Concurrency Bug Finder

Roslyn-Based Static Code Analysis for C#

Christoph Amrein

University of Applied Sciences Rapperswil
Supervised by Prof. Dr. Luc Bläser
Fall Term 2016

December 16, 2016

Abstract

This thesis presents a catalog of concurrency-related bug patterns. In contrast to design patterns, bug patterns describe specific code constellations that are error-prone. Each pattern is introduced with a brief description of the possible concurrency issues that may occur and what changes are necessary to avoid them. Moreover, the patterns have a short explanation of how their presence can be observed.

Concurrency Bug Finder is a *Roslyn*-based static code analysis tool for the introduced bug patterns. The underlying core idea of the analyses is: "Code made to run concurrently will run concurrently." This presumption removes the necessity to identify if a code is truly executed concurrently. Instead, the concurrent execution of the code is assumed if synchronization primitives are used.

The experimental evaluation verified the reliability of the different bug pattern analyses by scanning various projects with *Concurrency Bug Finder* and manually assessing the correctness of the reported issues. It revealed that more than 36% of the 365 findings are malign and only 29% are false positives. The remaining 35% indicate code locations that could be improved with regard to the design.

Contents

1. Introduction

The goal of *Concurrency Bug Finder* is to identify potential concurrency-related bugs. Instead of a sophisticated analysis (e.g. determining concurrently running code first and then finding potential issues with this information), it searches for specific code constellations that are error-prone. These constellations are termed as *Bug Patterns*. A popular static analysis tool applying this principle is *Find Bugs* [16] for Java. While it does identify certain concurrency related bugs, it is mainly aiming for general bug patterns.

The applied analysis is only local, per document to be more specific. Therefore, it is often not possible to clearly identify concurrently running code. To overcome this, a basic idea for every pattern is used: "Code made to run concurrently will run concurrently." This idea means, as soon as concurrency control methods are identified, e.g. the use of locks, the code is assumed to run concurrently. Furthermore, this thesis and the implementation try to determine how reliable *Bug Patterns* can be identified without the use of sophisticated data and control flow analysis.

The preceding project thesis was based on the same idea of bug patterns. However, in contrast to this thesis, the pattern detection implementations were more extensive. It applied static code analysis techniques like control and data flow analysis. This thesis focuses on avoiding such techniques by using simple approaches with the help of Roslyn and .NET features. Furthermore, while this thesis concentrates on a smaller set of bug patterns, it presents more substantial descriptions of them and their analysis implementations.

The thesis will first introduce several bug patterns covered by the analysis in Chapter 2. Each pattern comes with a description of possible issues that may arise and a general idea how its presence can be observed. In Chapter 3, several key techniques used by the implementation are introduced. Additionally, a description how *Concurrency Bug Finder* identifies the different bug patterns is given. To verify the reliability of the various analyses, an experimental evaluation is made whereas the results are shown in Chapter 4. Ultimately, Chapter 5 gives a conclusion about the results.

2. Bug Patterns

This chapter proposes different concurrency related bug patterns. Each pattern is introduced with a description of the potential issues related to it. The presence section briefly describes how the pattern can be observed within the code.

The introduced bug patterns have been mindfully collected from different sources. The primary source is the personal experience by reviewing various applications over the years. Also by thinking of what could go wrong when using certain concurrency related functions. Another important source are communities like *stackoverflow* [4] where people can ask for advice for different issues. As Microsoft gives recommendations for concurrency, also some of the patterns are based on them.

The patterns are not valid for all situations, meaning they lead to false positives. A very general exception of these patterns is the possibility that the code does not run concurrently with others. But the patterns are defined on the assumption that the code will run concurrently. General exceptions are applied to reduce the chance of false-positives. For example, it is assumed that constructors will never run concurrently. However, as this is only an assumption, it is not always true. It is still possible to create a new thread or TPL task within the constructor, making some parts of the constructor run concurrently. But as this is rather an uncommon situation and the detection mechanisms would get complex, these cases are not taken into account. By the nature of simplifications, this will lead to undesired false positives or the even more unwanted false negatives. Such false negatives are situations where the code is indeed a bug but not detected by the analyzer. A general source for such a false negative would be the overall exclusion of constructors. Nevertheless, the primary goal is to reduce the number of false positives. The reason is that people tend to ignore reports from static analysis tools if they receive an excessive amount of false positives.

For the ease of identification, unique identifiers are given to the patterns. An identifier is constructed out of the patterns category and its name. Table 2.1 lists all the abbreviations with the pattern's full name. Additionally, the table denotes the severity for each pattern. A severity level is awarded by considering three points for each pattern:

- How local are the effects of the pattern?
- How easily can its effects be observed during the system's execution?
- How are the system's stability and correctness affected?

Sections 2.1 to 2.4 propose the pattern catalog. Each section thereby introduces a particular kind of concurrency related bugs with a brief explanation of

ID	Name	Severity
Data Races		
DR_UA64V	Unsafe Accesses to 64-bit Values	critical
DR_IDCL	Insufficient Double-Checked Locking	critical
DR_ULVV	Unguaranteed Loop Variable Visibility	moderate
Race Conditions		
RC_NAAV	Non-Atomic Assignment on Volatile	critical
RC_UCCA	Unsafe Concurrent Collection Access	critical
RC_PFSE	Parallel For with Side Effects	major
Pitfalls		
PF_INIT	Interruption of not Interruptible Thread	moderate
PF_MCS	Missing Cancellation Strategy	low
PF_SLLO	String Literal as Lock Object	critical
Recommendations		
RA_PSV	Prefer Slim Versions	low
RA_PLEM	Prefer Lock over Explicit Monitor	moderate

Table 2.1: Pattern IDs and Severities

the underlying issues.

2.1 Data Races

Most definitions describe data races as a low-level problem. More specifically, a problem with the memory model. Optimizers are allowed to modify instructions as they wish. They only have to ensure that the behavior for sequential executions stays the same [3]. For example, an optimizer may decide to cache the value of a variable in a register leading to that the thread does not see changes made by other threads. Another possibility is that an optimizer reorders instructions in an unfortunate way. Therefore, while the order of instructions might be correct in the code, this is not necessarily the case during the execution.

2.1.1 Unsafe Accesses to 64-bit Values

The types `long` and `double` cannot be declared `volatile`. Therefore, variables of this type have to be accessed with an atomic method, for example, a member of `System.Threading.Thread.Interlocked`, when read and written concurrently.

Besides the possible existence of data races, visibility problems, and unfortunate reordering, further issues arise when not accessing variables of these types correctly. Both types require 64 bit of memory. Though a 32-bit system can neither write nor read a 64-bit value in a single atomic operation [6, 8]. These unsafe accesses may lead to the result that a read operation observes an only partially written value, i.e. the first 32 bits of the variable hold the new value and the last 32 bits the old value.

```
1  public class Shared {
2    // Could also be of the type double.
3    private long value;
4
5    // Non-atomic access in one place
6    public long Value => value;
7
8    public void Update(long newValue) {
9      // Atomic access in another.
10     Interlocked.Exchange(ref value, newValue);
11   }
12 }
```

Listing 2.1: Unsafe Accesses to a 64-bit Value

Presence

The bug pattern is present in the analyzed syntax tree if 64-bit typed variables are being inconsistently accessed using atomic operations. More specifically, an atomic operation is used at one point of the syntax tree but not at another. This situation is demonstrated in Listing 2.1.

2.1.2 Insufficient Double-Checked Locking

Singleton is one of the patterns that is found in many applications. Proper synchronization is required when accessing the singleton concurrently. One very common technique to accomplish this is double checked locking [17]. But the implementation of double-checked locking is only correct if the memory model is respected [2, 3]. This is usually ensured by declaring the singleton field with the `volatile` modifier. Of course, the read and write accesses could be protected explicitly with `Volatile.Read` and `Volatile.Write` (or similar) instead of declaring the variable `volatile`. Another possibility is the use of `Interlocked` to create a memory barrier between the object construction and the field assignment.

If this is omitted, it is possible that the instance of the singleton object is already visible to other threads without being entirely constructed [3]. This undesired effect could happen if an optimizer decides to re-order the instructions of the object construction and assignment to the shared field.

Presence

The pattern is present if multiple conditions are fulfilled. First, there is a double-checked locking. Secondly, within the double-checked locking there is an assignment combined with an object construction. Lastly, the field the object is assigned to was not declared with the `volatile` modifier. This constellation is sketched in Listing 2.2.

```
1  public class Shared {
2    private static object syncLock = new object();
3
4    // The variable protected by the double-checked locking
5    // is not volatile.
6    private static Shared instance;
7
8    public static Shared Instance {
9      get {
10       if(instance == null) {
11         lock(syncLock) {
12           if(instance == null) {
13             // No memory barrier between object
14             // construction and variable write access.
15             instance = new Shared();
16           }
17         }
18       }
19       return instance;
20     }
21   }
22 }
```

Listing 2.2: Double-Checked Locking with Data Race

2.1.3 Unguaranteed Loop Variable Visibility

A typical case of variable visibility issues is that threads utilize a while loop which runs for as long as a shared variable has a particular value. To stop the thread from doing its continuous work, a different thread sets the shared variable to a value that no longer satisfies the loop's condition. If the shared variable is neither `volatile` nor accessed with a suitable synchronization primitive, there is no guarantee that the thread running the loop will ever see the update of the other thread.[1]

This issue arises from the optimization process of the compiler, the run-time system, or the hardware [5, 15]. Without the use of the `volatile` modifier on the shared variable, the optimizers are allowed to store the variable's value in a register without ever updating the initially retrieved value. This optimization could lead to the situation that another thread may update the value without the looping thread getting aware of the new value.

