DOI: 10.15514/ISPRAS-2022-34(4)-5

Обнаружение ошибок взаимоисключающей блокировки в программах на языке C# при помощи методов статического анализа

П.И. Рагозина, ORCID: 0000-0003-4219-7203 <pragozina@ispras.ru>
В.Н. Игнатьев, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
Московский государственный университет им. М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. В статье рассматриваются алгоритмы статического анализа, которые направлены на поиск трех типов ошибок, связанных с понятием синхронизирующего монитора: переопределение переменной взаимоисключающей блокировки внутри критической секции; использование переменной некорректного типа при входе в монитор; блокировка с задействованием объекта, имеющего методы, которые используют для блокировки ссылку на экземпляр (this). Разработанные алгоритмы опираются на технологию символьного исполнения и опираются на межпроцедурный анализ с применением резюме функций, что обеспечивает масштабируемость, чувствительность к полям, контексту, потоку управления. Полученные методы реализованы в инфраструктуре статического анализатора SharpChecker, использующего элементы компиляторной платформы Roslyn, в виде трёх детекторов. С их помощью при тестировании на проектах с открытым кодом найдено 23 ошибки и получена доля верных срабатываний в 88.5%, в то время как потребление времени каждым детектором составляет от 0.1 до 0.7% от общего времени работы анализатора. Ошибки, для поиска которых были разработаны данные детекторы, сложно обнаружить другими способами, помимо статического анализа, из-за того, что они тесно связаны с понятием многопоточности. При этом находить их необходимо: всего один подобный дефект может привести к нестабильности работы программы и даже сделать её уязвимой для злоумышленников.

Ключевые слова: статический анализ; поиск дефектов; символьное исполнение; язык C#; ошибки синхронизации; критическая секция; межпроцедурный анализ.

Для цитирования: Рагозина П.И., Игнатьев В.Н. Обнаружение ошибок взаимоисключающей блокировки в программах на языке C# при помощи методов статического анализа. Труды ИСП РАН, том 34, вып. 4, 2022 г., стр. 63-78. 10.15514/ISPRAS-2022-34(4)-5

Detection of erroneous usage of synchronization monitor in C# via static analysis

P. Ragozina, ORCID: 0000-0003-4219-7203 <pragozina@ispras.ru>
V. Ignatyev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>
Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia
Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russian Federation

Abstract. The paper describes static analysis algorithms aimed at finding three types of errors related to the concept of a synchronizing monitor: redefinition of a variable of mutually exclusive locking inside a critical section; use of an incorrect variable type when entering the monitor; blocking involving an object that has methods that use a reference to an instance (this) to lock. Developed algorithms rely on symbolic execution technology and involve interprocedural analysis via summary of functions, which ensures scalability, field-, context-, and flow-sensivity. Proposed methods were implemented in the infrastructure of a static analyzer in the form of three separate detectors. Testing on the set of open source projects revealed 23 errors and the true positive ratio of 88.5% was obtained, while the time consumption only made up from 0.1 to 0.7% of the total analysis time. The errors that these detectors were designed to find are difficult to detect by testing or dynamic analysis because of their multithreading nature. At the same time, it is necessary to find them: just one such defect can lead to incorrectness of the program and even make it vulnerable to intruders.

Keywords: static analysis; software error detection; symbolic execution; C# language; synchronization errors; critical section; inter-procedural analysis.

For citation: Ragozina P., Ignatyev V. Detection of erroneous usage of synchronization monitor in C# via static analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 4, 2022. pp. 63-78 (in Russian). DOI: 10.15514/ISPRAS-2022-34(4)-5

1. Введение

В многопоточном программировании взаимоисключающая блокировка – это механизм, обеспечивающий синхронизацию данных. Синхронизация играет важную роль в обеспечении безопасности кода, так как её отсутствие или неправильное функционирование могут привести к *состоянию гонки* (race condition) – серьёзная ошибка, при которой работа системы или приложения зависит от того, в каком порядке выполняются разными потоками части кода.

С помощью традиционных методов диагностики ошибки блокировки трудно обнаружить и исправить ввиду осложнений воспроизводимости, вызываемых многопоточностью. Так, например, состояние гонки сделает ненадёжным применение отладки с помощью тестов, особенно в ситуациях, когда проблема возникает лишь для малой доли потоков при строго определенных условиях – тогда вероятность воспроизведения ошибки также значительно уменьшается, и возможна ситуация, когда программист просто не сможет воспроизвести возникшую у пользователя проблему. В качестве решения возможно создание значительно более громоздких тестов для повышения вероятности обнаружения значений, но даже тогда дефект может быть пропущен и не исправлен. Намного эффективнее оказываются методы статического анализа, так как они не требуют реального выполнения проверяемой программы.

