Deadlocks Detection in Multithreaded Applications Based on Source Code Analysis

Damian Giebas † and Rafał Wojszczyk *,†

Faculty of Electronics and Computer Science, Koszalin University of Technology, Sniadeckich 2, 75-453 Koszalin, Poland; damian.giebas@s.tu.koszalin.pl
* Correspondence: rafal.wojszczyk@tu.koszalin.pl
† These authors contributed equally to this work.

Received: 3 December 2019; Accepted: 7 January 2020; Published: 10 January 2020

Abstract: This paper extends multithreaded application source code model and shows how to using it to detect deadlocks in C language applications. Four known deadlock scenarios from literature can be detected using our model. For every scenario we created theorems and proofs whose fulfillment guarantees the occurrence of deadlocks in multithreaded applications. Paper also contains comparison of multithreaded application source code model and Petri nets and describe advantages and disadvantages both of them.

Keywords: multithreaded applications; resource conflicts; deadlock; pthread

1. Introduction

During the creation of any multithreaded application that uses the pthreads library, there is a possibility that a programmer’s mistake will lead to resource conflicts causing undesirable phenomena, such as race condition, deadlock, atomicity violation, and order violation. Writing the source code of complex programs using the pthreads library is very difficult, and even the best programmers can cause resource conflicts. The article presents the source code of four applications, implementing one of four possible scenarios within which deadlock occurs. Also presented is the source code of the application, which is a leading example showing how to use a graph to locate a conflict causing a deadlock.

The results of research on errors in software indicate that an effective solution for detecting a race condition would be a method based on static code analysis. However, due to technological limitations, currently, the best solutions are based on a combination of static code analysis methods and dynamic analysis methods [1]. Both the phenomenon of atomicity violation and the phenomenon of deadlock are the results of resource conflicts stemming from the lack of application of thread synchronisation mechanisms or their incorrect use, as is the case with the phenomenon of race condition. It is therefore assumed that there is an effective method of static code analysis that can detect resource conflicts causing these phenomena.

Undesirable phenomena in software should be studied not only in high-level object-oriented programming languages such as Java, C#, Python, or Ruby. Due to the high performance and many high-level mechanisms provided in the standard library, C language has become the primary language for programming microcontrollers. The increase in the performance of these devices with simultaneous low power consumption has led to the possibility of running on them not only dedicated applications written in C, but also entire operating systems such as Linux. These systems provide all the mechanisms necessary to write multithreaded applications for microcontrollers. Despite the possibility of running C# and Java virtual machines or Python and Ruby interpreters on these devices, C language is still widely used. In [2] presents the use of Raspberry Pi 3B+ devices to analyse sounds.
in urban environments using software written in Python. Implementation of the program in C using the pthreads library to parallelize some tasks would probably turn out to be faster for this application. However, more difficult implementation, as well as the possibility of challenging to locate resource conflicts that cause, among others, the deadlock phenomenon, probably influenced the creators’ decision to use a language other than C.

1.1. Resource Conflict and Deadlock Definition

The phenomenon of deadlock, like the race condition, the atomicity violation, and the order violation in the literature on the subject, are included in the category of errors specific for multithreaded applications. The listed errors will be listed in this work because the causes for these phenomena should be sought in the way resources are handled by parallel application threads. Resources that are shared by threads must be handled in a specific way using locks, which should also be treated as resources. Thus, incorrect resource handling is referred to as a resource conflict, where a resource conflict should be understood as a subset of structural errors occurring in the source code of a multithreaded application that causes these phenomena. In the context of the deadlock phenomenon, the basis for the conflict is the inability to gain control over the lock, which is necessary to secure the operation on the shared resource against the conflict, causing the phenomenon of race condition. The inability to gain control over the lock usually results from the wrong order of operations performed within the thread, the lack of the lock release operation, or placing the release operation in the wrong place in relation to the operation establishing the lock.

The deadlock phenomenon has been defined as a situation in which several processes wait indefinitely for an event that can only be started by one of the waiting processes [3]. The difference between threads and processes is slight because threads operating within one process share the entire address space and other resources, while processes have their own resources and their own address spaces [4].

In other areas where deadlock also occurs, different varieties of Petri nets (PN) are used to detect them [5]. In addition to PN, which makes deadlock detection possible, also in multithreaded applications [6], a directed graph [3] and Gadara nets are used [7], which are also a type of PN.

In 2014, one of the articles was developed discussing the method of automatic fixing of deadlocks [8]; however, these methods met with considerable criticism [9]. The most important disadvantages of these solutions include the low quality of patches, which results from creating them based on a limited group of tests, not necessarily adequately matched to the code to be fixed [9].

In [10], deadlocks were divided into resource deadlocks and communication deadlocks. Resource deadlocks in programming occur when processes or threads need a particular group of shared resources to work, and each of them has only selected resources, which results in waiting indefinitely [11]. The thread can wait for access to resources that it already has (see Sections 5.3 and 5.4), and this is due to the construction of a mechanism of mutual exclusion (mutex), which can be locked and unlocked.

Communication deadlocks [10] are more abstract and more general than resource deadlocks. They occur in distributed systems, in interprocess communication, as well as in multithreaded applications, in which a cycle of waiting is created, causing deadlocks. This article does not deal with issues related to communication deadlocks; more on this topic can be found in [10,12].

1.2. The Structure of the Article

The next issue is a detailed description of thread synchronisation mechanisms. Section 3 contains a detailed description of the source code model for a multithreaded application, which forms the basis for the remainder of the article. The wording of the deadlock detection problem is provided in Section 4, which is an introduction to the detailed description of the deadlock phenomenon, taking into account the division into conflicts causing this phenomenon. Section 6 contains a sufficient condition that was used to start work on the conflict location method in the source code. This method was used to locate the resource conflict in the lead example described in Section 7. There is also a detailed
description of the method implementation in the authors’ utility program, how it works, as well as the results of the experiment carried out with its help. In Section 8 you can find competitive method’s description. Last section contains conclusions and plans for further research work.

2. Thread Synchronisation Mechanisms

The pthreads library used to write the source code of multithreaded applications provides mechanisms for writing secure applications, provided that the programmer uses them correctly. Among these mechanisms, one can find locks and condition variables. In practice, multithreaded applications always use synchronisation mechanisms.

In addition to locks and condition variables, other mechanisms are not present in the pthreads library. They include, among others, semaphores and monitors described by Edsger Dijkstra [13].

2.1. Locks

The locks are used to implement critical sections, where the critical section should be understood as a fragment of the application code located between calling the lock function and calling the unlock function. The critical section is most often used to perform operations on a shared resource. Locking is performed by three functions, i.e., pthread_mutex_lock, pthread_mutex_trylock and pthread_mutex_timedlock. These functions differ in behaviour when they cannot set a lock when it is already in use. The first of them waits for the lock to be released by the thread that currently occupies it. The second function returns the EBUSY value when the lock fails and allows the thread that called it to continue working. Calling the last function results in waiting for the lock to be occupied until the time specified as the second parameter. If the operation fails, the ETIMEDOUT value is returned, and the thread that was waiting for the lock resumes. Any thread that has created a lock must release it by using the pthread_mutex_unlock function regardless of which function was used to create the lock.