But not all of these loops necessarily yield the described bug. Loops that are enclosed by a monitor-lock will have up-to-date values when entering the critical section. Besides, loops inside monitor-locks are commonly used in combination with `Monitor.Wait` to await a certain value of a shared variable.

[1] The source of this pattern is the lecture "Parallel Programming" by L. Bläser, HSR

```
1  public class Shared {
2    // The type is irrelevant
3    private bool state;
4
5    public void SomeMethod() {
6      while(state) {
7        // do any kind of work but do not write state
8      }
9    }
10
11   public void SomeOtherMethod() {
12     // A write-access somewhere outside of the loop.
13     state = false;
14   }
15 }
```

Listing 2.3: Example Loop with Condition Variable Data Race

Presence

The presence can be observed by looking at the loops of the syntax tree. If there are any accessed fields inside the loops condition that are only updated outside of the loop's body, chances are given that the pattern is present. Listing 2.3 illustrates this situation a bit more concrete with a variable of the type `bool`. However, the type is irrelevant for this pattern.

2.2 Race Conditions

Race conditions are —unlike data races— not bugs that occur because of not respecting the memory model of the language. They are bugs that indicate themselves as synchronization problems on a semantically higher level. This means, there is already an issue with the problem implementation itself.

2.2.1 Non-Atomic Assignment on Volatile

The field modifier `volatile` is one of the low-level utilities provided by the .NET framework. It ensures that changes to variables of one thread are visible to other threads, prevents re-ordering by applying a memory-fence and ensures atomic write-accesses. Nevertheless, operations like `++` and `+=` will not be thread-safe [3]. Therefore, they should be replaced with their equivalent of the `System.Threading.Interlocked` type. This requirement does not only apply to `volatile` fields. But this pattern is restricting itself to them to reduce the chance of false positives.

It is noteworthy that there have been cases identified that use `volatile` fields but enclose the write access with synchronization primitives like monitor locks. This renders the use of the interlocked operations unnecessary.

```
1 public class Shared {
2   // The type is irrelevant
3   private volatile int value;
4
5   public void SomeMethod() {
6     // Or any other read and write access instruction.
7     ++value;
8   }
9 }
```

Listing 2.4: Volatile Variable Write Access with a Race Condition

Presence

Any non-atomic assignment like `+=` to `volatile` fields in the syntax tree can be considered an instance of this pattern. The skeleton of this pattern is shown in Listing 2.4. The type `int` is purely for demonstration purposes.

2.2.2 Unsafe Concurrent Collection Access

The concurrent collections of the .NET framework provide a rich functionality when accessing collections in parallel. However, they do not guarantee the consistency between two operations. Consider, for example, checking for the existence of a key in a dictionary and then retrieving the corresponding value. The corresponding key could have been removed just in-between these two operations, resulting in a failure of the succeeding value retrieval.

Presence

Checking a concurrent collection for a specific state and immediately retrieving a value based on this test is most likely a consistency error. So, checking if there is any concurrent collection access nested within an `if`-statement querying the collection is a reasonable indicator. The structure illustrated in Listing 2.5 is rather concrete as it is meant to aid the understanding of the pattern. Line 7 sketches the state query and line 9 the succeeding action which depends on the resolved state.

A common source of this pattern is by converting code into a concurrently accessed variant and carelessly replacing basic collections with their concurrent counterpart.

2.2.3 Parallel For with Side Effects

.NET's TPL allows the parallel execution of a loop's body with the help of the methods `For` and `ForEach` of `System.Threading.Tasks.Parallel`. So different from the other patterns, this pattern does not simply assume the concurrent execution of code sections. Due to the ease of use of these parallel constructs, race conditions can be quickly introduced. This means, accesses to shared variables such as closures need to be synchronized.

Of course, there are exceptions which do not require any synchronization. For example when accessing arrays. If each of the created tasks by the loop accesses

```
1  public class Shared {
2    private ConcurrentDictionary<int, int> data =
3      new ConcurrentDictionary<int, int>();
4
5    public int GetValue(int id) {
6      // State query
7      if(data.ContainsKey(id)) {
8        // Action depending on the state
9        return data[id];
10     }
11     return -1;
12   }
13 }
```

Listing 2.5: State-Relevant Action on Concurrent Collection

a different index of said array, there is no further synchronization necessary as the threads do not access the same memory.

Presence

The pattern can be observed by looking at the body of a parallel loop. Any un-synchronized read and write accesses to a variable declared outside of the loop are candidates.

2.3 Pitfalls

This section introduces patterns that do not necessarily lead to a bug but may result in unexpected behaviors.

2.3.1 Interruption of not Interruptible Thread

Threads that are the target of an interruption have to make use of an interruptible method. More specifically, the threads have to either be or move to any of the wait, sleep, or join states after the interruption signal is sent [14]. This is necessary because there is no possibility manually checking if a thread was interrupted in .NET. If the thread does not change into any of the mentioned states, the interruption will have no effect and the desired `ThreadInterruptedException` is never thrown.

Presence

This pattern is present if the executed code of the interrupted thread does not invoke any of the methods that are known to move the thread to the wait, sleep or join state. Methods doing so are the following:

- `Monitor.Enter`, thus `lock` statements as well
- `Monitor.Wait`

```
1  public class Shared {
2    private Thread thread;
3
4    public void SomeThreadBody() {
5      // Do anything but move the thread
6      // into the wait, sleep or join state.
7    }
8
9    public void SomeThreadCreation() {
10     thread = new Thread(SomeThreadBody);
11   }
12
13   public void SomeStopMethod() {
14     thread.Interrupt();
15   }
16 }
```

Listing 2.6: Interruption of not Interruptible Thread

- `Thread.Sleep`
- `Thread.Join`

Listing 2.6 sketches the structure. The interrupted thread executes code that does not invoke any of the interruptible methods.

2.3.2 Missing Cancellation Strategy

Threads receiving a thread interruption have to implement a sufficient cancellation strategy. More specifically, they have to catch `System.Threading.ThreadInterruptedException`. Without doing so, the application may crash because of an uncaught exception.

Presence

The presence of this pattern can be observed if there is a thread interruption but the receiving thread does not catch `ThreadInterruptedException`. The simple handling of this exception type does not necessarily mean that a cancellation logic is present. Nonetheless, its absence is most likely a reliable indicator.

There is the possibility that the cancellation strategy is implemented in a catch clause which catches `System.Exception`. As it is recommended to not use general exception handlers [1, 7], this case is neglected. While this will possibly result in false positives, it makes engineers aware of possible design issues. The major structure of this pattern is the same as the one sketched in Listing 2.6. But instead of not invoking an interruptible method, it does not implement the necessary exception handler.

Fat-Version	Slim-Version
`ManualResetEvent`	`ManualResetEventSlim`
`Mutex`	`SempahoreSlim`
`ReaderWriterLock`	`ReaderWriterLockSlim`
`Semaphore`	`SemaphoreSlim`

Table 2.2: Fat Synchronization Primitives and their Slim Counterparts

2.3.3 String Literal as Lock Object

With the help of the `lock` keyword, the application of monitor locking is straightforward. Nevertheless, there is a pitfall when using strings as lock objects. The issue arises from .NET's string pooling. This string pooling leads to the situation that equivalent string literals that appear in the same program result in the same instance [5]. This can cause that code sections are synchronized with each other which are not meant to be synchronized.

To clarify, this is purely an issue with string literals available at compile-time. Strings being constructed during run-time are not subject to string pooling. Therefore, they will not lead to undesired synchronization if used as lock objects.

Presence

This pattern can be observed if there are any string literals used for monitor locks.

2.4 Recommendations

This section introduces general recommendations for concurrency. They do not present bug patterns as presented in the previous sections but motivate the use of certain patterns. These recommendations are either to reduce the possibility of potential bugs or to improve the quality and performance of the code.

2.4.1 Prefer Slim Versions

The .NET framework introduced new synchronization primitives ending with *-Slim*, e.g. `System.Threading.ReaderWriterLockSlim`. These slim versions are specifically designed for thread synchronization within an application and offer improved performance [13, 11, 9].

The *fat* versions are OS-level synchronization primitives. These can not only be used for local but also for inter-process synchronization. For most applications, inter-process synchronization is not needed and therefore the use of OS-level synchronization is an unnecessary overhead. Table 2.2 sketches a table which maps the fat synchronization primitives with their slim counterpart.

Presence

The use of any OS-level synchronization primitive can be considered unnecessary unless they are named instances as these could be used for inter-process

synchronization.