Частым способом защитить данные при многопоточном программировании является создание *критической секции* – фрагмента кода, который в один момент времени может выполняться только одним потоком. В работе рассмотрен способ поиска ошибок в характерном для языка C# способе реализации критической секции – при помощи некоторой переменной синхронизации, выполняющей роль мьютекса для процессов. Изменение значения такой переменной внутри критической секции недопустимо, так как может

привести к её использованию несколькими потоками одновременно, а следовательно, и к состоянию гонки.

```
1    class A
2    {
3       static object obj = new object();
4       static int a = 0;
5       public static void Proc()
6       {
7          lock (obj)
8          {
9             obj = new object(); // !!!
10            a++;
11         }
12      }
13   }
```

Рис. 1. Простейший пример ошибки переопределения переменной блокировки
Fig. 1. Simple example of lock reassignment error

Рассмотрим пример на рис. 1. Если один из потоков, выполняющих функцию Proc, поменяет значение obj внутри критической секции (строка 9), то другой поток сможет попасть внутрь занятой первым процессом секции, так как новый объект-мьютекс obj, в отличие от старого, свободен. В свою очередь, это может привести к одновременному выполнению потоками строки 10 и некорректной работе программы.

```
 1   public class A
 2   {
 3         public void fa()
 4      {
 5         lock (typeof(int)) // !!!
 6         {
 7            /* Некоторый код */
 8            lock (typeof(float)) // !!!
 9            {
10            }
11         }
12      }
13      }
14   public class B
15   {
16         public void fb()
17      {
18         lock (typeof(float)) // !!!
19         {
20            /* Некоторый код */
21            lock (typeof(int)) // !!!
22            {
23            }
24         }
25      }
26      }
```

Рис. 2. Пример ошибочного типа переменной блокировки (System.Type)
Fig. 2. Simple example of improper lock object type (System.Type)

Также важно, чтобы используемая переменная сама не была общим ресурсом (к примеру, типом или строковой константой) – это может стать источником уязвимостей и взаимоблокировок. Рассмотрим пример на рис. 2. Здесь, если функция fb будет запущена одновременно с fa, то это приведет к взаимной блокировке (deadlock), так как ни одна из них не сможет продолжить работу без занятого другой ресурса.

Частный случай общего ресурса как объекта блокировки – ссылка на текущий экземпляр класса (this), так как экземпляр может быть легко получен методом извне. Такое решение становится особенно опасным, когда объект типа, имеющий функции с такой блокировкой, сам используется как аргумент lock в другом участке программы. Рассмотрим случай, изображенный на рис. 3. Здесь любой сторонний класс A может приостановить работу функции fb(), потенциально блокируя выполнение всех методов класса B, использующих lock(this).

```
 1      public class B
 2      {
 3      public void fb()
 4         {
 5            lock (this) // !!!
 6            {
 7               /* Некоторый код */
 8            }
 9         }
10   }
11   public class A
12   {
13      public void fa(B b)
14      {
15         lock (b) // !!!
16         {
17            Thread.Sleep (100000);
18         }
19      }
20   }
21      public class C
22      {
23      public void fc()
24      {
25           A a = new A();
26         B b = new B();
27         Task.Factory.Run(() => a.fa(b));
28         Task.Factory.Run(b.fb);
29      }
30   }
```

Рис. 3. Пример недопустимого задействования в мониторе объекта с методом, применяющим блокировку на this
Fig. 3. Example of invalid lock object with a method using 'this' as lock

Данная работа посвящена задаче обнаружения в программах, написанных на языке C#, подобных ошибок.

2. Механизм синхронизации

Основной способ реализации критической секции в программах на языке C# – при помощи класса System.Threading.Monitor, использующего в качестве мьютекса для её защиты некоторый объект. Основные методы – void Monitor.Enter(object obj), обеспечивающий вход

в критическую секцию, защищенную с помощью объекта obj, и void Exit(object obj), обеспечивающий освобождение obj и выход из секции [1]. Также можно вызывать Monitor.Enter с дополнительным булевым параметром ref _lockWasTaken. Тогда значение true этого параметра означает, что блокировка была успешно выполнена. Однако более распространённым инструментом является оператор lock(object obj), созданный для упрощения работы с классом Monitor. Он представляет собой сочетание Monitor.Enter(obj, ref _lockWasTaken), Monitor.Exit(obj) и блока try … catch … finally. В finally при истинности _lockWasTaken вызывается Monitor.Exit. Таким образом, блокировка освобождается, даже если возникает исключение в теле оператора. При анализе кода на этапе построения графа потока управления эти два способа создания критической секции будут аналогичны (табл. 1). При этом важно отметить, что при входе в монитор с использованием объекта obj роль мьютекса исполняет не переменная, а сам объект, на который она ссылается.