Locks established using the above functions can have one of four types:

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

The lock type has no effect on cross-thread collaboration. However, it has significance within one thread in situations when there is an attempt to re-lock or attempt to release a lock that is not created.

2.2. Condition Variables

Condition variables are the second mechanism provided with the pthreads library. They allow one to force the order of operations by using signals. Thanks to this mechanism, the waiting thread can go into a state that does not load the processor and resume its work when it receives a signal from the thread, whose work had to be done before.

Condition variables always use locks for their work, and the programmer using condition variables is obliged to release them; otherwise, the application may cause serious memory leaks. In the cases discussed in this article, condition variables are not used.

3. Multithreaded Application Source Code Model

The work [14] presents the author’s multithreaded application source code model. Based on that model a theorem was developed, the fulfilment of which means that, in the application, the phenomenon of race condition occurs. For this article, this model was extended and is as follows:

$$ C_P = (T_P, U_P, R_P, O_P, Q_P, F_P) $$

(1)

where:
1. \( P \) is the application index.
2. \( T_P = \{ t_i | i = 0...a \}, (a \in \mathbb{N}) \) is a set of \( t_i \) threads of \( C_P \) application, where \( t_0 \) is the main thread, \(|T_P| > 1\),
3. \( U_P = (u_\beta | \beta = 1...b), (\beta \in \mathbb{N}^+) \) is an \( u_\beta \) sequence of sets that are subsets of \( T_P \) set containing threads working in the same time interval in \( C_P \) application, with \(|U_P| > 2, u_1 = \{ t_0 \} \& u_\beta = \{ t_0 \}\),
4. \( R_P = \{ r_{\gamma | \gamma = 1...n}, r_{\eta | \eta = 1...m} \} \) is a set of shared resources of \( C_P \) application, and the next elements are sets of names of variables related to a single resource,
5. \( O_P = \{ o_{i | i = 1...\delta, j = 1...\epsilon} \}, (\delta, \epsilon \in \mathbb{N}^+) \) is a set of all \( C_P \) application operations which are atomic operations at a certain level of abstraction, i.e., dividing them into smaller operations is impossible. An operation should be understood as an instruction or function defined in a programming language. Index \( i \) indicates the number of the thread in which the operation is performed, and index \( j \) is the ordinal number of operations working within the same thread,
6. \( Q_P = \{ q_{i | i = 1...\kappa}, q_{k} = (w_s, x_s), (\kappa \in \mathbb{N}^+) \} \) – a set of locks available in the program, defined as a pair (mutex variable name \((w_s)\), a type of lock \((x_s)\)), where type is understood as one from the set value (PMN, PME, PMR, PMD),
7. \( F_P = \{ f_n | n = 1...i \} \) \& \( F \subseteq (O_P \times O_P) \cup (O_P \times R_P) \cup (R_P \times O_P) \cup (O_P \times Q_P) \cup (Q_P \times O_P), (i \in \mathbb{N}^+) \) – set of edges including:
   (a) transition edges – specifying the order in which the operation is performed. These edges are pairs \( f_n = (o_{i,j}, o_{j,k}) \), where the elements describe two successive operations \( o_{i,j} \in O_P \),
   (b) usage edges – indicating resources that change during the operation. These edges are \( f_n = (o_{i,j}, r_c) \), in which one element is operation \( o_{i,j} \in O_P \) and the other is resource \( r_c \in R_P \),
   (c) dependency edges – indicating operations depending on the current value of one of the resources. These edges are pairs \( f_n = (r_c, o_{i,j}) \), in which one element is the resource \( r_c \in R_P \) and the other is the operation \( o_{i,j} \in O_P \),
   (d) locking edges – indicating the operation applying the selected lock. These edges are pair \( f_n = (q_k, o_{i,j}) \), in which one element is the lock, and the other is the locking operation,
   (e) unlocking edge – indicating the operation releasing the selected lock. These edges are pairs \( f_n = (o_{i,j}, q_k) \), in which one element is the unlocking operation, and the other is the released lock.

With the model, the concept of \( G_P \) directed graph is associated, which maps relations between operations of threads of the \( T_P \) set and \( R_P \) and \( Q_P \) sets. The \( G_P \) graph definition is as follows:

\[
G_P = (V_P, F_P)
\]  \hspace{1cm} (2)

where:
- \( V_P = Q_P \cup O_P \cup R_P \) should be understood as a set of vertices consisting of \( P \), application operations, \( P \) application locks, and that application’s shared resources.
- \( G_P \) is understood here as a subgraph of \( G_P \) graph, which is built as a result of removing:
  - all vertices of \( O_P \) and their associated edges excluding \( t_i \) thread operation,
  - all vertices of \( Q_P \) set and the associated edges, excluding \( q_i \) vertex,
  - all vertices of \( R_P \) set and their associated edges, excluding the edges between \( t_i \) thread operations and the resources they use,
  - transition edges \((o_{i,j}, o_{j,k})\) for \( o_{i,j} \) vertex, when there is an unlocking edge for this vertex \((o_{i,j}, q_k)\).

Compared to [14], the model was extended by \( Q_P \) set, which is the set of all locks defined by the programmer. This allows locks to be treated as a special type of resource that can be created and released.
Another significant change is the definition of the elements of the $R$ resource set. C language has a mechanism of indicators with which one can access the selected memory area in which the resource is located. Therefore, it is reasonable to store in the model all possible variable names referring to the same resource.

Two new edge types were also introduced to the model: the locking edge and the unlocking edge. The first type leads from the lock to the operation, and the second type leads from the operation to the lock to be released. This solution is a more faithful representation of what is happening in the application code than the solution presented in previous works on the source code model of a multithreaded application.

In the graphic representation, the frames surrounding the operations will no longer be seen. In further works, the locking and unlocking operations will consist of operation symbols, edges, and a lock symbol. A separate symbol is assigned to each of the three types of locks, i.e., PMN is a triangle symbol, PME is a diamond symbol, PMR is a trapezoid symbol. In the implementation of the pthreads library for Linux, the PMD type is an alias of the PMN type; therefore, in this paper, the PMD type will be synonymous with PMN. If another implementation is used, it should be clearly stated how PMD is represented.

4. Problem Definition

There is a multithreaded $P$ application written in C using the pthreads library. There is a deadlock phenomenon in this application. Is it possible to detect a conflict causing the application deadlock using the model from Section 3?

An example will be the applications that implement the scenarios described in Section 5.

