**	0	0	0	0	0	0	0	0	0	0	0	0
string_match	1.91s	2.80 MB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
word_count	2.66s	3.36 MB	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0	0

It was also mentioned earlier that a mechanism should be developed to track arguments to handle functions with a variable number of arguments. The implementation of this type of mechanisms is very difficult because C language allows passing parameters through value and indicator. In C language, it is also possible to operate on indicators, which should also be taken into account when tracking resources, e.g. the process of projecting an indicator from one type to another, does not affect in any way the state of the resource placed under the indicated address, but only the way it is interpreted. The omission of such important details as the one mentioned with the projection affects the number of reported false-positive errors.

F. MODEL ALTERNATIVE APPLICATIONS

The Multithreaded Application Source Code Model discussed in this paper was developed to locate errors in multithreaded applications based on their architecture. In other words, this model is used to check whether the application architecture allows for the probability of one of these errors.

The application of the developed model in other areas without making changes in the model structure may prove impossible. However, any process whose architecture allows parallel execution of tasks, while limiting access to selected resources to only a selected unit at the same time, can be reduced to a model instance based on MASCM. As an example, let's use a laboratory, where there is a limited number of rooms allowing to perform very dangerous tests (let their number be N). Due to the high priority of these tests, many teams work in the laboratory in parallel (let their number be M) and these teams have to work alternately in the laboratory (M > N). In this process, however, there are no specially designated working hours in the laboratory, so access to the laboratory is random and queued. In the example shown, the laboratory corresponds to the resources and the teams to the threads. The equivalent of mutexes, here the safety mechanisms of such a laboratory, e.g. doors, may be used.

Of course, working in such a laboratory may be more complicated, i.e. one team may need to occupy not one but three rooms in such a way that none of them can be accessible to another team at that time. Specialized tools, or even whole sets of them, may also be a resource to carry out complicated research processes. In the case of tool sets, poor tool allocation can lead to the blocking of each team that received an incomplete tool set. In many cases, such as the division of tools, groups of people can solve the problem in a natural collaborative process or through the decision of a supervisor. However, if these teams weren't made of humans, but robots, they wouldn't necessarily be able to solve such problems in a human-like manner. Therefore, the architecture of the process of using laboratories and tools must be very precise so as not to lead to an accident or blocking the work of groups.

VI. SUMMARY

This work contains a complete description of the previously developed multithreaded application source code model. This model was developed to detect race condition, deadlock, atomicity violation, and order violation errors in multithreaded applications written in C language using pthreads library. The errors sought were described in detail, examples of applications containing these errors were presented, and then mechanisms for detecting them using MASCM were developed. The theorems and proof for locating these errors using the developed model were also worked out. This allowed to develop a tool, i.e. a *rdao detector* application, which is a sample implementation of an automatic mechanism of locating the discussed errors in the source code of the application.

The effectiveness of this application was also investigated using it to analyze sample multithreaded applications of the Phoenix project. The outcome of the analysis obtained as a result of *rdao detector* was manually verified in the process of source code review of the tested applications. This process showed that the effectiveness of the tool in the case of race condition and deadlock is estimated at 100%, while pointing out that this value may be different in reality, because the actual state of errors in the tested applications is not known. In case of order violation and atomicity violation errors, the tool's effectiveness is also affected by the presence of an error located in the application during testing, namely the lack of a resource tracking mechanism for functions with a variable number of arguments.

The subject of research presented in this paper is not yet closed. First of all, the method presented in this paper should be confronted with other methods that allow to locate the same group of errors, or with a group of tools, the results of which together will allow to effectively evaluate *rdao detector* results. However, the literature has not found such a tool set that could be used for this purpose. The developed tools for static analysis focus on race condition and deadlock errors, so the process can only be carried out for two types of errors. In other cases, the only solution is to confront the results with the results of mixed and dynamic methods if they are able to clearly indicate the place in the application code where the error occurs. An alternative solution is to build a team of programmers with varying degrees of knowledge of C and multithreaded programming, as well as software architects. This team would be divided into a group of architects and programmers, with the former focusing on the development of application architectures containing the identified bugs and evaluation of their implementation, while the latter would focus on the implementation of the developed scenarios. The result of the work of such a team should be a set of many well-documented applications in which the errors discussed in this study would occur. The documentation would include a detailed description of the application and the errors in them so that the description would leave no doubt as to where the bug is. Only on this basis, it would be possible to evaluate the existing solutions and further improve them. Continuous development of such a set of applications and its documentation, in order to provide as many different types of scenarios as possible, would be very good material for further research.