Табл. 1. Эквивалентное представление оператора lock
Table 1. Equivalent representation of lock operator

Функция, использующая оператор lock	**Аналогичная функция, использующая метод Monitor**

`public void f()`
`{`
`  object obj = new object();`
`  lock (obj)`
`  {`
`  /* Некоторый код */`
`  }`
`}`
	`public void f()`
	`{`
	`  object obj = new object();`
	`  bool _lockWasTaken = false;`
	`  try`
	`  {`
	`  Monitor.Enter(obj, ref _lockWasTaken);`
	`  /* Некоторый код */`
	`  }`
	`  finally`
	`  {`
	`  if (_lockWasTaken)`
	`  Monitor.Exit(obj);`
	`  }`
	`}`

Ещё один метод синхронизации – применение разделяемых ресурсов с использованием классов System.Threading.ReaderWriterLock [2] и System.Threading.ReaderWriterLockSlim [3]. Оба они позволяют получить доступ к ресурсу одному записывающему процессу (writer/"писатель") и нескольким считывающим (readers/"читатели"). При этом "читатели" могут получать доступ к ресурсу только в том случае, если в текущий момент "писатель" его не блокирует. Их принципы работы очень сходны между собой: создается объект класса, затем выполняется работа с ресурсами при помощи методов этого класса. Разница состоит в том, что ReaderWriterLockSlim проще в использовании и позволяет избежать ряда случаев, приводящих к тупикам. В связи с этим ReaderWriterLock применяется значительно реже и в данной статье рассматриваться не будет.

Основные методы класса ReaderWriterLockSlim:

- EnterReadLock(), TryEnterReadLock(), ExitReadLock() – для блокировки на чтение;
- EnterWriteLock(), TryEnterWriteLock(), ExitWriteLock() – для блокировки на запись;
- EnterUpgradableReadLock(), TryEnterUpgradableReadLock(), ExitUpgradableReadLock() – для обновляемой блокировки на чтение. Обновляемая блокировка аналогична блокировке на чтение за исключением того, что позднее она может быть расширена до блокировки на запись.

Так же, как и в случае с обычной критической секцией, используемый в блокировке объект (в данном случае типа ReaderWriterLockSlim) нельзя переопределять.

Кроме того, по уже рассмотренным ранее причинам (см. рис. 3), нежелательными аргументами являются объекты, являющиеся общими ресурсами, неограниченно доступными из других участков программы. К таким объектам относятся [4]:

- объекты Type;
- строковые константы и интернированные строки (interned string);
- ссылки на текущий экземпляр класса (this).

Их опасность в том, что любой метод – в ряде случаев даже принадлежащий сторонней программе – может бесконтрольно получить доступ к общедоступному объекту блокировки, заблокировать с его помощью (возможно, умышленно) необходимый ресурс и спровоцировать взаимную блокировку (deadlock).

3. Особенности реализации

3.1 Статический анализатор

Как уже указано выше, работа над детекторами выполнена в рамках разработки *SharpChecker* – инструмента статического анализа для программ на языке C#, основанного на компиляторной платформе Microsoft .NET Compiler Platform (Roslyn) [5]. Roslyn — компиляторная инфраструктура с открытым исходным кодом для языков C# и Visual Basic, предоставляющая интерфейсы для компиляции, анализа, рефакторинга кода. В нашем случае, Roslyn предоставляет АСД, таблицу символов и основу для ГПУ, а SharpChecker использует их для построения графов потока управления всех методов и дальнейшего анализа. Этот процесс происходит следующим образом [6][7].

A. Проводится разбор предоставленного синтаксического дерева. Производится анализ всех возможных явных и неявных вызовов. Строится иерархия наследования для классов. Выполняется поиск синтаксических ошибок, которые возможно обнаружить с помощью разбора АСД.
B. Строится статический граф вызовов.
C. Производится обход графа от вызываемой функции к вызывающей. Если в нём есть циклы, то они разрываются в произвольном месте. Для каждой функции выполняется анализ на основе символьного выполнения при помощи разработанного в SharpChecker движка статического символьного выполнения с объединением состояний. Выполняют свою работу детекторы, чувствительные к путям. При анализе часть собранной информации сохраняется в резюме метода для использования в методах, которые вызывают его. Анализ происходит параллельно.
   a. Производится построение ГПУ для каждой функции.
   b. Производится обход базовых блоков. Движок анализа генерирует используемые впоследствии события различных типов, например, присваивание значения в переменную или разыменование переменной.
   c. Детекторы, подписываясь на указанные события, могут накапливать необходимые данные и проводить проверки.
D. Вызывается обработчик завершения анализа, выдающий предупреждения на основе собранной информации.