5. Deadlock

For a deadlock to occur, four conditions must be met. The conditions can be found in operating system manuals [3] as well as in scientific papers [15]:

1. Mutual exclusion: At least one resource must be non-shareable; that is, only one process can use this resource at a time. If another process requests access to the resource, it must be delayed until the resource is released.
2. Hold and wait: There must be a process that has been allocated at least one resource, and that is awaiting the allocation of an additional resource that is being held by another process.
3. No preemption: Resources are not subject to preemption, which means that a resource can only be released on the initiative of the holding process after the process has ended.
4. Circular wait: There must be set $P_0, P_1, \ldots, P_n$ of waiting processes, such that $P_0$ is waiting for the resource held by $P_1$. $P_1$ is waiting for the resource held by $P_2$, $P_2$ is waiting for the resource held by $P_3$, and $P_n$ waiting for the resource held by $P_0$ process.

The above necessary condition has the annotation that all four conditions must be met for a deadlock to occur. It was also noted that the circular wait condition implies the hold and wait condition, so these four conditions are not completely independent.

The deadlock phenomenon very often occurs as a result of the programmer’s actions when he tries to prevent resource conflicts that cause the phenomena of a race condition and atomicity violation [16,17].

The word deadlock is most often understood as a resource deadlock. There are four scenarios in which the phenomenon of deadlock may occur through (sources can be found at https://github.com/PKPhdDG/deadlock_examples):

1. mutually exclusive lock pairs (application implementing S1 scenario, let it be called DL0),
2. missing the unlocking operation, e.g., as a result of a control instruction (application implementing the S2 scenario, let it be called DL1),
3. retrying the lock as a result of:
   
   (a) loop operation (application implementing the S3 scenario, let it be called DL2),
   (b) recursive function call (application implementing the S4 scenario, let it be called DL3).

The condition of the deadlock phenomenon in the scenario implemented in the DL0 application code is the appropriate point in time. If the operating system scheduler runs threads at the same time (or the time difference between the runs is insignificant) and both threads put locks at the same time, mutual waiting occurs, and deadlock occurs.

The condition of deadlock in the source code of DL1 and DL2 applications are the appropriate input data. It depends on the values given by the user, how the application will look like, and whether these values will lead to the fulfillment of the condition, which in turn will lead to deadlock.

A deadlock in the DL3 application always occurs when the lock used in the recursively called code is of a different type than PMR. Figures 1 and 2 present four scenarios in which the phenomenon of deadlock may occur. Scenarios from Figure 1 were considered in several papers [16,17], including those regarding the automatic fixing of errors related to atomicity violation. The AFix and Axis tools described there can automatically remove the atomicity violation without introducing new conflicts that could lead to a deadlock.

**Figure 1.** Deadlock scenarios using one lock. Source [16].

**Figure 2.** Scenario 1 (S1): deadlocks due to mutually exclusive locking pairs.
The proposed source code model of a multithreaded application is designed to enable detection of resource conflicts that cause these phenomena faster than other methods, by performing only static source code analysis. This approach is faster than the method used in the CTrigger tool. This tool carries out the complicated process of extracting pieces of code executed in parallel, and then repeatedly tests these fragments with various parameters [18].

5.1. Gadara Project

One of the solutions that make the elimination of deadlock possible is a project called Gadara [7,19,20]. This project focuses on placing a supervisor in a multithreaded application during the source code compilation process, within which an application state machine is also created using Petri Nets. The created PN is then used by the supervisor to force the application to work correctly. The way the supervisor works is based on a discrete event dynamic system.

The supervisor solution developed under the Gadara project is undoubtedly an outstanding achievement in the field of multithreaded applications. However, this solution, like every other, has several disadvantages. The supervisor implementation may have its own errors that may affect the operation of the supervised application. As a result, additional elements may affect the application in an unpredictable manner, which may result in failures as serious as the deadlock.

An additional supervisor in a multithreaded application also means more memory and processor time usage, 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 that of the Gadara project and then used by an unaware user.

5.2. DL0 Application Model

According to the model in Section 3, the source code of DL0 application is as follows:

\[
\begin{align*}
T_{DL0} &= \{t_0, t_1, t_2\} \\
U_{DL0} &= \{\{t_0\}, \{t_1, t_2\}, \{t_0, t_1\}\} \\
R_{DL0} &= \{\text{counter}\} \\
O_{DL0} &= \{0_{1,1}, 0_{1,2}, 0_{1,3}, 0_{1,4}, 0_{1,5}, 0_{1,6}, 0_{2,1}, 0_{2,2}, 0_{2,3}, 0_{2,4}, 0_{2,5}, 0_{2,6}\} \\
Q_{DL0} &= \{(m, PMD), (n, PMD)\} \\
F_{DL0} &= \{\{(0_{0,1}, 0_{0,2}), (r_1, 0_{0,2})\}
\end{align*}
\]

The phenomenon of deadlock in the DL0 application is caused by the work of a pair of threads using a pair of locks to exclude each other. In large multithreaded applications, this happens very often because in order to achieve maximum performance, while maintaining the correctness of the application code, one shared resource usually has one lock. According to the above, there is a very high probability that there will be threads working with a group of shared resources, and thus also with a group of locks. As a result, in all of these threads, the order of establishing lock groups must be identical because a different order leads to the deadlock phenomenon. It is possible, however, that the programmer, through the use of C language structures or intentionally through the use of specialised tools, will obfuscate the code, and the resource conflict causing this phenomenon will be hidden from him.

The resource conflict in the case of the S1 scenario is a different order of establishing lock pairs in two threads. Code obfuscation should be understood as the lack of using standard rules for writing the application code, e.g., MISRA C, Netrino, or GNU coding standard.

A graph of the DL0 application source code operation is shown in Figure 3. This representation locates the resource conflict that causes deadlock. This conflict in the graph manifests itself by the crossing of the edges leading from the lock to the operation in thread \(t_2\), while the edges leading from
the locks to the operation in \( t_1 \) thread do not cross. The crossing of the edges is the effect of a different order of locking.

**Figure 3.** DL0 application operation graph.

Based on the scenario presented in the form of the DL0 application source code and operation graph, it can be concluded that:

**Definition 1.** Let \( \lambda^{P,i} \), \( i \) be the set of all paths that can be created from \( t_1 \) thread operation beginning with operation \( o_{i,j} \). Formally, the \( i \)th path of \( \lambda^{P,i} \) set is defined as the sequence:

\[
\lambda^{P,i}_a,1 = (\lambda^{P,i}_a,2, \ldots, \lambda^{P,i}_a,q)
\]

where \( \lambda^{P,i}_a,k \neq \lambda^{P,i}_a,l \) for \( k \neq l \).

**Definition 2.** Let in \( \lambda^{P,i}_l \) path be two such operations \( o_{i,a} \) and \( o_{i,b} \) where locks \( q_c \) and \( q_d \). Lock \( q_c \) is said to precede lock \( q_d \) in path \( \lambda^{P,i}_l \), if \( a \prec b \). This relation is denoted \( q_c \prec_l q_d \).