2.4.2 Prefer Lock over Explicit Monitor

.NET provides the `lock` keyword for straightforward and correct monitor synchronization. Nevertheless, explicit monitor synchronization can still be achieved with the use of `Monitor.Enter` and `Monitor.Exit`. Though implementing the monitor synchronization fully correctly requires more than simply putting these instructions inside a `try...finally` block. Two cases have to be considered when doing this: When not putting `Monitor.Enter`, there is a chance for an exception right after acquiring the lock but before entering the `try...finally` block. This means that the lock will never be released again. However, if the enter statement is enclosed by the `try...finally` block, there is a chance that an exception is thrown right before acquiring the lock. This would lead to that the `Monitor.Exit` attempts to release a lock that was not acquired.

Because of this problematic, beginning with .NET 4 the enter instructions were extended with an additional argument [1, 10]. This argument helps to identify if a lock was acquired or not, supporting the identification of the actual situation.

Presence

The usage of explicit monitor synchronization is present if any of the manual monitor enter instructions —`Monitor.Enter` and `Monitor.TryEnter`— is used. However, the explicit usage could be implemented correctly. There is the chance for this situation if an argument with the `ref` modifier is used. On the other hand, it has to be considered if there is a justification not to use the lock-statement instead. Most likely only for the use of `TryEnter` to implement alternative execution paths if the lock could not be acquired.

3. Implementation

The implementation uses Roslyn [12] to analyze the syntax tree. Roslyn provides a rich set of functions to work with the syntax tree and the semantic model. Furthermore, it supports straightforward integration into Visual Studio as an extension or NuGet package.

General restrictions of the implementation are that it is limited to what information Roslyn provides for the currently analyzed document. This means that there is no data collection across documents. But differently to the syntax tree, the semantic model may hold information of documents other than the currently analyzed. For example, type information. A type must not be declared within the same document to retrieve general information like the full type name.

The upcoming sections describe different parts of the implementation. Section 3.1 briefly describes general functions used by the analyses which represent basic Roslyn and C# functionalities. The utilized key techniques —functions that are used across multiple pattern analysis— are introduced in Section 3.2. These key techniques are functionalities that are not provided by Roslyn and have to be implemented manually. In Section 3.3, the analyses applied to identify the patterns are explained in detail with remarks of their drawbacks.

3.1 Notation

Within this thesis, different analysis methods are described with mathematical symbols. The basic notation is outlined in Table 3.1. Uppercase letters like N represent sets and lowercase like n a single entity. Functions that have a name in the plural will return a set whereas names in the singular will return a single entity.

The illustrated functions primarily reflect basic functionalities of *Roslyn* and *C#*. For example, consider the $Constructors(N)$ function: It retrieves all constructor declarations within the set N. This is achieved with the *LINQ* filter `OfType` as illustrated in Listing 3.1. The illustrated code shows the extension method `Constructors` which selects any node of the type `ConstructorDeclarationSyntax` within the given enumerable `nodes`. The `nodes` enumerable represents thereby the set N.

As all the other functions illustrated in Table 3.1 can be implemented as straightforward as $Constructors(N)$, further description is omitted.

Notation	Definition
S_*	The set of nodes of the analyzed syntax tree
$Modifiers(n)$	Set of modifiers of n
$Parent(n)$	Parent node of n
$Symbol(n)$	Symbol of n
$Constructors(N)$	Set of constructors in N
$Nodes(n)$	Set of nodes of n
$Identifiers(N)$	Set of identifiers in N
$Fields(N)$	Set of fields in N

Table 3.1: General Notation - N represents a set and n a single entity

```
1 public static IEnumerable<ConstructorDeclarationSyntax>
2     Constructors(this IEnumerable<SyntaxNode> nodes) {
3   return nodes.OfType<ConstructorDeclarationSyntax>();
4 }
```

Listing 3.1: C# Representation of $Constructors(N)$

3.2 Key Techniques

This section introduces several key techniques applied across the different analysis implementations. These are functions that are not provided by Roslyn; hence they have to be implemented manually.

3.2.1 Type Resolution

With the help of the semantic model, it is possible to retrieve the exact types of a symbol. The drawback is, the resolved type is represented by the type `Microsoft.CodeAnalysis.ITypeSymbol` which is not compatible with the type `System.Type` that is known from reflection. There is no straightforward approach to mapping these types at this time.

To overcome this, an own type model —uniform type— is introduced. This type model provides interfaces to efficiently check if two types are compatible, no matter what type representation is beneath. It can encapsulate both previously mentioned type representations. Furthermore, types may also be constructed manually. This manual construction can be helpful to avoid making use of assemblies solely for type resolution. Also, individual assemblies and types cannot be referenced from portable class libraries. One reason is that a portable class library may only reference another portable class library. One example of a type not available in a portable class library is `System.Threading.Thread`.

Each uniform type implementation provides the possibility to retrieve the fully qualified name. These names are then used internally to ensure the type compatibility (string comparison). In future, it is desirable to replace this mechanic with a more sophisticated approach. Since the comparison methods of the uniform types are completely transparent, they can be easily enhanced.

3.2.2 Member Identification

Sometimes it is necessary if a certain syntax node accesses a member of a specific type. The identification if a node accesses a member of a particular type is achieved by retrieving the symbol of the syntax node. For example, an invocation expression would result in an `IMethodSymbol`. But it is sufficient to have the generalized symbol type `ISymbol`. Of this symbol, the containing type is checked with the help of the uniform types introduced in Section 3.2.1. If this type is the expected type, it is guaranteed that the node is accessing a member of this type. Otherwise, either the containing type would be different to the expected or the symbol retrieval will fail.

$$IsMemberOf(s, t) = (s \neq \emptyset \wedge t = ContainingType(s))$$

The identification, if a specific member is accessed is achieved by doing a string-equality check on the name.

$$IsMember(s, t, n) = (IsMemberOf(s, t) \wedge n = Name(s))$$

A small drawback of this method is that it cannot distinguish between overloads. To overcome this, the arguments would have to be checked as well.

3.2.3 Reference Resolution

In this thesis, references are identifiers that refer to a certain syntax node within the syntax tree. An identifier which represents a method name references the method declaration. Identifiers that represent variables reference the expressions of assignments. As a variable may receive multiple assignments, a variable can reference multiple different nodes. Such references are commonly used within C#. For example when creating a thread object. Instead of defining the thread's body inline as a lambda expression, an identifier of a method can be passed as a delegate. Therefore, a limited approach to reference resolution is necessary and explained within this section.

First of all, if the node is not an identifier, it cannot reference something, that is why the node itself is returned. This case makes it possible that usages of this function do not need to handle it. If the symbol referenced by the node is a method, the declaration of the method is resolved. If the symbol is a variable, all the right-hand sides of the assignments to this variable are returned.

$$ReferencedNodes(n) = \begin{cases} \{n\}, & \text{if } n \notin Identifiers(S_*) \\ MethodBodies(n), & \text{if } Symbol(n) \in Methods(S_*) \\ AssignedNodes(n), & \text{if } Symbol(n) \in Variables(S_*) \end{cases}$$

The $AssignedNodes(i)$ function resolves all the nodes assigned to the given identifier somewhere within the syntax tree. This includes all assignments and the initializer of a variable declaration.

$$\begin{aligned} AssignedNodes(i) = &\{Right(a) \mid Symbol(i) = Symbol(Left(a)) \\ &\wedge a \in Assignments(S_*)\} \\ &\cup \{Initializer(d) \mid Symbol(i) = Symbol(d) \\ &\wedge d \in Declarators(S_*)\} \end{aligned}$$

The $MethodBodies(i)$ function works very similar, but it only searches for method declarations instead of assignments.

$$MethodBodies(i) = \{m \mid Symbol(i) = Symbol(m) \land m \in Methods(S_*)\}$$

It is important to note that this mechanism does not resolve aliases, or in other words: references of references. This would require recursion which ensures that it does not run infinitely as there is the possibility that variables alias each other (a loop). An alternative to recursion would also be an iterative worklist algorithm.

3.2.4 Resolving Write Accesses

To collect all write accesses to a variable, the function $GetWriteAccesses(i)$ is introduced. To accomplish this, a function to check if a node is a write access to the symbol of the desired identifier is required. A node is a write access if it is an assignment, an unary expression or an argument with side-effects. Of the resolved node kind, it is then necessary to check if the symbol of the accessed identifier is equal to the provided symbol.

$$IsWriteAccess(n, s) = \begin{cases} s = Symbol(Left(n)), & \text{if } n \in Assignments(S_*) \\ s = Symbol(Operand(n)), & \text{if } n \in Unary(S_*) \\ IsSideEffect(n, s), & \text{if } n \in Arguments(S_*) \\ false, & \text{otherwise} \end{cases}$$

Besides checking if an argument is the searched symbol, it is further analyzed if they have side-effects. This is the case if they a have the modifier `ref` or `out`. If this check were not made, simple read accesses would be collected as well.

$$\begin{aligned} IsSideEffect(a, s) =& (ref \in Modifiers(a) \lor out \in Modifiers(a)) \\ & \land (s = Symbol(a)) \end{aligned}$$

With the help of the $IsWriteAccessTo(n, i)$ function, it is now possible to construct a set of all write accesses to the desired identifier. That is, any node satisfying this condition.

$$GetWriteAccesses(i) = \{n \mid IsWriteAccess(n, Symbol(i)) \land n \in S_*\}$$

Because this method uses symbols to identify the write-access to a variable, there is no issue if multiple variables have the same identifier. This is because each variable will have its own symbol, no matter if they share their identifiers. Even if the analyzed code applies variable shadowing, this behavior stays the same, meaning they will receive different symbols.