3.2 Символьное выполнение

Рассмотрим подробнее основной метод статического анализа, обеспечивающий внутрипроцедурный анализ и активно применяемый в разработанных детекторах. Каждая функция при символьном выполнении полагается точкой входа в программу. При её непосредственном запуске параметры и состояние кучи могут быть произвольными, поэтому начальное состояние параметризуется набором символьных переменных. Множество способов запуска функции можно получить путем их замены на всевозможные значения.

Основная цель метода символьного выполнения – построение для любого выбранного пути ГПУ символьного состояния, то есть множества выражений, которое при замене символьных переменных на конкретные значения будут соответствовать верным результатам выполнения операторов функции.

Такой метод позволяет "смоделировать" работу функции для всевозможных вариантов ветвления, не запуская её, однако у него есть свои ограничения. В частности, требуется строго определить набор возможных путей. Для этого вводится понятие графа развёртки – ациклического графа, полученного из ГПУ путём допущения некоторого максимального числа k итераций циклов и представление всех возможных при таком ограничении участков кода, достигаемых через обратные рёбра, в виде отдельных подграфов, которые достигаются с помощью только прямых ребер. Такой метод представляет собой аппроксимацию и снижает находимое число ошибок, однако потенциально почти бесконечное число путей сделало бы символьное выполнение малоэффективным.

3.3 Детекторы

3.3.1 LockObjectReassignment

Для обнаружения ошибок, вызванных переопределением мьютекса блокировки, использование анализа синтаксического дерева возможно в некоторых частных случаях, однако в большинстве ситуаций синтаксического анализа недостаточно, так как он не является межпроцедурным и не может гарантировать обязательную для ошибки подобного типа чувствительность к контексту, потокам и полям, а также желательную чувствительность к путям (см. рис. 4 и 5).

```
  1     var rwls = new ReaderWriterLockSlim();
  2     rwls.EnterReadLock();
  3     try
  4     {
  5        rwls = new ReaderWriterLockSlim();
// !!!
  6        /*Некоторый код*/
  7     }
  8  finally
  9  {
 10     rwls.ExitReadLock();
 11  }
```

Рис. 4. Пример кода с ошибкой переопределения объекта ReaderWriterLockSlim
Fig. 4. Example of erroneous ReaderWriterLockSlim reassignmemt


```
 1    class A
 2    {
 3         public object x;
 4    }
 5    class B
 6    {
 7       void f(A firstA, A secondA)
 8       {
 9          lock (secondA.x)
10          {
11             firstA = secondA;
12             firstA.x = new object();
// !!!
13          }
14       }
15    }
```

Рис. 5. Пример кода, в котором нечувствительный к полям анализатор не обнаружит ошибку (после приравнивания в строке 11, firstA ссылается на тот же объект, что и secondA, и поле firstA.x становится эквивалентно полю secondA.x)
Fig. 5. Example code in which the field-insensitive analyzer will not detect an error (after assignment in line 11, firstA refers to the same object assecondA, and field firstA.x becomes equivalent to field secondA.x)

Таким образом, метод разбора AST способен обнаружить явное переопределение переменной, но не неявное. Более подходящим оказался метод символьного выполнения.

Поиск ошибки реализован следующим образом: в контексте каждого базового блока создаётся некоторый словарь, по которому можно получить информацию обо всех переменных, которые хотя бы на одном из путей ГПУ были задействованы в незавершённой синхронизации (lock, ReaderWriterLockSlim) и их возможных значениях с учётом прошлых переопределений.

```
1    static object obj;
2    static int num;
3    lock (obj)
4    {
5          if (num<10)
6          if (num>11)
7             obj = new object;
8    }
```

Рис. 6. Здесь нечувствительным к путям анализатором будет выдано ложное сообщение об ошибке в строке 7 (строка 7 недостижима)
Fig. 6. In this code, a path-insensitive analyzer would issue a false error message in line 7 (line 7 is unreachable)

Если несколько ветвей передают управление в один блок, то словарь в этом блоке будет объединением словарей во всех возможных блоках-предшественниках.

Среди событий, регистрируемых при анализе потока данных, фиксируются следующие.

- События вызова функции (чтобы при вызове функций, связанных с критическими секциями, корректировать словарь, а для остальных – сравнивать значения переменных из словаря до и после выполнения, обнаруживая переопределение переменной внутри данной функции).
- События присваивания – чтобы выявлять выражения, для которых с левой стороны

находится задействованный в синхронизации элемент.

- События объединения контекстов – для работы со словарями переменных из контекстов блоков.

С учётом этих данных выводятся сообщения об ошибках и сохраняются данные о новых значениях объектов блокировки, чтобы избежать лишних ложных срабатываний.