**Lemma 1.** Two threads \( t_i \) and \( t_j \) are given in the source code of \( P \) application, and path sets \( \lambda^{P,i} \) are known. In both threads, there are operations putting locks \( q_c \) and \( q_d \). If there is such a pair of paths: \( \lambda^{P,i}_k \), \( \lambda^{P,j}_k \) for which the following condition is met:

\[
(q_c \prec_l q_d) \land (q_d \prec_k q_c)
\]

then in \( P \) application source code there is a resource conflict that will cause a deadlock phenomenon with \( t_i \) and \( t_j \) threads.

**Proof.** According to Figure 1, the necessary condition for the occurrence of a deadlock of type S1 (exclusion of a pair of mutexes) between two threads \( t_i \) and \( t_j \) is the existence of a operations sequence to establish \( q_c \) and \( q_d \) locks in reverse order in each of these threads. This condition is met when the \( q_c \) lock in the \( t_i \) thread precedes the \( q_b \) lock (according to def. 2) and \( q_b \) lock in thread \( t_i \) precedes the \( q_c \) lock or vice versa (\( q_c \prec_l q_d \) \( \land q_d \prec_k q_c \)).

In summary, static code analysis allowed one to locate the S1 scenario causing a deadlock phenomenon.

5.2.1. DL1 Application Model

The resource conflict causing a deadlock in DL1 is more challenging to detect than the one causing a deadlock in DL0. This is due to the use of the control instruction, which introduces the possibility of the thread to operate in two different ways, depending on the condition used in this instruction. Figure 4 presents a graph of the DL1 application operation, the model of which is presented in a further
part of the article. This model is much more complex than the DL0 application model. It contains many more edges and has branching resulting from the use of the if–else control instruction.

Figure 4. DL1 application operation graph.

The resource conflict in the source code of the DL1 application is therefore manifested by the presence of the path in the operation graph, which includes the locking operation, while the unlocking operation is absent in the path. As a result, there is no fragment in the path that is the cycle responsible for establishing the lock and releasing it.

According to the model in Section 3, the source code of DL1 application is as follows:

\[
T_{DL1} = \{t_0, t_1, t_2\},
\]
\[
U_{DL1} = \{\{t_0\}, \{t_1, t_2\}, \{t_0\}\},
\]
\[
R_{DL1} = \{\{counter\}, \{i, t2arg\}\},
\]
\[
O_{DL1} = \{o_{0,0}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{2,1}, o_{2,2}, o_{2,3}, o_{2,4}, o_{2,5}, o_{2,6}, o_{2,7}\},
\]
\[
Q_{DL1} = \{\{m, PMD\}\},
\]
\[
F_{DL0} = \{(o_{0,1}, o_{0,2}), (r_1, o_{0,2}), (o_{0,2}, o_{0,3}), (o_{0,3}, o_{0,4}), (r_1, o_{0,4})\},
\]
\[
F_{DL2} = \{(o_{0,4}, o_{0,5}), (q_1, o_{1,1}), (o_{1,1}, o_{1,2}), (o_{1,2}, r_1), (o_{1,2}, o_{1,3}), (o_{1,3}, q_1), (o_{1,3}, q_1), (o_{2,1}, o_{2,2}), (o_{2,2}, o_{2,3}), (r_2, o_{2,3}), (o_{2,3}, o_{2,4}), (o_{2,4}, r_1), (o_{2,4}, o_{2,5}), (o_{2,5}, q_1), (o_{2,5}, o_{2,7}), (o_{2,5}, o_{2,6}), (o_{2,6}, r_1), (o_{2,6}, o_{2,7})\}.
\]

Very complicated expressions in control instructions can lead to the situation that even a correctly placed unlocking operation will never be performed, because all the necessary conditions for the control expression will not be met. There is also a chance that the conditions of the control instruction will be mutually exclusive as a result of the programmer’s mistake, which may also fail to call the unlocking operation. Therefore, it seems good practice to place locking and unlocking operations out of the body of all types of control instructions, and when that is not possible, try to place unlocking operations in the same block of code as the locking operations.

To sum up, in order for the deadlock phenomenon presented in the DL1 application to take place, one should find in the graph such a path, whose part is not the cycle consisting of the locking and unlocking operation. Cases in which the unlocking operation is not called by the erroneous conditions of the control instructions are not the subject of this research.

5.3. DL2 Application Model

The reason for the deadlock phenomenon in the DL2 application is very similar to that in the DL1 application. Namely, the locking operation is in the loop body block, while the unlocking operation is outside the loop. As a result, no more than one thread is required for a resource conflict that causes a deadlock phenomenon, because the thread is waiting for itself due to a programmer error. Figure 5
presents a graph of the DL2 application operation, the model of which is presented in a further part of the article.

According to the model in Section 3, the source code of DL2 application is as follows:

\[
T_{DL2} = (t_0, t_1), \\
U_{DL2} = \{(t_0), \{t_1\}, \{t_0\}\}, \\
R_{DL2} = \{\{\text{counter}\}, \{i, t2arg\}\}, \\
O_{DL2} = \{o_{0,1}, o_{0,2}, o_{0,3}, o_{0,4}, o_{0,5}, o_{1,1}, o_{1,2}, o_{1,3}, o_{1,4}, o_{1,5}, o_{1,6}\}, \\
Q_{DL2} = \{(m, PMD)\}, \\
F_{DL2} = \{(o_{0,1}, o_{0,2}), (r_1, o_{0,2}), (o_{0,2}, o_{0,3}), (o_{0,3}, o_{0,4}), (o_{0,4}, o_{0,5}), (o_{1,1}, o_{1,2}), (r_2, o_{1,2}), (o_{1,2}, o_{1,3}), (o_{1,3}, o_{1,4}), (o_{1,4}, o_{1,5}), (o_{1,5}, o_{1,6})\}.
\]

Figure 5. DL2 application operation graph.

Resource conflict in the DL2 application will occur when the appropriate input parameter is given. The parameter value will cause the while loop to be executed more than once, and an attempt will be made to re-lock, as a result of which the thread will stop its operation and begin to wait indefinitely for itself. In the DL2 application, the resource conflict causing the deadlock phenomenon is manifested by attempting to re-establish the existing lock. Detection of this conflict in the graphic representation boils down to finding such a cycle in which locking operation is present, while it does not contain the unlocking operation.

The next lemmas, therefore, read as follows and apply to both DL1 and DL2 applications.

**Lemma 2.** If in the \(^s_i\)G\(_P\) subgraph of \(G\(_P\) graph of P application (subgraph for thread \(t_i\) and \(q_s\) lock), there is a path that starts with \(q_s\) lock and is not the cycle, then the phenomenon of deadlock type S2 and S3 occurs in P application.