3.2.5 Enclosing Statements

For some analyses, it is necessary to know the nesting of various statements within the syntax tree. Therefore, a function which resolves such nesting is required. This is accomplished by recursively checking the parent node of a node

is part of the desired set N.

$$Enclosing(c, N) = \begin{cases} \emptyset, & \text{if } c = \emptyset \\ c, & \text{if } c \in N \\ Enclosing(Parent(c), N), & \text{otherwise} \end{cases}$$

However, this only resolves the first occurrence of an enclosing node. Sometimes it is necessary to collect all the enclosing nodes of a selected set. This is achieved by not stopping the recursion at the first occurrence and storing each resolved occurrence in the result set.

$$AllEnclosing(c, N) = \begin{cases} \emptyset, & \text{if } c = \emptyset \\ \{c\} \cup AllEnclosing(Parent(c), N), & \text{if } c \in N \\ AllEnclosing(Parent(c), N), & \text{otherwise} \end{cases}$$

As the syntax tree does not have an infinite depth, the recursions will terminate in any case.

Nevertheless, if an analysis heavily uses any of these functions, a performance impact is inevitable. Therefore, some analyses implement the semantic of this function as a visitor which tracks the enclosing nodes of the desired set.

3.3 Pattern Detection

This section describes the different detection mechanisms of the bug patterns introduced in Chapter 2. Each detection mechanism is presented with an abstract representation of the actual implementation. Additionally, a description of the known drawbacks is given as well. These are, for example, certain code constellations that are not respected by the analysis.

As already mentioned in the introduction of this thesis, all the analyses are strictly based on the assumption that code that is made to run concurrently will run concurrently. This assumption is applied across the whole document. This means if one method uses certain synchronization primitives it is assumed that every other method will run concurrently and has to use synchronization too. As this is an adamant assumption that affects the analysis of a complete document, it could be necessary to relax it. An idea could be, that a method is only considered to run concurrently if it uses any of the synchronization primitives. While this relaxation would decrease the chance of false positives, it would increase the chance of false negatives, especially for one-liners commonly used in property declarations.

3.3.1 Unsafe Accesses to 64-bit Values

Operations that the implementation considers as atomic are all members of `System.Threading.Interlocked` and `System.Threading.Volatile`. Additionally, the implementation respects the members `VolatileRead` and `VolatileWrite` of `System.Threading.Thread` too.

$$\begin{aligned} AtomicMethods = & Members(Interlocked) \cup Members(Volatile) \\ & \cup \{Thread.VolatileRead, Thread.VolatileWrite\} \end{aligned}$$

Because not all arguments of these methods are part of the actual atomic operation, only arguments declared with the `ref` modifier are acknowledged as being accessed atomically.

$$\begin{aligned} AtomicArguments = \{a \mid & ref \in Modifiers(a) \\ & \wedge Symbol(Parent(a)) \in AtomicMethods \\ & \wedge a \in Arguments(S_*)\} \end{aligned}$$

The identification if a symbol is a 64 bit typed variable is achieved by checking its type against the known 64 bit types.

$$Is64BitVariable(s) = Type(s) \in \{long, double\}$$

The analysis starts by collecting all the symbols of the atomic arguments referencing a 64 bit typed variable.

$$\begin{aligned} AtomicAccessed64 = \{Symbol(a) \mid & Is64BitVariable(Symbol(a)) \\ & \wedge a \in AtomicArguments\} \end{aligned}$$

Finally, all usages of the identified variables are analyzed if they are accessed every time with an atomic operation. This is achieved by checking all the identifiers of the syntax tree. Any non-atomic access to a variable that was once found to be atomically accessed is reported as a potential bug. Additionally, nodes being part of the constructor are excluded.

$$\begin{aligned} UnsafeAccesses = \{i \mid & Symbol(i) \in AtomicAccessed64 \\ & \wedge Parent(i) \notin AtomicArguments \\ & \wedge Enclosing(i, Constructors(S_*)) = \emptyset \\ & \wedge i \in Identifiers(S_*)\} \end{aligned}$$

Drawbacks

The exceptions already mention that all the constructing structures should be considered, but the detection only respects nodes that are explicitly part of the constructor. Therefore, initialization methods that are solely invoked by the constructor are not excluded from the report. Supporting this case would require inter-procedural analysis.

Another possible issue could be the use of lambdas within the constructor. Consider a lambda expression that writes to a shared 64-bit field. If this lambda is made accessible outside of the constructor (e.g. by assigning it to a variable), the write-accesses inside the body of the lambda expression have to be reported. This would be necessary, even though the nodes are part of the constructor.

3.3.2 Insufficient Double-Checked Locking

Initially, the static fields without the `volatile` modifier and an initializer are collected. If this set is empty, no further analysis is necessary.

$$\begin{aligned} StaticFields = \{Symbol(f) \mid & static \in Modifiers(f) \\ & \wedge \emptyset = Initializer(f) \\ & \wedge f \in Fields(S_*)\} \end{aligned}$$

The check for double-checked locking is made by searching for all assignments to the previously identified static fields. If an assignment is found, it is ensured that it is an object creation. If that is the case, the syntax nesting is checked if the double checked locking pattern is present. That is if an if statement is enclosed by a lock statement which itself is again enclosed by an if statement.

$$\begin{aligned} InsufficientDCL = \{a \mid & i_{outer} \neq \emptyset \\ & \wedge i_{outer} = Enclosing(l, IfStatements(S_*)) \\ & \wedge l = Enclosing(i_{inner}, LockStatements(S_*)) \\ & \wedge i_{inner} = Enclosing(a, IfStatements(S_*)) \\ & \wedge Symbol(Left(a)) \in StaticFields \\ & \wedge Right(a) \in ObjectCreations(S_*) \\ & \wedge a \in Assignments(S_*)\} \end{aligned}$$

Because the detection uses block nesting to identify the pattern, only double-checked locking implemented with the lock-statement is detected. Other implementations that use, for example, `SemaphoreSlim` would require a control and possibly data flow analysis.

Drawbacks

The analysis only considers double-checked locking that is implemented with the help of lock statements. Furthermore, if the implementation somehow differs from the conventional implementation style (e.g. by reduction of the nesting), the presence of double-checked locking is not detected.

Additionally, only double-checked lockings for `static` fields are detected. As the detection is probably too restrictive, the filter for the `static` modifier could be removed.

3.3.3 Unguaranteed Loop Variable Visibility

To identify the write-accesses to a certain variable inside a syntax tree, the function $GetWriteAccesses(i)$ introduced in Section 3.2.4 is used. Out of the resolved write-accesses, the ones inside constructors are excluded.

$$\begin{aligned} NoConstructorWrites(f) = \{f \mid & \emptyset = Enclosing(f, Constructors(S_*)) \\ & \wedge f \in GetWriteAccesses(f)\} \end{aligned}$$

The detection is accomplished by checking if a field without the `volatile` modifier used as a loop condition is updated only outside of the loop. This is the case if none of the write accesses is inside the loop's body.

$$\begin{aligned} IsOnlyUpdatedOutsideOfBody(f, w) = & (NoConstructorWrites(f) \neq \emptyset) \\ & \wedge((NoConstructorWrites(f) \cap Nodes(w)) \neq \emptyset) \end{aligned}$$

All the accessed variables inside the loop's condition have to be checked. Though only accessed fields without the `volatile` modifier are subject for further analysis. Any of these collected fields is then checked if all the write accesses to it are

made outside of the loop.

$$SharedCandidatesOfLoop(w) = \{i \mid IsOnlyUpdatedOutsideOfBody(i, w) \\ \wedge volatile \in Modifiers(Symbol(i)) \\ \wedge i \in Identifiers(S_*) \\ \wedge i \in Nodes(Condition(w))\}$$

Now being able to collect the shared identifiers a loop uses for its condition, all the fields that satisfy the conditions can be collected. This is accomplished by simply iterating over all present loops within the syntax tree. As mentioned in the description of the pattern, loops that are enclosed by lock-statements should be excluded. This is achieved with the help of the $Enclosing(c, N)$ function described in Section 3.2.5.

$$LoopsWithUntouchedConditions = \{i \mid i \in SharedCandidatesOfLoop(w) \\ \wedge Enclosing(w, Locks(S_*)) = \emptyset \\ \wedge w \in WhileStatements(S_*)\}$$

Drawbacks

The analysis is limited to intra-procedural analysis. It only sees write-accesses that are directly within the loop's body. If a variable is written outside of it using a method invocation, the variable change is not observed. More critically, it will be reported as a visibility issue.

Avoiding this behavior would require an inter-procedural analysis that resolves write-accesses made through method invocations.