3.3.2 WrongTypeLock

При решении задачи поиска переменных-мьютексов неправильного типа (за исключением ссылок на экземпляр this, о которых будет рассказано ниже) также избран алгоритм, задействующий метод символьного выполнения, но несколько более простой, нежели в предыдущем случае. Поиск аргументов-типов и аргументов-строковых литералов осуществляется путём простого анализа синтаксического дерева. Однако этого метода оказалось недостаточно для поиска аргументов – интернированых строк (interned string), так как строка, задействованная в блокировке, могла, например, быть заранее интернирована внутри некоторого метода. Итоговый алгоритм для каждой функции фиксирует в резюме строки, которые были интернированы внутри неё (в резюме включаются в том числе и интернированные строки из вложенных функций). Единственное фиксируемое детектором событие – вызов функции. При обнаружении такого события рассматриваются три варианта.

- Вызван метод входа в критическую секцию – тогда его аргумент проверяется;
- Вызвана функция String.Intern – тогда интернируемая строка заносится в резюме вызвавшей функции;
- Вызвана другая функция – тогда из её резюме в резюме вызвавшей функции копируются данные об интернированных строках.

3.3.3 ThisLockObject

Эта задача, фактически являющаяся подзадачей поиска аргументов неправильного типа, вынесена в отдельный детектор, так как в процессе работы решено было использовать для неё другой алгоритм, нежели для других возможных неправильных типов.

Кроме того, итоговая реализация выдаёт предупреждения не на все вызовы формата lock(this), так как на этапе тестирования на проектах с открытым кодом выяснилось, что программисты часто пренебрегают правилами, предостерегающими от этой практики. По этой причине число срабатываний на крупных проектах исчислялось сотнями. Вместо этого избрана другая тактика: в текущей версии детектора выдаются предупреждения только о самой опасной ситуации, возможной при данной ошибке – о случаях, когда внешняя функция использует для блокировки объект класса, предки/потомки которого (или он сам) используют this в блокировках внутри своих функций (рис. 9). В том числе к таким функциям относятся методы с атрибутом [MethodImpl(MethodImplOptions.Synchronized)], добавление которого эквивалентно добавлению оператора lock(this), заключающего в себя всё тело метода. Такая возможность также учитывается.

Важную роль в принципе работы детектора ThisLockObject играет вспомогательный AST-анализатор, во время анализа синтаксического дерева сохраняющий в своих записях информацию о создаваемых классах с «опасными» методами и их предках. Основной детектор, как и предыдущие, является символьным, однако, как и WrongTypeLock, использует только события вызова функций. Для каждой функции входа в критическую секцию проверяется, не принадлежит ли её первый аргумент списку небезопасных для блокировки классов, составленному вспомогательным анализатором типов. Если это так, то выводится соответствующее сообщение.

4. Результаты тестирования

В процессе разработки работа детекторов проверена как на наборе из более чем 100 модульных тестов, так и на open source проектах. Разработанные синтетические тесты включают проверки:

- на чувствительность к контексту:
  - для LockObjectReassignment учитывается возможность изменения переменной с помощью функции – рассмотрены варианты передачи параметра по ссылке и по значению, изменение внутри функции статического члена; учитывается использование оператора lock или класса Monitor);
  - для WrongTypeLock – производится проверка на интернирование строки внутри функции;
- на чувствительность к полям (учитывается структура используемых данных);
- на чувствительность к потоку управления (учитываются такие особенности потока управления, как циклы и ветвления);
- на чувствительность к путям (ошибки в недосягаемых участках кода не фиксируются).

Табл. 2. Статистика работы детекторов
Table 2. Statistics of detectors' performance

	Время без новых детекторов	**Время со включенным детектором**	**Срабатываний**		
			TP	**FP**	**Пропущенных**
LockObjectReassignment	01:20:15	01:20:49 (+0.7%)	7	3	1
WrongTypeLock		01:20:19 (+<0.1%)	9	0	Нет данных
ThisLockObject		01:20:41 (+0.5%)	7	0	Нет данных

Рассмотрим подробнее тестирование на ПО с открытым исходным кодом. Оно производилось на наборе из 21 проекта общим объёмом в 6 млн. строк кода. Анализ выполнялся на машине с 8-ядерным процессором Intel(R) Core(TM) i7-6700, имеющей 32 Gb RAM.Ниже приведены несколько примеров найденных ошибок (рис. 7-9) и статистика работы детекторов (табл. 2).

Данные о необнаруженной ошибке для LockObjectReassignment получены с помощью стороннего анализатора. Эта ошибка связана с более углублённым использованием возможностей ReaderWriterLockSlim, которые текущая версия детектора не учитывает.