**Proof.** The necessary condition for the occurrence of deadlocks of type S2 and S3 in thread \(t_i\) is the existence of a operations sequence, resulting in the failure to unlock a pre-established \(q_s\) lock. This condition is met when there is a path in the \(^s_i\)G\(_P\) subgraph that starts with \(q_s\) (establish a lock) and unrecurrent to \(q_s\) (release of lock)—a path that is not a cycle. 

5.4. DL3 Application Model

The last scenario considered in [16] refers to the code in which the recursively called function attempts to perform the locking operation every time, which causes a deadlock phenomenon. Figure 6 presents a graph of the DL3 application operation. The sample DL3 application implements the S4 scenario, according to the model in Section 3, the source code of DL3 application is as follows:
Like in the DL2 application, also in the DL3 application, there is a deadlock phenomenon, which is the result of the circular nature of the operation. In the DL2 application, the deadlock was due to incorrect placement of locking and unlocking operations, while in the case of the DL3 application, the operations are correctly placed. The difference is that in the case of recursive calls, one should use the pthread_mutex_lock function with the PMR type lock. This type guarantees that if the recursive function tries to re-lock, the thread ID will be checked, and the internal lock counter will be incremented, which will allow further operation.

Recursion in the operation graph is manifested by the presence of a path that leads from the last operation to the first. This path may consist of one edge, but it is not a rule, as it is the case in the DL3 application. In the case of recursive functions, it should be borne in mind that there is always a run that ends at the last operation or an operation from which no other edges are derived. Therefore, in this case, to detect a resource conflict causing a deadlock, one must determine whether the thread is calling the function recursively and whether it is using a PMR type lock. If the lock type is different, a deadlock will occur. So the third lemma is as follows.

**Lemma 3.** If \( t_i \) thread of P application calls a recursive function that does not use PMR type locks, then the deadlock phenomenon will occur.

**Proof.** Lemma is a direct consequence of the assumptions of the invocation of recursive functions.

6. Sufficient Condition

The application examples described above present four scenarios in which resource conflicts cause the deadlock phenomenon. Based on the examples described in Section 5 and the lemmas developed based on these examples, it can be concluded that the phenomenon of deadlock occurs when:

1. there is a pair of threads setting the same pair of locks, with the order in which these locks are set is different in both threads (Lemma 1),
2. in the operation graph involving \( t_i \) thread, there exists a path that starts with \( q_i \) and is not a cycle (Lemma 2),
3. the thread calls the recursive function, in which the PMR type lock represented by the trapezium is not used (Lemma 3).

The above causes of the deadlock phenomenon in multithreaded applications can be generalized to the following theorem.

**Theorem 1.** Let \( T_P = \{ t_0, ..., t_\alpha \} \) mean a set of threads using \( Q_P = (q_1, ..., q_\kappa) \) locks of \( P \) application.

The S1-S3 deadlock phenomenon occurs if there is such a pair of threads \( t_i, t_j \in T_P \) for which lemma 1 is met, or there is a thread \( t_k \in T_P \) for which any of Lemma 2 and 3 is fulfilled.

**Proof.** The Theorem is a direct consequence of Lemmas 1–3. □

7. Using the Source Code Model of the Multithreaded Application in Practice

7.1. Leading Example

A system where it is essential to locate the resource conflict causing the deadlock phenomenon can be an application in which users define tasks to be performed by the system based on date and time. Let this system be able to take non-standard actions if one of the predefined dates occurs. Figure 7 shows a block diagram of an algorithm describing the operation of such an application, and its input data provided by the user will be a sequence with the following structure, compatible with the cron program:

1. minute,
2. hour,
3. day of the month,
4. month,
5. day of week,
6. user,
7. job.

![Block Diagram](image_url)

**Figure 7.** A simplified block diagram of a sample application algorithm.

Testing such applications is very difficult and time-consuming. The sample application runs tasks with an accuracy of one minute. To test how such an application works, one needs to perform tests
for each scheduled minute of the year, which, assuming it’s a leap year, gives 527,040 combinations. After taking into account the days of the week, the number of combinations increases to 648,000. Depending on how complex the application code is, it may not be possible to test all possible combinations. Assuming that it will take 0.5 ms to perform one test, testing all possible combinations will take approximately 5 min and 24 s. Tools for detecting undesirable phenomena in multithreaded applications, including the CTrigger tool focused on the phenomenon of atomicity violation, are based on performing tests on a selected piece of code that potentially has a resource conflict causing this phenomenon [18,21,22]. As a result, these types of tools do the same work as load tests, but the former is more efficient. This is because the whole code is not tested, but only parts of it. Further in the article, it is assumed that the solution based on the source code model of a multithreaded application will be compared with a tool similar to CTrigger, however, focused on detecting resource conflicts causing deadlocks. For the sake of simplicity, let the proposed solution be called deadlock detector (DD).

Figure 8 has an operation graph that corresponds to a fragment of the application described above; i.e., it runs a given number times two tasks. Let it be called API1 (source code: http://bit.ly/2ZBqhrO). If the task data meet the start condition, then the task is run. If the condition is not met, the application does not take any action. Running tasks is displaying data to the standard output of the application, which is a shared resource.

![Figure 8. API application operation graph.](image)

The sample application has an error resulting from skipping the unlocking operation in one of the control instructions branches. In the figure showing the algorithm of operation, this error is located in the block “Perform Other Task”. Therefore, let the application described above, whose operation graph is below, be an example to which the rest of the article will refer.

\[
T_{API} = \{t_0,t_1,t_2\}, \\
U_{API} = \{\{t_0\}, \{t_1,t_2\}, \{t_0\}\}, \\
R_{API} = \{\{\text{std}::\text{cout}\}, \{\text{std}::\text{cerr}\}\}, \\
O_{API} = \{0_{0,1},0_{0,2},0_{0,3},0_{0,4},0_{0,5},0_{1,1}, \}
\]

\[
0_{1,2},0_{1,3},0_{1,4},0_{1,5},0_{1,6},0_{1,7},0_{1,8},0_{1,9},0_{1,10},0_{1,11},0_{1,12},0_{1,13},0_{1,14}, \}
\]

\[
0_{1,16},0_{1,17},0_{1,18},0_{2,1},0_{2,2},0_{2,3},0_{2,4},0_{2,5},0_{2,6},0_{2,7},0_{2,8},0_{2,9},0_{2,10},0_{2,11}, \}
\]

\[
0_{2,12},0_{2,13},0_{2,14},0_{2,15},0_{2,16},0_{2,17},0_{2,18}, \}
\]

\[
Q_{API} = \{(m, PMD)\}, \\
F_{API} = \{(0_{0,1},0_{0,2}), (0_{0,2},0_{0,3}), \}
\]

\[
(0_{0,3},0_{0,4}), (0_{0,4},0_{0,5}), (0_{0,5},r_1), \}
\]