3.3.4 Non-Atomic Assignment on Volatile

The detection is accomplished by analyzing the identifiers of the syntax tree. With the help of the semantic model, it is possible to check if an identifier is a `volatile` field.

$$IsVolatile(i) = Symbol(i) \in Fields(S_*) \wedge volatile \in Modifiers(Symbol(i))$$

Non-atomic operations are all assignments that are not "simple". A simple assignment is with the operator `=` whereas a non-simple would be with an operator of the style `+=`. Therefore, any `volatile` variable being accessed with a non-simple assignment has to be collected.

$$NonAtomicAssignments = \{Left(a) \mid IsVolatile(Left(a)) \\ \wedge a \in NonSimpleAssignments(S_*)\}$$

As read and write access can also be achieved as unary expressions of the style `++`, they have to be taken into account too.

$$NonAtomicUnary = \{Operand(a) \mid IsVolatile(Operand(a)) \\ \wedge u \in Unary(S_*)\}$$

All the nodes that are being collected by the previous two rules belong to the introduced pattern. To ignore cases that apply monitor synchronization

when performing write-accesses, identifiers enclosed with a lock-statement are excluded. The same exception is made for nodes being enclosed by a constructor declaration.

$$\begin{aligned} NonAtomic = \{v \mid & Enclosing(v, LockStatements(S_*)) = \emptyset \\ & \wedge Enclosing(v, Constructors(S_*)) = \emptyset \\ & \wedge v \in (NonAtomicAssignments \cup NonAtomicUnary)\} \end{aligned}$$

One interesting thing here is the use of the semantic model. As the semantic model allows to identify if a variable is `volatile`, the analysis is not bound to variables declared within the same document.

Drawbacks

The drawbacks are the same as the ones described by the pattern DR_UA64V in Section 3.3.1.

3.3.5 Unsafe Concurrent Collection Access

The analysis tries to map queries that check for a particular state of the collection to succeeding actions. This is accomplished by respecting the nesting of the action by one or more `if` statements. Therefore, it is required that the different interactions with collections can be identified. Table 3.2 lists the members of each concurrent collection type that could be used after state queries. So as a first step, collecting these actions within the current document is necessary. Besides that, the special case for concurrent dictionaries has to be respected, that an element can be accessed using the element access operator. So retrieving the affected symbol —the expression— out of an invocation or element access allows the identification of the affected variable by resolving its symbol. If this variable is of any of the concurrent collection types, it has then to be checked if it is any of collection actions mentioned in Table 3.2.

$$\begin{aligned} Actions = \{(s, m) \mid & IsCollectionAction(n) \\ & \wedge Type(s) \in ConcurrentCollectionsTypes \\ & \wedge s = Symbol(Expression(n)) \\ & \wedge n \in (Invocations(S_*) \cup ElementAccesses(S_*))\} \end{aligned}$$

The next step is to identify the state queries. This is accomplished by checking the conditions within each of the enclosing `if` statements. Firstly, the member accesses of the conditions of the enclosing `if` statements have to be identified. Analogous to the step of the action identifications, the symbol of the member access is resolved and the type checked against the concurrent collection types. The remaining check is if the member access is a collection query. The

Collection Type	Members
BlockingCollection	Add, Take, TryAdd, TryTake
ConcurrentBagType	Add, TryPeek, TryTake
ConcurrentDictionaryType	AddOrUpdate, GetOrAdd, TryAdd, TryGetValue, TryRemove, TryUpdate
ConcurrentQueueType	Enqueue, TryDequeue, TryPeek
ConcurrentStackType	Push, PushRange, TryPeek, TryPop, TryPopRange

Table 3.2: Action Members of Concurrent Collections

Collection Type	Members
BlockingCollection	Count
ConcurrentBag	Count
ConcurrentDictionary	ContainsKey, Count
ConcurrentQueue	Count
ConcurrentStack	Count

Table 3.3: Querying Members of Concurrent Collections

possible queries are illustrated in Table 3.3.

$$\begin{aligned} Queries = \{(s,q) \mid & IsCollectionQuery(m) \\ & \wedge Type(s) \in ConcurrentCollectionsTypes \\ & \wedge s = Symbol(q) \\ & \wedge q \in (Nodes(Condition(i)) \cap MemberAccesses(S_*)) \\ & \wedge i \in AllEnclosing(a, IfStatements(S_*))\} \end{aligned}$$

The queries and actions are grouped by the symbol of the accessed variable. Therefore, the last step is to map the queries with their respective action. The mapping is made with the help of the symbol. Two equal symbols denote the possibility of accessing the same variable. If there is a combination found, a potential race condition has been identified.

$$UnsafeAccesses = \{(a,q) \mid s_a = s_q \wedge (s_a, a) \in Actions \wedge (s_q, q) \in Queries\}$$

Drawbacks

The analysis does not check if there is any real relation between the query and the following action. For example, the key provided to `ContainsKey` of `ConcurrentDictionary` could be completely different to the key used for the actual action.

Another drawback is that the analysis only works with the nesting of `if` statements and their condition. If the query is made outside of the `if` statement

Node Kind	Read Access	Write Access
Argument with `ref` Modifier	✓	✓
Argument with `out` Modifier	✗	✓
Argument without Modifier	✓	✗
Simple Assignment (`=`)	✗	✓
Non-Simple Assignment (such as `+=` and `-=`)	✓	✓
Unary Expression	✓	✓
Identifiers in other Expressions	✓	✗

Table 3.4: Identification of Read and Write Accesses of a Node

and the result stored in a variable, the potential bug is not detected. Also, avoiding nesting of `if` statements by using them as a guard in combination with control flow breaking statements like `return` will make the analysis fail.

3.3.6 Parallel For with Side Effects

The basic idea of the analysis is collecting all variables declared outside of the loop that are read and written within the loop. So first of all, it is necessary to identify which kind of accesses a node represents to a variable. These access kinds are listed in Table 3.4.

All the write accesses need to be collected. The collection is made by checking all nodes of the loop's body if they are a write access. If they are a write-access, the affected identifier is stored. To respect the possibility of accesses to array elements, these are simply excluded.

$$\begin{aligned} WrittenExpressions(l) = \{w \mid & \emptyset = (Nodes(w) \cap ElementAccesses(S_*)) \\ & \wedge w = Written(n) \\ & \wedge n \in Nodes(l)\} \end{aligned}$$

If a a node is a write-access or not is achieved with the identification according to Table 3.4.

$$Written(n) = \begin{cases} Operand(n), & \text{if } n \in Unary(S_*) \\ n, & \text{if } n \in Arguments(S_*) \wedge \emptyset \neq Modifiers(n) \\ Left(n), & \text{if } n \in Assignments(S_*) \\ \emptyset, & \text{otherwise} \end{cases}$$

Analogous to the write-accesses, all the read accesses need to be collected. Additionally, all the identifiers representing variables not being explicitly identified as a write-target have to be included as well.

$$\begin{aligned} ReadExpressions(l) = & \{r \mid \emptyset = (Nodes(r) \cap ElementAccesses(S_*)) \\ & \wedge r = Read(n) \\ & \wedge n \in Nodes(l)\} \\ & \cup (Identifiers(Nodes(l)) \setminus WrittenIdentifiers(l)) \end{aligned}$$

Again, if a node is a read-access or not is determined with the help of Table 3.4.

$$Read(n) = \begin{cases} Operand(n), & \text{if } n \in Unary(S_*) \\ n, & \text{if } n \in Arguments(S_*) \wedge out \notin Modifiers(n) \\ Left(n), & \text{if } n \in Assignments(S_*) \wedge simple \neq Kind(n) \\ \emptyset, & \text{otherwise} \end{cases}$$

To identify if a variable was declared outside of the loop, it is simply checked if the declaration is part of the loop's body. This can easily be achieved by retrieving the symbol of the identifier and then resolving the declaring syntax.

$$IsDeclaredOutsideOf(i, b) = Declaration(Symbol(i)) \notin Nodes(b)$$

The final step is to compare the sets of read and written identifiers. This is made for each parallel loop separately. To recognize if identifiers access the same variable, the symbol is utilized. If the symbols of both identifiers are the same, the identifiers access the same variable. Of course, only variables declared outside of the loop are of interest.

$$\begin{aligned} PossibleSideEffects = \bigcup \{\{r, w\} \mid & IsDeclaredOutsideOf(r, l) \\ & \wedge Symbol(r) = Symbol(w) \\ & \wedge r \in ReadExpressions(l) \\ & \wedge w \in WrittenExpressions(l) \\ & \wedge l \in ParallelLoops(S_*)\} \end{aligned}$$

Drawbacks

Race conditions that are not direct part of the delegate or lambda being invoked by parallel loop are not detected. This are, for example, the ones of methods being invoked by the executed delegate or lambda.

Moreover, the filter for array accesses is rather radical. It does not check if there is the possibility that the accesses to said array could be overlapping. Also, if the element access does not represent an access to an array, this is an error. For example, such element accesses can also be achieved for collection types like dictionaries.

An enhancement to the last step should be considered as well. There is the chance that the `ref` modifier denotes the use of a method of `Interlocked` or similar. So it might be a reasonable idea to check if all the accesses made to the variable is only made with an atomic operation. If that is the case, they can be excluded from the analysis.