Future studies should be carried out taking into account state space explosion, control flow sensitivity, and pointer analysis to avoid false positives. Research into all these areas will allow the developed method to be improved in few areas: reducing memory usage, speeding up error detection, and reducing false positives. Currently, *rdao detector* algorithm basis on 105 tests and most of them use small C programs which allow do not introduce regression bugs.

REFERENCES

[1] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. 13th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*. New York, NY, USA: Association Computing Machinery, 2008, vol. 43, no. 3, pp. 329–339, doi: 10.1145/1346281.1346323.

[2] *Linux Programmer's Manual*. Accessed: Apr. 10, 2021. [Online]. Available: https://www.man7.org/linux/man-pages/man7/pthreads.7.html

[3] A. Koltsidas and O. Peterson, "Virtual AUTOSAR environment on Linux-evaluation study on performance gains from running ECU applications on POSIX threads," Dept. Comput. Sci. Eng., Chalmers Univ. Technol. Univ. Gothenburg, Gothenburg, Sweden, Tech. Rep., 2016. Accessed: Apr. 10, 2021. [Online]. Available: http://publications.lib.chalmers.se/records/fulltext/238391/238391.pdf

[4] J. Park, B. Choi, and S. Jang, "Dynamic analysis method for concurrency bugs in multi-process/multi-thread environments," *Int. J. Parallel Program.*, vol. 48, no. 6, pp. 1032–1060, Dec. 2020, doi: 10.1007/s10766-020-00661-3.

[5] S. A. Asadollah, D. Sundmark, S. Eldh, and H. Hansson, "Concurrency bugs in open source software: A case study," *J. Internet Services Appl.*, vol. 8, no. 1, pp. 1–15, Dec. 2017, doi: 10.1186/s13174-017-0055-2.

[6] K. B. Pierce and H.-Y. Chau, "Tools and methods for discovering race condition errors," U.S Patent US 7 174 554 B2, Feb. 6, 2007. Accessed: Apr. 10, 2021. [Online]. Available: https://patentimages.storage.googleapis.com/80/e3/9c/9e41ad7da9b31e/US7174554.pdf

[7] R. J. Berg, L. Rose, J. Peyton, J. J. Danahy, R. Gottlieb, and C. Rehbein, "Method and system for detecting race condition vulnerabilities in source code," U.S. Patent US 7 398 516 B2, Jul. 8, 2008. Accessed: Apr. 10, 2021. [Online]. Available: https://patentimages.storage.googleapis.com/2b/ca/ca/746ba6a3ed9914/US7398516.pdf

[8] Y. Wang, F. Gao, L. Wang, T. Yu, J. Zhao, and X. Li, "Automatic detection, validation and repair of race conditions in interrupt-driven embedded software," *IEEE Trans. Softw. Eng.*, early access, Apr. 20, 2020, doi: 10.1109/TSE.2020.2989171.

[9] H. Liang, M. Li, and J. Wang, "Automated data race bugs addition," in *Proc. 13th Eur. Workshop Syst. Secur.* New York, NY, USA: Association Computing Machinery, Apr. 2020, pp. 37–42, doi: 10.1145/3380786.3391401.

[10] J. S. Teh, M. Alawida, and A. Samsudin, "Generating true random numbers based on multicore CPU using race conditions and chaotic maps," *Arabian J. Sci. Eng.*, vol. 45, no. 12, pp. 10019–10032, Dec. 2020, doi: 10.1007/s13369-020-04552-0.

[11] D. Giebas and R. Wojszczyk, "Graphical representations of multithreaded applications," *Appl. Comput. Sci.*, vol. 14, no. 2, pp. 20–37, 2018, doi: 10.23743/acs-2018-10.

[12] K. Jensen and L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Berlin, Germany: Springer-Verlag, 2009, doi: 10.1007/b95112.

[13] R. Netzer, "Race condition detection for debugging shared-memory parallel programs," Dept. Comput. Sci., Univ. Wisconsin-Madison, Madison, WI, USA, Tech. Rep. 1039, 1991.

[14] C. Flanagan and S. N. Freund, "Type-based race detection for Java," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*. New York, NY, USA: Association Computing Machinery, 2000, vol. 35, no. 5, pp. 219–232, doi: 10.1145/349299.349328.