\[
(0_{0,5},0_{0,6}), (0_{0,6},0_{0,7}), (0_{0,7},0_{0,8}), \}
\]

\[
(0_{0,8},0_{0,9}), (0_{0,9},0_{1,0}), (0_{1,0},0_{1,7}), \}
\]

\[
(0_{1,0},r_1), (0_{1,0},0_{11}), (0_{1,1},0_{1,2}), \}
\]

\[
(0_{1,1},0_{1,2}), (0_{1,2},0_{1,3}), (0_{1,3},0_{1,4}), \}
\]

\[
(0_{1,4},0_{1,5}), (0_{1,5},0_{1,6}), \}
\]

\[
(0_{1,6},0_{1,7}), (0_{1,7},0_{1,8}), (0_{1,8},0_{1,9}), \}
\]

\[
(0_{1,9},0_{1,10}), (0_{1,10},0_{1,11}), (0_{1,10},0_{1,12}), \}
\]

\[
(0_{1,12},0_{1,1}), (0_{1,12},0_{1,18}), (0_{1,10},0_{1,13}), \}
\]

\[
(0_{1,13},0_{1,14}), (0_{1,14},0_{1,15}), (0_{1,15},r_1), \}
\]

\[
(0_{1,15},0_{1,18}), (0_{1,15},0_{1,16}), (0_{1,16},0_{1,17}), \}
\]

\[
(0_{1,17},0_{1,18}), (0_{1,18},0_{2,1},0_{2,2}), (0_{2,1},0_{2,2}), \}
\]

\[
(0_{2,2},0_{2,3}), (0_{2,3},0_{2,4}), (0_{2,4},0_{2,5}), \}
\]

\[
(0_{1},0_{2,5}), (0_{2,5},0_{2,6}), (0_{2,6},0_{2,7}), \}
\]

\[
(0_{2,7},0_{2,8}), (0_{2,8},0_{2,9}), (0_{2,9},0_{2,10}), \}
\]

\[
(0_{2,10},0_{2,11}), (0_{2,10},0_{2,12}), (0_{2,12},0_{1,1}), \}
\]

\[
(0_{2,12},0_{2,18}), (0_{2,10},0_{2,13}), (0_{2,13},0_{2,14}), \}
\]

\[
(0_{2,14},0_{2,15}), (0_{2,15},r_1), (0_{2,15},0_{2,18}), \}
\]

\[
(0_{2,18},0_{2,16}), (0_{2,16},0_{2,17}), (0_{2,17},0_{2,18}) \}. \]
7.2. Deadlock Detection Using Petri Nets

The deadlock phenomenon can be located by PN, and the AP1 application net is shown in Figure 9. This net is very extensive and has many more elements than the operation graph from Figure 8. Also, not all elements included in the operation graph from Figure 8 are in the net shown. Shared resources were omitted. Representation of the AP1 application in the form of PN is also not fully compatible with the representation that would be created if all the principles described in work [23] were applied. However, PN simplification does not adversely affect the process of locating the resource conflict, causing the deadlock. The status of tokens in Figure 9 corresponds to the situation where the thread \( t_2 \) finished its operation without releasing the lock. The token in place \( P_{1,5} \), which corresponds to operation \( o_{1,5} \), will not move further because the lock was released, which should be manifested by the return of one of the tokens from place \( P_{2,15} \) to place \( P_{18} \).

![Figure 9. Petri net for the leading example.](image)

The further simulation will move the token from \( P_{2,15} \) to \( P_{2,18} \) and will lock the entire net. This shift may suggest that operation \( o_{2,18} \), which of course is not true. However, after this shift, no further simulation is possible, and the presence of the token in place \( P_{1,5} \) indicates deadlocked thread \( t_1 \).

The disadvantage of using PN to locate deadlocks is the need to simulate application work. These simulations can be time-consuming for very large applications. It is also worth noting that the simplified PN for the AP1 application has many more elements than the graphical representation of the application in accordance with the source code model of the multithreaded application. Analysis of PN states also seems to be more complicated than analysis of operation graphs. In the net presenting applications with resource conflict, some states are very difficult to interpret.
7.3. Deadlock Phenomenon Detection Using a Multi-Thread Application Source Code Model

In the leading example, the same situation as in the S2 scenario occurs. Figure 8 shows a graph of the leading example operation. Therefore, to locate the resource conflict, one must follow Lemma 2.

7.4. Implementation of the Method

To locate a conflict that causes a deadlock using the source code model of a multithreaded application, the application must be represented as an operation graph. An example implementation of the method that allows one to detect deadlocks caused by bypassing the release of the lock in conditional instructions has been published on the GitHub portal (http://bit.ly/2IsK7zz) in the resource_conflict_detector repository. Python was used to implement the method, and the created application takes, as a parameter, the path to the directory where the code of the multithreaded application written using the pthreads library is located. In the further part of the article, the application with the method implementation will be referred to as resource conflict detector (RCD). RCD first divides the multithreaded application into functions. This division facilitates the conversion of code to the multi-thread application source code model. These functions are analysed to detect the presence of threads in the application. The next step is to assign threads to the appropriate time intervals. Having a set of threads and time intervals, it is possible to determine which of the operations performed by threads are protected by locks. These operations create a tree structure, which includes, among other things, locking and unlocking operations. Therefore, to locate the resource conflict causing the deadlock, one must find a sufficient condition for it to occur.

The current implementation has many limitations. The first is the assumption that there is a single operation on one line of the source code. Placing several operations on one line may result in skipping the operation, which will affect the process of locating the resource conflict. The application also cannot recognise locks. Therefore, a deadlock resulting from an attempt to re-establish an already established lock in the same thread will not be reported. The application’s behaviour will be similar when a lock other than the one established is released.

Some of the current limitations can be eliminated by using abstract syntax trees (AST) already described in the 1970s in “Principles of Compiler Design” [24]. Their use in the considered case would be redundant because a lot of information present in the application code is omitted as they do not affect the described phenomenon. In addition, some AST expressions of programming language have identical forms, which can lead to error reporting in places where errors are not found. This situation is called “false-positive error” and was described, among others, in work on the automated refactoring tool called CoderAider [25].

7.5. Experiment Assumptions

The previous section describes the capabilities and limitations of the program that uses the multi-thread application source code model to analyse the AP1 application described in Section 7.1. AP1 application meets the requirements of RCD so that it can be analysed without any changes. Also, the AP1 application does not perform real tasks and instead displays on the standard output only information about the task and the iteration in which the task was run.