3.3.7 Interruption of not Interruptible Thread

The detection is accomplished by initially collecting all thread objects being interrupted using `Thread.Interrupt` of the syntax tree.

$$InterruptedThreads = \{t \mid IsInterrupted(t) \wedge IsThreadObject(t) \wedge t \in S_*\}$$

These thread objects are most likely variables, so the possible values of these variables have to be resolved. Furthermore, they may have multiple possible

values. To find them, the symbol is used, and any assignment to this symbol within the syntax tree is part of the result. For simplification, only assignments of object constructions of the style `new Thread(...)` are considered, leaving the possibility to analyze threads created with factory methods.

$$\begin{aligned} ThreadCreations(t) = \{Right(a) \mid & Right(a) \in ObjectCreations(S_*) \\ & \wedge Symbol(t) = Symbol(Left(a)) \\ & \wedge a \in Assignments(S_*)\} \end{aligned}$$

The body a thread is constructed with can either be a reference using an identifier or a lambda expression. To resolve the body executed by a thread, the $ReferencedNodes(n)$ function introduced in Section 3.2.3 is used. The Arg function used here returns the argument passed to the thread construction.

$$ThreadBodies(t) = ReferencedNodes(Arg(t))$$

Checking if a method invocation is invoking any of the interruptible methods is accomplished by retrieving the method symbol and checking it against the known interruptible methods.

$$\begin{aligned} IsInterruptibleMethod(i) = Symbol(i) \in \{ & Monitor.Enter, Monitor.Wait, \\ & Thread.Sleep, Thread.Join\} \end{aligned}$$

A node is interruptible if it either is a lock statement or invokes any of the interruptible methods.

$$IsInterruptible(n) = \begin{cases} true, & \text{if } n \in Lock(S_*) \\ IsInterruptibleMethod(n) & \text{if } n \in Invocations(S_*) \\ false, & \text{otherwise} \end{cases}$$

The identified thread bodies can then be analyzed if they can be interrupted. It is the case if any node of the body is known to be interruptible.

$$UsesInterruptibleMethod(b) = IsInterruptible(n) \wedge n \in Nodes(b)$$

With the combination of the defined functions, the not interruptible threads can be collected.

$$\begin{aligned} NotInterruptible = \{t \mid & \neg UsesInterruptibleMethod(b) \\ & \wedge b \in ThreadBodies(c) \\ & \wedge c \in ThreadCreations(t) \\ & \wedge t \in InterruptedThreads\} \end{aligned}$$

Drawbacks

The limited quality of this analysis is primarily due to simple reference resolution of the $ReferencedNodes(i)$ function. It does not resolve references of references. This limits the space of analyzed thread bodies. If it is desired to support this, some kind of recursion or similar methods would be required. Additionally, it is not guaranteed that the resolved references are a target of the interruption. To

improve the correctness of possible thread bodies, a data flow analysis would be required.

A very general issue comes from the missing intra-procedural analysis. There is no guarantee that the interruptible method is not part of any of the invoked methods of the thread's body. This issue could be solved by initially collecting all methods that are invoking one of the known interruptible methods, making them *implicitly* interruptible.

3.3.8 Missing Cancellation Strategy

The basic setup of this analysis is the same as for PF_INIT as described in Section 3.3.7. Therefore, it will re-use some of the already defined functions. However, instead of checking if a thread can be interrupted, it will look if there is any `catch{...}` clause handling `ThreadInterruptedException`. To accomplish this, all the catched exception types within the thread body are collected.

$$Catched(b) = \{Type(c) \mid c \in CatchClauses(n) \wedge n \in Nodes(b)\}$$

Within the set of the catched exceptions, it can be checked if `ThreadInterrupted-Exception` is one of it.

$$CatchesInterruption(b) = ThreadInterruptedException \in Catched(b)$$

The full analysis is –as already mentioned– very similar to the one of PF_INIT, except for the replacement of $UsesInterruptibleMethod(b)$.

$$\begin{aligned} NotInterruptible = \{t \mid & \neg CatchesInterruption(b) \\ & \wedge b \in ThreadBodies(c) \\ & \wedge c \in ThreadCreations(t) \\ & \wedge t \in InterruptedThreads\} \end{aligned}$$

Drawbacks

Because of the same base idea, the drawbacks are the same as the ones described for the pattern PF_MCS introduced in Section 3.3.8.

Additionally, the analysis only checks for explicit handling of the `Thread.Interrupted.Exception`. Although, there is no guarantee that the catch block ends the thread execution. This might be solved with the help of a control flow analysis. Furthermore, general exception handlers which catch `Exception` are not considered, but may as well lead to an end of the thread execution.

Room for improvement also lies in the detected exception. It should be checked if the body of the `try` clause actually invokes any of the interruptible methods. This yields the idea that this analysis should be constructed as a combination with PF_MCS pattern.

3.3.9 String Literal as Lock Object

The detection is accomplished by looking if any of the used synchronization objects may refer a string literal. This can either be a string literal directly

passed to the lock statement or an identifier referencing a string literal.

$$IsString(o) = \begin{cases} true, & \text{if } o \in StringLiterals(S_*) \\ IsStringReference(i), & \text{if } o \in Identifiers(S_*) \\ false, & \text{otherwise} \end{cases}$$

Checking if an identifier references a string literal is accomplished by looking if any of the possibly referenced nodes is a string literal.

$$IsStringReference(i) = ((PossibleReferences(i) \cap StringLiterals(S_*)) \neq \emptyset)$$

The collection of all usages of string literals as lock objects is accomplished by searching for lock statements that have an argument that satisfies the $IsString(o)$ function.

$$StringUsages = \{Arg(l) \mid IsString(Arg(l)) \wedge l \in Locks(S_*)\}$$

Drawbacks

As the reference resolution of $PossibleReferences(i)$ only resolves direct references, not all possible values are analyzed. Also, there is no guarantee that an assigned node is ever used for locking. With the help of a data flow analysis, the space of possible values could be reduced.

3.3.10 Prefer Slim Versions

The detection is straightforward. It has to be identified if an object being created is of any of the fat types.

$$IsFat(c) = Type(c) \in \{Mutex, ReaderWriterLock, Semaphore, ManualResetEvent\}$$

Furthermore, named instances are excluded because of the possibility of inter-process synchronization. This is achieved by checking if any of the arguments passed to the object creation is a `string`.

$$IsNamed(c) = (Strings(S_*) \cap Args(c) \neq \emptyset)$$

Any object creation of the identified fat types which is not named will be reported.

$$FatVersions = \{c \mid IsFat(c) \wedge \neg IsNamed(c) \wedge c \in ObjectCreations(S_*)\}$$

Drawbacks

The assumption that a named instance of a fat-synchronization primitive is used for inter-process synchronization is probably not always true. But it is most likely impossible to identify whether such a primitive is used for inter-process synchronization or not. A possibility to overcome this could be a configuration. This configuration would allow users to tell the analysis that the analyzed software does indeed apply inter-process synchronization.

3.3.11 Prefer Lock over Explicit Monitor

The explicit monitor usage is detected by the presence of an invocation of `Monitor.Enter` or `Monitor.TryEnter`.

$$\begin{aligned} MonitorEntries = \{i \mid & HasArgWithRef(i) \\ & \wedge Symbol(i) \in \{Monitor.Enter, Monitor.TryEnter\} \\ & \wedge i \in Invocations(S_*)\} \end{aligned}$$

To respect the possibility of a correct implementation, only occurrences with arguments without the `ref` modifier are taken into account.

$$HasArgWithRef(i) = ref \in \bigcup \{Modifiers(a) \mid a \in Args(i)\}$$

Only the detected monitor entries are reported. Correctly identifying the corresponding `Monitor.Exit` instruction would require additional effort and complexity, but with only little benefits.

Drawbacks

The use of the `ref` modifier as a monitor argument does not necessarily mean that the explicit synchronization has been implemented correctly. To improve this, it would be necessary to analyze for nesting within a `try...finally` statement and the usage of the argument passed with the `ref` modifier. Moreover, it has to be checked if the `ref` argument is actually used within the `finally` block as a guard for the `Monitor.Exit` instruction. This could be improved with the help of a control flow analysis.

4. Evaluation

This chapter presents the results of the experimental evaluation of *Concurrency Bug Finder*. The analysis was made with a combination of a selection of "mature" projects and a random selection of projects. The publicly available mature projects that were analyzed are listed in Table 4.1.

The random selection was made with the help of the GitHub search function. About the first ten highest rated projects making use of a certain concurrency control (e.g. `Interlocked.Read`) have been included in the evaluation. For all projects, unit test projects have been excluded from the analyses.