[15] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for Java," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 2, pp. 207–255, Mar. 2006, doi: 10.1145/1119479.1119480.

[16] A. M. Alghamdi and F. E. Eassa, "OpenACC errors classification and static detection techniques," *IEEE Access*, vol. 7, pp. 113235–113253, 2019, doi: 10.1109/ACCESS.2019.2935498.

[17] A. Veidenbaum, K. Joe, H. Amano, and H. Aiso, *High Performance Computing*. Berlin, Germany: Springer-Verlag, 2003, doi: 10.1007/b14207.

[18] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 7th ed. Hoboken, NJ, USA: Wiley, 2005.

[19] D. Giebas and R. Wojszczyk, "Deadlocks detection in multithreaded applications based on source code analysis," *Appl. Sci.*, vol. 10, no. 2, p. 532, Jan. 2020, doi: 10.3390/app10020532.

[20] M. Mitchell, J. Oldham, and A. Samuel, *Advanced Linux Programming*. USA: New Riders Publishing Jun. 2001. [Online]. Available: http://www.cse.hcmut.edu.vn/~hungnq/courses/nap/alp.pdf

[21] Y. Wang, H. Liao, S. Reveliotis, T. Kelly, S. Mahlke, and S. Lafortune, "Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software," in *Proc. 48h IEEE Conf. Decis. Control (CDC) Held Jointly 28th Chin. Control Conf.*, Dec. 2009, pp. 4971–4976, doi: 10.1109/CDC.2009.5399950.

[22] Y. Lin and S. S. Kulkarni, "Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*. New York, NY, USA: Association Computing Machinery, 2014, pp. 237–247, doi: 10.1145/2610384.2610398.

[23] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proc. 10th Joint Meeting Found. Softw. Eng.* New York, NY, USA: Association Computing Machinery, Aug. 2015, pp. 532–543, doi: 10.1145/2786805.2786825.

[24] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 144–156, May 1983, doi: 10.1145/357360.357365.

[25] A. Ho, S. Smith, and S. Hand, "On deadlock, livelock, and forward progress," Comput. Lab., Univ. Cambridge, Cambridge, U.K., Tech. Rep. UCAM-CL-TR-633, 2005. Accessed: Apr. 10, 2021. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-633.pdf

[26] M. Singhal, "Deadlock detection in distributed systems," *Computer*, vol. 22, no. 11, pp. 37–48, Nov. 1989, doi: 10.1109/2.43525.

[27] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proc. 32nd ACM Sigplan Conf. Program. Lang. Design Implement.* New York, NY, USA: Association Computing Machinery, 2011, vol. 46, no. 6, pp. 389–400, doi: 10.1145/1993498.1993544.

[28] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 299–309, doi: 10.1109/ICSE.2012.6227184.

[29] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: Static race detection on millions of lines of code," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. (ESEC-FSE)*, 2007, pp. 205–214.

[30] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, no. 5, pp. 237–252, Dec. 2003, doi: 10.1145/1165389.945468.

[31] N. Koutsopoulos, M. Northover, T. Felden, and M. Wittiger, "Advancing data race investigation and classification through visualization," in *Proc. IEEE 3rd Work. Conf. Softw. Visualizat. (VISSOFT)*, Sep. 2015, pp. 200–204, doi: 10.1109/VISSOFT.2015.7332437.

[32] D. Qi, N. Gu, and J. Su, "Detecting data race in network applications using static analysis," in *Proc. Int. Conf. Netw. Netw. Appl. (NaNA)*, Oct. 2019, pp. 313–318, doi: 10.1109/NaNA.2019.00061.

[33] L. Lima, A. Tavares, and S. C. Nogueira, "A framework for verifying deadlock and nondeterminism in UML activity diagrams based on CSP," *Sci. Comput. Program.*, vol. 197, Oct. 2020, Art. no. 102497, doi: 10.1016/j.scico.2020.102497.

[34] J. Li, X. Liu, L. Jiang, B. Liu, Z. Yang, and X. Hu, "An intelligent deadlock locating scheme for multithreaded programs," in *Proc. 3rd Int. Conf. Intell. Syst., Metaheuristics Swarm Intell.*, Mar. 2019, pp. 14–18, doi: 10.1145/3325773.3325781.

[35] C. Laneve, "A lightweight deadlock analysis for programs with threads and reentrant locks," *Sci. Comput. Program.*, vol. 181, pp. 64–81, Jul. 2019, doi: 10.1016/j.scico.2019.06.002.