4.1 LockObjectReassignment

Ниже приведён пример реальной ошибки, найденной разработанным детектором в коде OpenSim – открытой платформы для создания пользовательских виртуальных симуляций [8]. Версия: 0.9.0.0.

```
Файл /OpenSim/Region/Framework/Scenes/Animation/MovementAnimationOverrides.cs:
92    public void CopyAOPairsFrom(Dictionary<string, UUID> src)
93    {
94       lock (m_overrides)
95       {
96          m_overrides.Clear();
97          m_overrides = new Dictionary<string, UUID>(src);
// !!! LOCK_OBJECT_REASSIGNMENT Lock object
MovementAnimationOverrides.m_overrides was changed in assignment
98       }
99    }
```

Рис. 7. Пример истинного срабатывания LOCK_OBJECT_REASSIGNMENT
Fig. 7. Example of true positive LOCK_OBJECT_REASSIGNMENT warning

4.2 WrongTypeLock

Эта ошибка найдена в эмуляторе с открытым кодом Wcell [9]. Версия: 9.0.

```
Файл /Core/Cell.Core/ObjectPoolMgr.cs:
59    public static bool RegisterType<T>
                          (Func<T> func) where T : class
60    {
61       long typePointer = GetTypePointer<T>();
62       lock (typeof(ObjectPoolMgr))
// !!! WRONG_TYPE_LOCK System.Type is used for locking
63       {
64          if (!Pools.ContainsKey(typePointer))
65          {
66             Pools.Add(typePointer, new ObjectPool<T>(func));
67             return true;
68          }
69       }
70       return false;
71    }
```

Рис. 8. Пример истинного срабатывания WRONG_TYPE_LOCK
Fig. 8. Example of true positive WRONG_TYPE_LOCK warning

4.3 ThisLockObject

Эта ошибка найдена в коде библиотеки Apache Lucene.NET [10]. Версия: 3.0.3.

По данным табл. 2 видно, что влияние разработанных детекторов на время выполнения анализа незначительно (<1%): с точки зрения временных затрат все они достаточно эффективны. Статистика результатов также приемлема. По таблице также можно увидеть, что детектор LockObjectReassignment выдаёт ложные срабатывания, однако их процент находится в допустимых пределах (30%), как и процент ненайденных ошибок.


```
Файл /src/Lucene.Net.Core/Index/BufferedUpdatesStream.cs:
49  public class BufferedUpdatesStream
50  {
       /.../
73        public virtual long Push(FrozenBufferedUpdates packet)
74        {
75        lock (this)
76        {
             /.../
98        }
99        }
       /.../
718  }

Файл /src/Lucene.Net.Core/Index/IndexWriter.cs:
275 internal readonly BufferedUpdatesStream
BufferedUpdatesStream;
1623  public virtual bool TryDeleteDocument(IndexReader
readerIn,
                                              int docID)
1624  {
1661     lock (BufferedUpdatesStream)
// !!! THIS_LOCK_OBJECT Lucene.Net.Index.BufferedUpdatesStream contains
locks on instance (this) in its functions. It is unsafe to use variable of this type for
locking.
1662     {
/.../
1687     }
1700 }
```

Рис. 9. Пример истинного срабатывания THIS_LOCK_OBJECT
Fig. 9. Example of true positive THIS_LOCK_OBJECT warning

5. Сравнение с существующими решениями

В самом инструменте SharpChecker есть следующие детекторы ошибок, также работающие с проблемами синхронизации потоков [11]:

- выявление переменных, к которым потенциально возможен несинхронизированный доступ нескольких потоков и на которые рекомендована установка синхронизирующей блокировки (NO_LOCK.STAT);
- обнаружение участков, где возможны взаимные блокировки – бесконечное ожидание нескольких процессов, каждый из которых использует ресурсы, необходимые другому (DEADLOCK).

Можно заметить, что ни один из них не может предложить полноценного решения нашей проблемы.

В других статических анализаторах существуют следующие детекторы ошибок в области синхронизации (все представленные анализаторы применяют метод символьного выполнения).

- Svace [12] – промышленный статический анализатор для Java и C/C++/C#:
  - Детектор возможных взаимных блокировок, сходный с упомянутым выше.
  - Выявление случаев, когда блокировка семафора выполняется одним потоком два раза подряд без освобождения (DOUBLE_LOCK).
  - Детектор ситуаций, когда функция, освобождая семафор в одних ветвях, сохраняет его до своего завершения в других (NO_UNLOCK).

Обнаружение ситуаций, когда переменная из стека используется для синхронизации (LOCK_ON_STACK). Это является ошибкой, так как каждый поток использует собственный стек.