Tools such as DD require several steps to determine if there is a conflict in the application code that causes unwanted phenomena. For the sake of simplicity, it is assumed that the application being tested is a fragment that the DD tool would extract from a large application. Additionally, the experiment omits the time of code conversion to the representation from which information about the conflict location is taken. As a result, the time of code analysis in the RCD application will be compared with the time of calls of all possible combinations of input data that the DD program would have to perform. It is assumed that a complete review should be carried out using the DD tool to be 100% sure that the application is free of deadlocks. In the event of a partial review, there is a chance that the data set will not contain values that could lead to a deadlock phenomenon.
The application’s running time will be measured using the time tool in the bash console. This tool adopts the program name and its parameters as parameters, and after finishing the work, the time tool displays the running time of the tested program. The experiment was carried out on a computer with the following properties:

- Processor: AMD Ryzen 5 1500X, 3.5 GHz,
- RAM: Corsair Vengeance LPX, DDR3, 32GB, 3000MHz, CL15,
- Hard Drive: SSD Intenso 128GB SATA3, read speed 520 MB/s.

The computer was running under the control of Linux Mint 19 Tara with the 4.15.0 kernel. The GCC 7.4.0 compiler was used to compile the AP1 application.

7.6. Results

During the simulation, the input data of the AP1 application, i.e., the number of tests that the DD application would carry out, must be taken from the user. Therefore, to minimize the user’s working time to a minimum, these data are downloaded from the data.in file, which has the value 648,000.

The screenshot shown in Figure 10 contains the compilation process as well as the running time of the tested application. The time tool returns three time values. For this test, the essential value is real, which shows the clock time of the application running from its launch to its completion. The value returned by the time tool is 2 min and 21 s. This value, however, is much higher than the time of reviewing the source code by RCD.

RCD running time is shown in the screenshot in Figure 11. Its value is approximately 0.06 s. The source code review was 2350 times faster than the complete review. This is a huge difference because the tested application has approximately 200 lines of source code. It is also worth considering the way the RCD works. The application is relatively simple, and at the time the article was written, it was able to detect only deadlocks caused by the lack of calling the pthread_mutex_unlock function. It should also be noted that as a result of RCD operation, the “false-positive error” was indicated in two places. It turns out that the code fragment with the condition that checks whether a lock was successfully created meets the condition of a resource conflict causing the deadlock.

Along with the further development of RCD, the application will have to be enriched with further functions that will affect the application’s running time. The model implemented for application analysis will be more accurate, which will affect the amount of memory that will be used by the application.
application. More important, however, is that the application will do its work longer and longer. The operation graph review to locate the resource conflict in the API1 application took 0.06 s. Section 5 provides four reasons for a deadlock. Assuming that the time to locate resource conflicts causing deadlocks in the code will be the same for all four reasons, the same location of deadlocks in the API1 application should take about 0.24 s. If the time of locating resource conflicts causing each of the undesirable phenomena described in work [14] is close to the time of locating deadlocks, it should take about 1 s to review the application code. As a result, code analysis would be faster, only 141 times.

Figure 11. Screenshot of the console window with resource conflict detector (RCD) time to locate the deadlock in the API1 application.

8. Competitive Method

Other well described static analysis method comes from article [26]. Authors of this paper was restricted to C language and pthreads library also. Their method basis on transformation from C language into a graph representation named interprocedural control flow automaton (ICFA) [26]. Control flow automaton is very similar to control flow graph. Developed in this article method has a lot of common elements with ICFA based method but there are two big differences. To use ICFA there is need to put changes into source code, because programmers write code in many different styles e.g., few return statements in function instead of one and the other case is need to remove pointers. This process was automated but it take a lot of time and has a problem with detection of dependencies. The second difference is lack of time as a variable in ICFA, which is one of most important variable in method described in this paper.

Implementation of ICFA showed there is huge problem with memory consumption when using this method. On the computer with 24 GB RAM for 138 from 997 of them process finished with out of memory error. Even today more than 16 GB RAM computers are not standard which is restriction for this method. Our method does not require so much memory. We study only threads in a single time unit, which reduce memory consumption in our method.

There are also other analysis methods [27] aimed at detecting deadlocks in a chosen group of languages and frameworks, which are modified to use mentioned method. Our attitude is targeted on the model which allow detect not only deadlocks but also race condition, order violation and
atomicity violation. In this field ICFA based method looks like more mature than ours, but high memory consumption is the cause this method cannot be applied into real-time tool e.g., IDE plugin.

9. Conclusions

This article describes four scenarios in which a resource conflict causes the deadlock phenomenon, and a sufficient condition is presented, the fulfilment of which results in the deadlock phenomenon. Our contribution in this field is the the method which allow to detect described scenarios by using multithreaded application source code model. Then, the RCD application was developed, which allows one to locate a single resource conflict in the leading example, which was the API application. The results of the experiment show that the analysis of the source code turned out to be much faster than the complete review based on testing the selected code. This is due to the fact that the complete review repeatedly performed the algorithm, in contrast to the algorithm used in RCD, which performs much less such review operations. It is worth noting, however, that this review would only theoretically be carried out by the DD tool. There is a chance that the data used to test the application would not have values that could cause a deadlock phenomenon, but there is also a chance that a complete review should not be performed to find a deadlock.

The RCD application described in this article is not yet complete, and its fast operation time results from the use of a simplified source code model of a multithreaded application, as well as locating only one type of resource conflicts causing a deadlock. However, there are indications that this tool will be more efficient than tools based on the process of testing the source code.

The use of AST should be considered in further work on the RCD. The author of the work [25] uses AST in the method of automated source code refactoring. His method is based on finding specific patterns in AST for code written in Java and C++ and then correcting that code. It is, therefore, possible that the use of AST, although introducing some redundancy, will allow one to implement a method that uses the source code model of a multithreaded application for compilers or IDE to detect phenomena such as deadlocks automatically. Unfortunately, the algorithm used in RCD also showed a "false-positive error", which indicates its imperfection. The popularity and multitude of ready-made code conversion tools also induce some bias in favour of AST.

Currently, large-scale research on locating undesirable phenomena in multithreaded applications is not conducted due to the high level of complexity of the issues, or research is limited to object-oriented languages, with Java as the most often chosen. This is because Java as a language is limited to only one programming paradigm, and there is no revolution in it, as is the case in C++. As a result, analysis of Java code is much simpler than other languages, such as C. Additionally, the analysis of Java binary files (as well as other languages whose binary files are run on virtual machines) is very simple, because the resulting file is not affected by the environment in which file is being built or the environment in which the file will be run. As a result, the analysis of a running application is simplified compared to applications written in C, which can be compiled with many different compilers for many different operating systems, which of course, results in many different binary files.

Further research work should also take into account the technique described in the published patent regarding a solution that guarantees the location of resource conflicts, causing deadlock [28]. This patent contains information about a hybrid method of locating resource conflicts causing the deadlock phenomenon. The combination of static code analysis and dynamic technique that tracks application threads made the development of an efficient and universal method possible. By analysing the source code, a set of potential conflicts in the code is created that can lead to a deadlock. Then, using the application tracking technique, these conflicts are verified. This approach, according to the creators, guarantees a negligible number of "false-positive error" reports.