[36] E. Giachino and C. Laneve, "Deadlock detection in linear recursive programs," in *Proc. Int. School Formal Methods Design Comput., Commun. Softw. Syst.* Cham, Switzerland: Springer, 2014, pp. 26–64, doi: 10.1007/978-3-319-07317-0_2.

[37] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing atomicity violation bugs from their hiding places," in *Proc. 14th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*. New York, NY, USA: Association Computing Machinery, 2009, vol. 44, no. 3, pp. 25–36, doi: 10.1145/1508244.1508249.

[38] D. Giebas and R. Wojszczyk, "Atomicity violation in multithreaded applications and its detection in static code analysis process," *Appl. Sci.*, vol. 10, no. 22, p. 8005, Nov. 2020, doi: 10.3390/app10228005.

[39] D. Giebas and R. Wojszczyk, "Order violation in multithreaded applications and its detection in static code analysis process," *Appl. Comput. Sci.*, vol. 16, no. 4, pp. 103–117, 2020, doi: 10.23743/acs-2020-32.

[40] L. Chew and D. Lie, "Kivati: Fast detection and prevention of atomicity violations," in *Proc. 5th Eur. Conf. Comput. Syst. (EuroSys)*. New York, NY, USA: Association Computing Machinery, 2010, pp. 307–320, doi: 10.1145/1755913.1755945.

[41] Y. Yang, A. Gringauze, D. Wu, and H. K. Rohde, "Detecting data race and atomicity violation via typestate-guided static analysis," U.S. Patent 8 510 722, Aug. 13, 2013. Accessed: Apr. 10. 2021. [Online]. Available: https://patentimages.storage.googleapis.com/32/aa/55/04a49ec0118682/US8510722.pdf

[42] J. Fiedor, B. Křena, Z. Letko, and T. Vojnar, "A uniform classification of common concurrency errors," in *Computer Aided Systems Theory—EUROCAST 2011*. Berlin, Germany: Springer, 2011, pp. 519–526, doi: 10.1007/978-3-642-27549-4_67.

[43] Y. Zhou, S. Lu, and J. A. Tucek, "Atomicity violation detection using access interleaving invariants," U.S. Patent 8 533 681, Sep. 10, 2013. Accessed: Apr. 10, 2021. [Online]. Available: https://patentimages.storage.googleapis.com/1c/46/35/4c85f52a086a2f/US8533681.pdf

[44] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting atomicity violations via access interleaving invariants," *ACM SIGOPS Operating Syst. Rev.*, vol. 40, no. 5, pp. 37–48, 2006, doi: 10.1145/1168917.1168864.

[45] M. Xu, R. Bodík, and M. D. Hill, "A serializability violation detector for shared-memory server programs," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 1–14, Jun. 2005, doi: 10.1145/1064978.1065013.

[46] U. Mathur and M. Viswanathan, "Atomicity checking in linear time using vector clocks," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: Association Computing Machinery, Mar. 2020, pp. 183–199, doi: 10.1145/3373376.3378475.

[47] *TIOBE Index*. Accessed: Jul. 4, 2020. [Online]. Available: https://www.tiobe.com/tiobe-index/

[48] X. Chang, W. Dou, Y. Gao, J. Wang, J. Wei, and T. Huang, "Detecting atomicity violations for event-driven Node.Js applications," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 631–642, doi: 10.1109/ICSE.2019.00073.

[49] Z. Yu, L. Song, and Y. Zhang, "Fearless concurrency? Understanding concurrent programming safety in real-world rust software," 2019, *arXiv:1902.01906*. [Online]. Available: http://arxiv.org/abs/1902.01906

[50] M. Roberson and C. Boyapati, "A static analysis for automatic detection of atomicity violations in Java programs," Dept. Elect. Eng. Comput. Sci., Univ. Michigane, Ann Arbor, MI, USA, Tech. Rep. CSE-TR-569-11, 2011. Accessed: Apr. 10, 2021. [Online]. Available: https://www.eecs.umich.edu/eecs/etc/research/techreports/cse_tr/database/reports.cgi?11

[51] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller, "HAVE: Detecting atomicity violations via integrated dynamic and static analysis," in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.* Berlin, Germany: Springer, 2009, pp. 425–439, doi: 10.1007/978-3-642-00593-0_30.