- Обнаружение вызовов блокирующих функций внутри критической секции, что может привести к долгому ожиданию не только для данного процесса, но и для остальных (LONG_TIME_IN_LOCK).
- Детектор синхронизации доступа к статическому полю при помощи нестатической переменной (WRONG_LOCK.STATIC). Такой доступ может создать ситуацию гонки, т.к. даже для разных объектов синхронизации обращение к статическому полю означает доступ к одному и тому же участку памяти из разных потоков выполнения.

- Infer Static Analyzer [13] – инструмент статического анализа с открытым исходным кодом, разработанный командой инженеров Facebook [14] для языков Java и C/C++/Objective-C (включает в себя инструмент RacerD [15], нацеленный на поиск потенциальных состояний гонки):
  - Поиск классов, в которых есть публичный метод, записывающий данные в некоторый член, используя блокировку и публичный метод, считывающий эти данные без использования блокировки (LOCK_CONSISTENCY_VIOLATION).
  - Поиск объектов, к которым возможен несинхронизированный доступ двух и более потоков, по крайней мере один из которых выполняет запись (THREAD_SAFETY_VIOLATION).
  - Поиск взаимных блокировок.
- Clang Static Analyzer [16] – статический анализатор, который находит ошибки в программах на языках C, C++ и Objective-C:
  - BlockInCriticalSection – обнаруживает использование внутри критической секции блокирующих функций (аналогично LONG_TIME_IN_LOCK в Svace).
  - PthreadLock– обнаруживает случаи неправильной последовательности блокировок/освобождений ресурса.
- SonarQube [17] – платформа с открытым исходным кодом для непрерывного анализа и измерения качества программного кода для большого набора языков, включающего в себя в том числе и C#.
  - Детектор использования переменных блокировки, являющихся общими ресурсами (частично соответствует сочетанию разработанных нами WrongTypeLock и ThisLockObject с некоторыми различиями – например, сигнализирует о любых случаях использования lock(this)).
- Coverity [18] – пакет программного обеспечения, состоящий из статического и динамического анализаторов кода, нацеленный на поиск ошибок и недочётов в безопасности исходных кодах программ, написанных на Си, C++, Java, C# и JavaScript.
  - OVERWRITE_OF_LOCK_FIELD_DURING_CRITICAL_SECTION – детектор переопределений объекта блокировки в критической секции – практически аналогичен LockObjectReassignment.
  - BAD_LOCK_OBJECT – обнаружение объектов блокировки неправильного типа, в частности, интернированных строк, по критериям поиска схож с WrongTypeLock.

  - Поиск взаимных блокировок.

Общее сравнение анализаторов см. в табл. 3.

Табл. 3. Возможности разработанных детекторов по сравнению с существующими анализаторами (анализатор Svace также обнаруживает использование в lock неподходящих объектов, но имеет другие критерии – нацелен на поиск объектов стека и статических полей)
Table 3. Capabilities of the developed detectors compared to existing analyzers (the Svace analyzer detects the use of incorrect objects in lock as well, but has other criteria – it is aimed at searching for stack objects and static fields)

	Переопределение переменной	**Неправильный тип (this)**	**Неправильный тип (другой)**
Svace	-	-	+-
Infer Static Analyzer	-	-	-
Clang Static Analyzer	-	-	-
SonarQube	-	+	+
Coverity	+	-	+
SharpChecker	+	+	+

Таким образом, несмотря на возможность поиска ошибок, связанных с критическими секциями, ошибки описанных в статье видов не находятся большинством рассмотренных сторонних анализаторов. Тем не менее, примеры подобного переопределения несколько раз были обнаружены при исправлении других дефектов в проектах с открытым исходным кодом. Планируется, что разрабатываемый метод поможет эффективнее находить их в будущем.

6. Заключение

Был разработан, реализован и протестирован алгоритм поиска комплекса ошибок осуществления взаимоисключающей блокировки. По итогам тестирования разработанный детектор показал приемлемые скорость и точность, зачастую обнаруживая ошибки, которые не могут найти другие инструменты анализа. Так, поиск ошибки вида Lock Reassignment среди множества рассмотренных анализаторов осуществляется только в инструменте Coverity, который, в отличие от SharpChecker, является закрытым. Помимо этого, новизна проекта состоит в реализации алгоритма поиска подобных ошибок в рамках возможностей и ограничений, заданных конкретно анализатором SharpChecker, а именно, конкретной реализацией межпроцедурного чувствительного к контексту и путям статического символьного выполнения с объединением состояний.

В дальнейшем планируется ещё больше повысить эффективность работы полученного анализатора, а также продолжить расширять возможности нашего инструмента в области ошибок многопоточности, рассмотрев такие ошибки, как выход из критической секции, в которую не был совершён успешный вход, многократный захват и освобождение одной и той же переменной блокировки и т.д.