Author Contributions: All authors contributed to the research work. D.G. proposed the main idea. G.D. and R.W. conceived the multithreaded application model, wrote the article together. G.D. implemented software tools and made experiments. R.W. reviewed the research work. All authors worked together and responded to the reviewers comments. All authors have read and agreed to the published version of the manuscript.
**Funding:** This research received no external funding.

**Conflicts of Interest:** The authors declare no conflict of interest.

**Abbreviations**

The following abbreviations are used in this manuscript:

- **AST** Abstract Syntax Trees
- **DD** Deadlock Detector
- **DL0** Example application implementing S1 scenario
- **DL1** Example application implementing S2 scenario
- **DL2** Example application implementing S3 scenario
- **DL3** Example application implementing S4 scenario
- **PMD** Pthreads Mutex Default
- **PME** Pthreads Mutex Error Check
- **PMN** Pthreads Mutex Normal
- **PMR** Pthreads Mutex Recursive
- **PN** Petri Net
- **RCS** Resource Conflict Detector
- **S1** Deadlock scenario caused by mutually exclusive locking pairs
- **S2** Deadlock scenario with missing unlock
- **S3** Deadlock scenario with double locking the same lock
- **S4** Deadlock scenario with double locking during calling recursive functions

**References**

1. Savage, S.; Burrows, M.; Nelson, G.; Sobalvarro, P.; Anderson, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* 1997, 15, 391–411. [CrossRef]

2. Silva, B.D.; Happi, A.W.; Braeken, A.; Touhafi, A. Evaluation of Classical Machine Learning Techniques towards Urban Sound Recognition on Embedded Systems. *Appl. Sci.* 2019, 9, 3885. [CrossRef]

3. Silberschatz, A.; Galvin, P.B.; Gagne, G. *Operating System Concepts*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2013.

4. Mitchell, M.; Oldham, J.; Samuel, A. *Advanced Linux Programming*; New Riders Publishing: Indianapolis, IN, USA, 2001.

5. Lefebvre, D. Dynamical Scheduling and Robust Control in Uncertain Environments with Petri Nets for DESs. *Processes* 2017, 5, 54. [CrossRef]

6. Lautenbach, K.; Schmid, H.A. Use of Petri Nets for Proving Correctness of Concurrent Process Systems. In Proceedings of the IFIP Congress, Stockholm, Sweden, 5–10 August 1974; pp. 187–191.

7. Wang, Y.; Liao, H.; Reveliotis, S.; Kelly, T.; Mahlke, S.; Lafontaine, S. Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software. In Proceedings of the 48th IEEE Conference on Decision and Control (CDC) Held Jointly with 2009 28th Chinese Control Conference, Shanghai, China, 15–18 December 2009; pp. 4971–4976.

8. Lin, Y.; Kulkarni, S.S. Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, San Jose, CA, USA, 21–26 July 2014; pp. 237–247.

9. Smith, E.K.; Barr, E.T.; Le Goues, C.; Brun, Y. Is the cure worse than the disease? overfitting in automated program repair. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 30 August–4 September 2015; pp. 532–543.

10. Chandy, K.M.; Misra, J.; Haas, L.M. Distributed deadlock detection. *ACM Trans. Comput. Syst.* 1983, 1, 144–156. [CrossRef]

11. Ho, A.; Smith, S.; Hand, S. *On Deadlock, Livelock, and Forward Progress*; Technical Report; University of Cambridge, Computer Laboratory: Cambridge, UK, 2005.

12. Singhal, M. Deadlock detection in distributed systems. *Computer* 1989, 22, 37–48. [CrossRef]

13. Guo, J.; Seely, S. Monitor preprocessor for Pthreads. In Proceedings of the 34th Annual Frontiers in Education, Savannah, GA, USA, 22–23 October 2004.
14. Giebas, D.; Wojszczyk, R. Multithreaded Application Model. In Proceedings of the 16th International Conference: Distributed Computing and Artificial Intelligence, Avila, Spain, 26–28 June 2019; Springer: Berlin/Heidelberg, Germany, 2019; Volume 1004, pp. 93–103.

15. Shih, C.S.; Stankovic, J.A. Survey of Deadlock Detection in Distributed Concurrent Programming Environments and Its Application to Real-Time Systems and Ada; Technical Report UM-CS-1990-069; University of Massachusetts: Fall River, MA, USA, 1990.

16. Jin, G.; Song, L.; Zhang, W.; Lu, S.; Liblit, B. Automated atomicity-violation fixing. In ACM Sigplan Notices; ACM: New York, NY, USA, 2011; Volume 46, pp. 389–400.

17. Liu, P.; Zhang, C. Axis: Automatically fixing atomicity violations through solving control constraints. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 299–309.

18. Park, S.; Lu, S.; Zhou, Y. CTrigger: exposing atomicity violation bugs from their hiding places. In ACM SIGARCH Computer Architecture News; ACM: New York, NY, USA, 2009; Volume 37, pp. 25–36.

19. Wang, Y.; Kelly, T.; Kudlur, M.; Lafontaine, S.; Mahlke, S.A. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, San Diego, CA, USA, 8–10 December 2008; Volume 8, pp. 281–294.

20. Lafontaine, S.; Wang, Y.; Reveliotis, S. Eliminating concurrency bugs in multithreaded software: An approach based on control of petri nets. In Proceedings of the International Conference on Applications and Theory of Petri Nets and Concurrency, Milan, Italy, 24–28 June 2013; pp. 21–28.

21. Sen, K. Race directed random testing of concurrent programs. ACM Sigplan Not. 2008, 43, 11–21. [CrossRef]

22. Cai, Y.; Wu, S.; Chan, W. ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 491–502.

23. Kavi, K.; Moshtaghi, A.; Chen, D.J. Modeling multithreaded applications using Petri nets. Int. J. Parallel Program. 2002, 30, 353–371. [CrossRef]

24. Ullman, J.D.; Aho, A.V. Principles of Compiler Design; Addison Wesley: Reading, MA, USA, 1977.

25. Durve, N. Automatically Improving the Design of Code. Master’s Thesis, Department of Computing Imperial College, London, UK, 2007.

26. Kroening, D.; Poetzl, D.; Schrammel, P.; Wachter, B. Sound static deadlock analysis for C/Pthreads. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 379–390.

27. Li, J.; Liu, X.; Jiang, L.; Liu, B.; Yang, Z.; Hu, X. An Intelligent Deadlock Locating Scheme for Multithreaded Programs. In Proceedings of the 2019 3rd International Conference on Intelligent Systems, Metaheuristics & Swarm Intelligence, Male, Maldives, 23–24 March 2019; pp. 14–18.

28. Tongping, L.; Zhou, J.; Silvestro, S.; Liu, H. Defeating Deadlocks in Production Software. U.S. Patent App. 16/159,234, 18 April 2019.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).