[52] C. T. Lopez, S. Marr, E. G. Boix, and H. Mössenböck, "A study of concurrency bugs and advanced development support for actor-based programs," in *Programming With Actors: State-of-the-Art and Research Perspectives*. Cham, Switzerland: Springer, 2018, pp. 155–185, doi: 10.1007/978-3-030-00302-9_6.

[53] D. Chen, Y. Jiang, C. Xu, X. Ma, and J. Lu, "Testing multithreaded programs via thread speed control," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.* New York, NY, USA: Association Computing Machinery, Oct. 2018, pp. 15–25, doi: 10.1145/3236024.3236077.

[54] Z. Yu, Y. Zuo, and W. C. Xiong, "Concurrency bug avoiding based on optimized software transactional memory," *Sci. Program.*, vol. 2019, pp. 1–19, Feb. 2019, doi: 10.1155/2019/9404323.

[55] N. Vinesh, S. Rawat, H. Bos, G. Herbert, C. Giuffrida, and M. Sethumadhavan, "ConFuzz—A concurrency fuzzer," in *Proc. 1st Int. Conf. Sustainable Technol. Comput. Intell.* Singapore: Springer, 2020, pp. 667–691, doi: 10.1007/978-981-15-0029-9_53.

[56] Z. Yu, Y. Zuo, and Y. Zhao, "Convoider: A concurrency bug avoider based on transparent software transactional memory," *Int. J. Parallel Program.*, vol. 48, no. 1, pp. 32–60, Feb. 2020, doi: 10.1007/s10766-019-00642-1.

[57] E. D. Berger, T. Yang, T. Liu, and G. Novark, "Grace: Safe multithreaded programming for C/C++," in *Proc. 24th ACM SIGPLAN Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)* New York, NY, USA: Association Computing Machinery, 2009, vol. 44, no. 10, pp. 81–96, doi: 10.1145/1640089.1640096.

[58] C.-S. Shih and J. A. Stankovic, "Survey of deadlock detection in distributed concurrent programming environments and its application to real-time systems and Ada," Dept. Comput. Inf. Sci., Univ. Massachusetts, Shrewsbury, MA, USA, Tech. Rep. UM-CS-1990-069, 1990.

[59] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke, "Gadara: Dynamic deadlock avoidance for multithreaded programs," in *Proc. OSDI*, vol. 8, 2008, pp. 281–294.

[60] S. Lafortune, Y. Wang, and S. Reveliotis, "Eliminating concurrency bugs in multithreaded software: An approach based on control of Petri nets," in *Proc. Int. Conf. Appl. Theory Petri Nets Concurrency*. Berlin, Germany: Springer, 2013, pp. 21–28, doi: 10.1007/978-3-642-38697-8_2.

[61] C. V. Praun and T. R. Gross, "Static detection of atomicity violations in object-oriented programs," *J. Object Technol.*, vol. 3, no. 6, pp. 103–122, 2004.

[62] K. M. Kavi, A. Moshtaghi, and D.-J. Chen, "Modeling multithreaded applications using Petri nets," *Int. J. Parallel Program.*, vol. 30, no. 5, pp. 353–371, 2002, doi: 10.1023/A:1019917329895.

[63] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit.*, Feb. 2007, pp. 13–24, doi: 10.1109/HPCA.2007.346181.

[64] H. K. Pyla and S. Varadarajan, "Avoiding deadlock avoidance," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2010, pp. 75–85.

[65] P. Louridas, "Static code analysis," *IEEE Softw.*, vol. 23, no. 4, pp. 58–61, Jul. 2006, doi: 10.1109/MS.2006.114.

[66] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller, "Automated type-based analysis of data races and atomicity," in *Proc. 10th ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*. New York, NY, USA: Association Computing Machinery, 2005, pp. 83–94, doi: 10.1145/1065944.1065956.

[67] C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2010, pp. 328–342, doi: 10.1007/978-3-642-12002-2_27.