Список литературы / References

[1] Microsoft Docs: .NET API browser: System.Threading: Monitor Class. Available at: https://docs.microsoft.com/en-us/dotnet/api/system.threading.monitor, accessed: 29.07.22.
[2] Microsoft Docs: .NET API browser: System.Threading: ReaderWriterLock Class. Available at: https://docs.microsoft.com/en-us/dotnet/api/system.threading.readerwriterlock, accessed: 01.08.22.
[3] Microsoft Docs: .NET API browser: System.Threading: ReaderWriterLockSlim Class. Available at: https://docs.microsoft.com/en-us/dotnet/api/system.threading.readerwriterlockslim, accessed: 01.08.22.

[4] Microsoft Docs: .NET API browser: C# Keywords: Statement Keywords (C# Reference): lock statement. Available at: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock, accessed: 03.08.22.
[5] The .NET Compiler Platform ("Roslyn"). Available at: https://github.com/dotnet/Roslyn, accessed: 29.07.22
[6] Кошелев В.К. Межпроцедурный статический анализ для поиска ошибок в исходном коде программ на языке С. Диссертация на соискание учёной степени кандидата физико-математических наук, Москва, ИСП РАН, 2017 г., 104 стр. / Koshelev V.K. Interprocedural static analysis for finding errors in the source code of C programs. Thesis for the degree of candidate of physical and mathematical sciences, Moscow, ISP RAS, 2017, 104 p. (in Russian).
[7] Кошелев В.К., Игнатьев В.Н., Борзилов А.И. Инфраструктура статического анализа программ на языке C#. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 21-40 / Koshelev V.K., Ignatyev V.N., Borzilov A.I. C# static analysis framework. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 1, 2016, pp. 21-40 (in Russian). DOI: 10.15514/ISPRAS-2016-28(1)-2.
[8] OpenSimulator. Available at: http://opensimulator.org, accessed: 23.10.2021.
[9] WCell: World of Warcraft emulator written in C#/.NET 4.0, with design and extensibility in mind. Available at: https://github.com/WCell/WCell, accessed at 15.08.2022.
[10] Welcome to the Lucene.NET website! | Apache Lucene.NET 4.8.0. Available at: https://lucenenet.apache.org, accessed: 23.10.2021.
[11] Белеванцев А.А. Многоуровневый статический анализ исходного кода для обеспечения качества программ. Диссертация на соискание учёной степени доктора физико-математических наук, Москва, ИСП РАН, 2017 г., 229 стр. / Belevantsev A.A. Multi-level static analysis of the source code to ensure the quality of programs. Thesis for the degree of Doctor of Physical and Mathematical Sciences, Moscow, ISP RAS, 2017, 229 p. (in Russian).
[12] Иванников В.П., Белеванцев А.А. и др. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 231-250 / Ivannikov V.P., Belevantsev A.A. et al. Static analyzer Svace for finding of defects in program source code. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 231-240 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7.
[13] Calcagno C., Distefano D. Infer: An automatic program verifier for memory safety of C programs. Lecture Notes in Computer Science, vol. 6617, 2011, pp. 459-465
[14] Calcagno C., Distefano D. et al. Moving fast with software verification. Lecture Notes in Computer Science, vol. 9058, 2015, pp. 3-11.
[15] Liu B., Liu P. et al. When threads meet events: efficient and precise static race detection with origins. In Proc. of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021, pp. 725-739.
[16] Kremenek T. Finding software bugs with the clang static analyzer. Available at: https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf, accessed: 23.10.2021.
[17] Campbell G.A., Papapetrou P.P. SonarQube in action. Manning, 2013, 392 p.
[18] Synopsys Software Security | Software Integrity Group, Available at: http://www.coverity.com, accessed: 13.09.2022.

Информация об авторах / Information about authors

Полина Ильинична РАГОЗИНА – студентка бакалавратуры кафедры системного программирования ф-та ВМК МГУ, исследователь в отделе компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, символьное выполнение, поиск дефектов синхронизации в многопоточных программах.

Polina Ilyinichna RAGOZINA is a bachelor's student of the System Programming Department of the CMC Faculty of Moscow State University, a researcher in the Department of Compiler Technologies of ISP RAS. Scientific interests: static analysis of programs, symbolic execution, search for synchronization defects in multithreaded programs.

Валерий Николаевич ИГНАТЬЕВ, кандидат физико-математических наук, старший научный сотрудник ИСП РАН, старший преподаватель кафедры системного программирования

факультета ВМК МГУ. Научные интересы включают методы поиска ошибок в исходном коде ПО на основе статического анализа.

Valery Nikolayevich IGNATYEV, PhD in computer sciences, senior researcher at Ivannikov Institute for system programming RAS and senior lecturer at system programming division of CMC faculty of Lomonosov Moscow State University. He is interested in techniques of errors and vulnerabilities detection in program source code using static analysis.