4.1 Overview

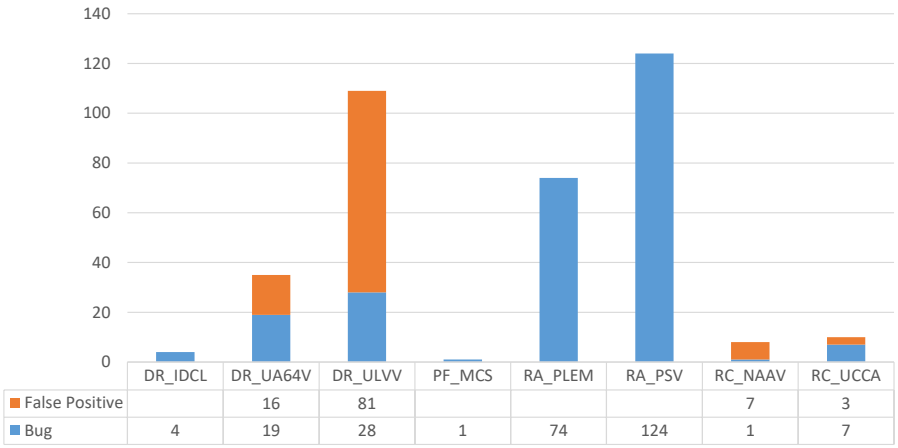
Figure 4.1 sketches an outline over the analysis outcome, whether the finding is a false positive or effectively a bug. The actual numbers of the results are illustrated right below the bars. First of all, the patterns RC_PFSE, PF_INIT, and PF_SLLO are missing in the diagram. That is because there were no errors reported by the *Concurrency Bug Finder* in any of the analyzed projects.

Less compelling are the findings of the patterns RA_PLEM and RA_PSV. Any report of these analyses was correct. This is because of their very restrictive implementation, only allowing reports of findings that are 100% certain. The implementation RA_PLEM only reports usages of monitor synchronization that do not make use of the `lockTaken` parameter when acquiring a monitor lock. The same situation is for the implementation of RA_PSV. It only reports cases which can not be used for inter-process synchronization.

Not really representative are the results of the pattern PF_MCS. The interrupt function is only rarely used, at least when searching for usages on GitHub.

Project	URL
apache-nms	`http://svn.apache.org/viewvc/activemq/`
corefx	`https://github.com/dotnet/corefx`
nhibernate-core	`https://github.com/nhibernate/nhibernate-core`
Rx.NET	`https://github.com/Reactive-Extensions/Rx.NET`
SignalR	`https://github.com/SignalR/SignalR`
spring-net	`https://github.com/spring-projects/spring-net`

Table 4.1: Mature Projects Analyzed in Evaluation



	DR_IDCL	DR_UA64V	DR_ULVV	PF_MCS	RA_PLEM	RA_PSV	RC_NAAV	RC_UCCA
False Positive		16	81				7	3
Bug	4	19	28	1	74	124	1	7

Figure 4.1: False Positives and Correct Findings per Pattern

Also, the code of the error reported is more of a demonstration of how to interrupt a thread (but it has no cancellation strategy). Due to the rare use of interrupt, this situation also applies to the pattern PF_INIT.

A similar situation as for thread interruption was found for `Parallel.For`. The codes found on GitHub are primarily used for demonstration purposes of the .NET feature rather than actual use cases. Therefore, that there were no reports for RC_PFSE is not surprising. More interesting would be how the little use can be reasoned. As this feature is available since .NET 4.0, it can no longer be considered as a novelty.

More interesting are the findings of DR_IDCL. Even though its implementation is quite relaxed —it does not analyze the conditions of the enclosing if statements,— there were no false positives. Also, two out of the four findings are from mature projects. If the analysis implementation would not have been as relaxed as it is by now, it would most likely not have found the construct illustrated in Listing 4.1. This case was found in *corefx* and does not protect the write-access to the singleton variable. Therefore, this implementation could still lead to the situation that other threads see an incompletely constructed object.

DR_NAAV has a rather high chance to result in a false positive. More precisely, more than 85% of the findings are false positives. This is reasoned to the rather complex and large classes of *corefx*. All of the seven false positives were reported by the analysis of *corefx*. Five out of the seven reports were resulting from the analysis of the concurrency framework. The other two were from classes that make use of `volatile`, even though they are not meant for concurrent usage. In these two cases, it should be considered whether the use of `volatile` has any use at all and if this should be really judged as a false positive.

The results of the remaining three analyses —DR_UA64V, DR_ULVV, and RC_UCCA— are discussed in more detail in Sections 4.2 and 4.3 by providing ideas for possible improvements of the analyses.

```
1  public static SocketPerfCounter Instance {
2    get {
3      if(Volatile.Read(ref s_instance) == null) {
4        lock(s_lockObject) {
5          if(Volatile.Read(ref s_instance) == null) {
6            s_instance = new SocketPerfCounter();
7          }
8        }
9      }
10     return s_instance;
11   }
12 }
```

Listing 4.1: Only Partially Volatile Double Checked Locking

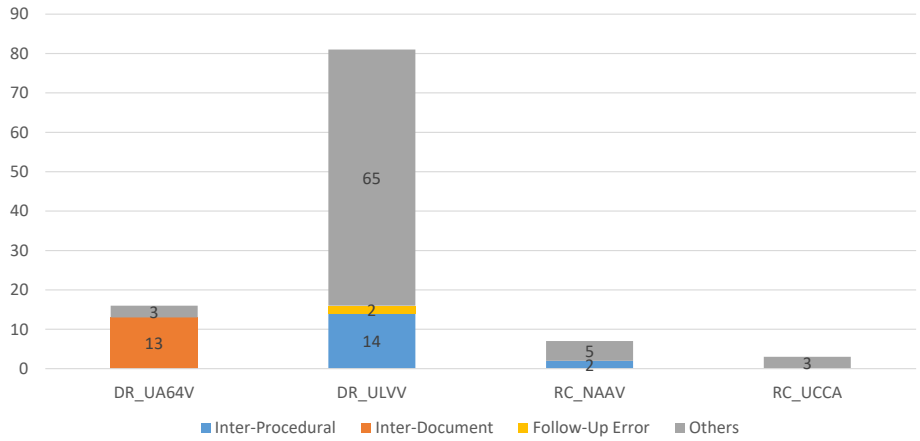


Figure 4.2: Number of False Positives Reduced by Analysis Extension

4.2 Possibility for Improvements

Having some undesired false positives, the question arises how these could be reduced. Figure 4.2 illustrates the false positives per pattern and with which technique how many could be avoided.

As it can be seen, more than 81% of the false positives of the pattern DR_UA64V can be eliminated by applying an inter-document analysis. However, it should be remembered that *Roslyn* does only support single-document analyzer implementations for use in *Visual Studio*. Doing so would either require an improvement of *Roslyn* or custom extensions which support this. There remain three false positives. One of them is not straightforward as the enclosing method is used for object construction but is invoked from outside of the document. The other two are more of a cosmetic error. The access to the variables is made using `Interlocked`. But the argument passed is not directly the identifier of the variable but a member access of `this`. Currently, the analyzer only checks if the parent syntax node of the identifier is an argument passed to an atomic operation. But in these cases, the parent node is a member access.

So instead of retrieving the direct parent with $Parent(i)$ which could be an argument, it should check if any of the enclosing nodes is an argument with $GetEnclosing(i, Arguments(S_*))$.

It has already been mentioned in Section 4.1 that five false positives of RC_NAAV are reasoned to the analysis of the concurrency framework of *corefx*. Reasoning a possible fix for such large classes is difficult. But it appears that they all implement an inner class whereas the false positive results from a public method which is only invoked by the containing type's constructor. So it could be possible to fix some of the false positives with the help of an inter-procedural analysis.

One of the three false positives of RC_UCCA is reasoned to the constellation of the accesses to the concurrent dictionary. A state query is followed by the deletion of the entry. But the other dictionary accesses make the situation impossible that the entry changed in-between the query and the deletion. The other two of the three false positives are resulting from the analysis of *corefx*. Due to the complexity of the code —both findings are part of the same class,— both are considered as false positives without further investigations.

Due to the high count of false positives by DR_ULVV, the results of the analysis are examined in more detail in Section 4.3.

4.3 Detailed Evaluation of DR_ULVV

For DR_ULVV, only about 20% of the false positives could be removed by applying a data flow and an inter-procedural analysis. It should be noted, that the two that could be fixed with a data flow analysis are resulting from a correct finding of another pattern. More precisely, they are enclosed by the explicit usage of monitors rather than the lock statement. Thus, they are rather a follow-up error than a false positive. The other 65 false positives are reasoned to the fact that the code does not run concurrently. Therefore, the analysis of the pattern is not strict enough.

Figure 4.3 depicts the distribution of the findings by the number of different variables accessed within the loop's condition. As it can be seen, the number of false positives of loops with only a single variable access is a little more than a third. With loops that make use of two variables, the number of false positives is 88% and increases with the number of variables. So, it should be considered introducing a threshold for the number of variables.