[68] J. Yang, B. Jiang, and W. K. Chan, ‘‘HistLock+: Precise memory access maintenance without lockset comparison for complete hybrid data race detection,’’ *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 786–801, Sep. 2018, doi: 10.1109/TR.2018.2832226.
[69] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, ‘‘Eraser: A dynamic data race detector for multithreaded programs,’’ *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997, doi: 10.1145/265924.265927.
[70] T. Liu, J. Zhou, S. Silvestro, and H. Liu, ‘‘Defeating deadlocks in production software,’’ U.S. Patent 16 159 234, Feb. 9, 2021. Accessed: Apr. 10, 2021. [Online]. Available: https://patentimages.storage.googleapis.com/e4/41/2f/1a383b1e8541b4/US10915424.pdf
[71] K. Lautenbach and H. A. Schmid, ‘‘Use of Petri nets for proving correctness of concurrent process systems,’’ in *Proc. IFIP Congr.* Amsterdam, The Netherlands: North-Holland, 1974, pp. 187–191.
[72] K. Serebryany and T. Iskhodzhanov, ‘‘ThreadSanitizer: Data race detection in practice,’’ in *Proc. Workshop Binary Instrum. Appl. (WBIA)*. New York, NY, USA: Association Computing Machinery, 2009, pp. 62–71, doi: 10.1145/1791194.1791203.
[73] A. Tehrani, M. Khaleel, R. Akbari, and A. Jannesari, ‘‘DeepRace: Finding data race bugs via deep learning,’’ 2019, *arXiv:1907.07110*. [Online]. Available: http://arxiv.org/abs/1907.07110
[74] C. Flanagan and S. N. Freund, ‘‘Atomizer: A dynamic atomicity checker for multithreaded programs,’’ *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 256–267, Jan. 2004, doi: 10.1145/982962.964023.
[75] M. Åsrud. (2017). *A Programming Language for the Internet of Things*. Accessed: Apr. 10, 2021. [Online]. Available: https://www.duo.uio.no/handle/10852/56894
[76] R. S. Engelschall, ‘‘Portable multithreading,’’ in *Proc. USENIX Annu. Tech. Conf.*, San Diego, CA, USA, 2000, pp. 1–12. Accesed: Apr. 10, 2021. [Online]. Available: https://static.usenix.org/event/usenix2000/general/full_papers/engelschall/engelschall.pdf
[77] Oracle. (2012). *Multithreaded Programming Guide*. Accessed: Apr. 11, 20220. [Online]. Available: https://docs.oracle.com/cd/E26502_01/pdf/E35303.pdf
[78] Y. Li, B. Liu, and J. Huang, ‘‘SWORD: A scalable whole program race detector for Java,’’ in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Companion Proc. (ICSE-Companion)*, May 2019, pp. 75–78, doi: 10.1109/ICSE-Companion.2019.00042.
[79] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang, ‘‘Static data race detection for concurrent programs with asynchronous calls,’’ in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. Eur. Softw. Eng. Conf. Found. Softw. Eng. Symp. (ESEC/FSE)*, 2009, pp. 13–22, doi: 10.1145/1595696.1595701.
[80] D. Giebas and R. Wojszczyk, ‘‘Multithreaded application model,’’ in *Proc. Int. Symp. Distrib. Comput. Artif. Intell.* Cham, Switzerland: Springer, 2020, pp. 93–103, doi: 10.1007/978-3-030-23946-6_11.
[81] M. Naik, A. Aiken, and J. Whaley, ‘‘Effective static race detection for Java,’’ in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*. New York, NY, USA: Association Computing Machinery, 2006, pp. 308–319, doi: 10.1145/1133981.1134018.
[82] P. Pratikakis, J. S. Foster, and M. Hicks, ‘‘LOCKSMITH: Practical static race detection for C,’’ *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 1, pp. 1–55, Jan. 2011, doi: 10.1145/1889997.1890000.
[83] M. Ben-Ari, ‘‘Tool presentation: Teaching concurrency and model checking,’’ in *Proc. Int. SPIN Workshop Model Checking Softw.* Cham, Switzerland: Springer, 2009, pp. 6–11, doi: 10.1007/978-3-642-02652-2_5.

DAMIAN GIEBAS received the M.Sc. degree from the Faculty of Electronics and Computer Science, Koszalin University of Technology, where he is currently pursuing the Ph.D. degree with the Faculty of Electronics and Computer Science. Since 2013, he has been working for several software companies. His research interests include the design of parallel performance tools and parallel programming.

RAFAŁ WOJSZCZYK received the Ph.D. degree from the Faculty of Electronics and Computer Science, Koszalin University of Technology, in 2018. He is currently an Assistant Professor with the Faculty of Electronics and Computer Science. His research interests include software engineering, and object oriented designing and programming. Since 2015, he has been a Mentor to a Scientific Club ‘‘.NET Team.’’ Moreover, he taught classes in such subjects as the Internet of Things, python, and Microsoft Kinect.