Reviewing the analysis results has shown a few vast possibilities for improvements. Figure 4.4 shows how the false positives are distributed among a certain code constellation. The extension of the implementation with an inter-procedural analysis would prevent around 17% of the overall false positives. Other 28% of the false positives are reasoned to the situation, that a variable is implicitly updated when its containing object is written. This is a common situation when iterating over structures like linked lists as illustrated in Listing 4.2. In this example, the current implementation would yield a false positive on line 3 for the variable `Next` because it is written on line 11. However, as it can be seen, the `Next` variable (or better symbol) implicitly changes its value as its enclosing object `current` is updated on line 4. Another 5% are reasoned to the opposite situation. A variable which is not written inside the loop's body has a member that is either updated within the loop's body, a method invocation or simply a

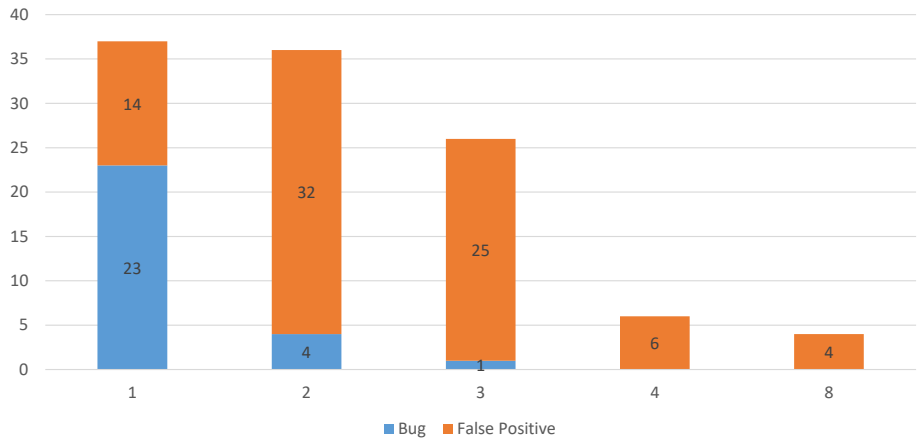


Figure 4.3: False Positive Dispersion of DR_ULVV by Variable Usage

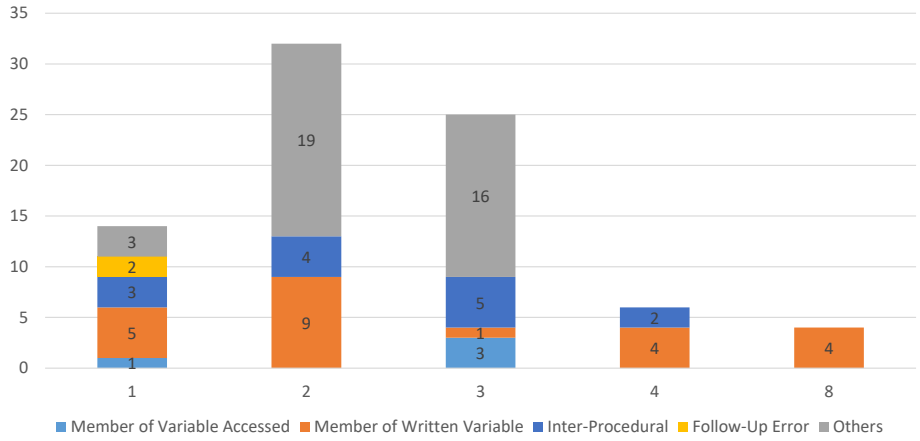


Figure 4.4: Observations of the False Positives of DR_ULVV by Variable Usage

```
1  public T Tail() {
2    Node current = head;
3    while(current?.Next != null) {
4      current = current.Next;
5    }
6    return current?.Value ?? default(T);
7  }
8
9  public void Add(T value) {
10   Node node = new Node(value);
11   node.Next = head;
12   head = node;
13 }
```

Listing 4.2: Example of a Member of a Written Variable

```
1  public void ReplaceAllWith(T value) {
2    var i = 0;
3    while(i < count) {
4      data[i] = value;
5      i++;
6    }
7  }
8
9  public void Add(T value) {
10   data[count] = value;
11   count++;
12 }
```

Listing 4.3: Example of Boundary Variable

property access. Thus, it should be considered limiting the analysis to the final variable of a member-access chain rather than each by itself.

A wide part of the false positives marked as "Others" are reasoned to the fact that the reported variable is used as a boundary variable. This situation is illustrated in Listing 4.3. The read-access of the variable `Count` on line 3 would be reported because it is written on line 11.

Summarizing the observations of this pattern, a first improvement would be introducing a threshold for the number of variables accessed inside a loop's condition. When setting the threshold to one variable access and respecting the illustrated situations, almost all false positives can be avoided.

5. Conclusion

Chapter 2 introduced eleven different concurrency related bug patterns. These patterns were awarded a severity by considering the locality and observability of the effects, and the system's stability after the occurrence. These severities range from *low* for general improvement suggestions to *critical* for issues that can spread over the whole system. Each pattern has been introduced with a characterization of the underlying problem and a brief description how the pattern's presence can be observed within a given code.

The basic principles of the implementation of *Concurrency Bug Finder* were introduced in Chapter 3. Overall, "Code made to run concurrently will run concurrently." is the core idea behind *Concurrency Bug Finder*. This idea is used to identify concurrently running code within the analyzed document. The implementation restricts itself to simple static code analysis with the help of *Roslyn* and *.NET* features. The implementations of the analyses try to apply the idea how the presence of a pattern can be observed as described in Chapter 2. All the mechanisms introduced make no use of sophisticated static analysis techniques like data flow analysis. Due to their simplicity, they are primarily implemented as *LINQ* expressions and only partially make use of visitors to avoid the heavy use of recursions.

An evaluation of the effectiveness of *Concurrency Bug Finder* has been performed by scanning a combination of random and mature projects. The results are introduced in Chapter 4 and have shown that the simple analysis methods and relaxed rules are not necessarily a disadvantage. On the contrary, certain implementations are probably more effective because of their relaxations. Out of all the 365 findings, only little more than 29% were false positives, 107 to be more specific. Leaving away the analysis which is purely a recommendation for performance improvements still makes 134 of the findings malign. Last but not least, suggestions how the false positive rate could be reduced further have been given. For example, making the implementations inter-procedural could reduce the false-positives by 15%. Overall, the implementation of DR_ULVV appeared to be the most unreliable —75% of all false positives— hinting that is too relaxed. Thus, the data has been investigated in more detail. This has shown that simply introducing a threshold would reduce the false positives by more than 82% while only marginally increasing the false negatives.

Figures

Tables

Listings

Bibliography

[1] Joseph Albahari and Ben Albahari. *C# 6.0 in a Nutshell: The Definitive Reference*. O'Reilly Media, Inc., 2015.

[2] Brian Goetz. Double-checked locking: Clever, but broken. http://www.javaworld.com/article/2074979/java-concurrency/double-checked-locking--clever--but-broken.html, 2001. Accessed: 2016-05-12.

[3] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[4] Stack Exchange Inc. stackoverflow. http://stackoverflow.com/. Accessed: 2016-10-24.

[5] Microsoft. C# Language Specification Version 5.0. https://www.microsoft.com/en-us/download/details.aspx?id=7029. Accessed: 2016-10-14.

[6] Microsoft. Double structure - thread safety. https://msdn.microsoft.com/en-us/library/system.double(v=vs.110).aspx#Anchor_8. Accessed: 2016-09-15.

[7] Microsoft. Exception Handling (C# Programming Guide). https://msdn.microsoft.com/en-us/library/ms173162.aspx. Accessed: 2016-11-04.

[8] Microsoft. Int64 structure - thread safety. https://msdn.microsoft.com/en-us/library/system.int64(v=vs.110).aspx#Anchor_6. Accessed: 2016-09-15.

[9] Microsoft. ManualResetEvent and ManualResetEventSlim. https://msdn.microsoft.com/en-us/library/5hbefs30(v=vs.110).aspx. Accessed: 2016-15-08.

[10] Microsoft. Monitor Class. https://msdn.microsoft.com/en-us/library/system.threading.monitor(v=vs.110).aspx. Accessed: 2016-10-18.

[11] Microsoft. Reader-Writer Locks. https://msdn.microsoft.com/en-us/library/bz6sth95(v=vs.110).aspx. Accessed: 2016-15-08.

[12] Microsoft. Roslyn. https://github.com/dotnet/roslyn. Accessed: 2016-09-13.

[13] Microsoft. Semaphore and SemaphoreSlim. https://msdn.microsoft.com/en-us/library/z6zx288a(v=vs.110).aspx. Accessed: 2016-12-05.

[14] Microsoft. Thread.Interrupt Method (System.Threading). https://msdn.microsoft.com/en-us/library/system.threading.thread.interrupt(v=vs.110).aspx. Accessed: 2016-10-03.

[15] Microsoft. volatile (C# Reference). https://msdn.microsoft.com/en-us/library/x13ttww7.aspx. Accessed: 2016-10-17.

[16] Bill Pugh, Andrey Loskutov, Keith Lea, David Hovemeyer, Nay Ayewah, Ben Langmead, Tomas Pollak, Phil Crosby, Peter Friese, Dave Brosius, Brian Goetz, and Rohan Lloyd. FindBugs. http://findbugs.sourceforge.net/. Accessed: 2016-10-03.

[17] Douglas C Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, 2013.

[18] Michael L Scott. Shared-memory synchronization. *Synthesis Lectures on Computer Architecture*, 8(2), 2013.