Practical Detection of Concurrency Issues at Coding Time

Luc Bläser
HSR Hochschule für Technik Rapperswil
Rapperswil, Switzerland
lblaeser@hsr.ch

ABSTRACT

We have developed a practical static checker that is designed to interactively mark data races and deadlocks in program source code at development time. As this use case requires a checker to be both fast and precise, we engaged a simple technique of randomized bounded concrete concurrent interpretation that is experimentally effective for this purpose. Implemented as a tool for C# in Visual Studio, the checker covers the broad spectrum of concurrent language concepts, including task and data parallelism, asynchronous programming, UI dispatching, the various synchronization primitives, monitor, atomic and volatile accesses, and finalizers. Its application to popular open-source C# projects revealed several real issues with only a few false positives.

CCS CONCEPTS

• **Software and its engineering** → **Concurrent programming structures**; **Software defect analysis**;

KEYWORDS

data race; deadlock; static analysis, concurrency, C#

ACM Reference Format:
Luc Bläser. 2018. Practical Detection of Concurrency Issues at Coding Time. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213853

1 INTRODUCTION

Concurrency errors, such as data races, deadlocks, and others, are a common plague in programs developed in C#, Java, C++, Python, and other multi-threaded programming languages. The increasing popularity of models with implicit multi-threading, notably asynchronous and task-based programming, makes the matter even more acute. Because of the non-deterministic occurrence, software developers commonly fear concurrency errors with good reason: These bugs are inherently difficult to detect and reproduce, such that they often remain undetected for a long time or even forever in productive code. Program analysis could help in this regard, but the availability of appropriate tools is scarce: Many of the industrially prevalent tools [15, 23] focus on specific bug patterns and do not detect data races or deadlocks in general. Only a few dynamic concurrency checkers [17, 34, 38] are available and working for current language versions. Static concurrency analysis continues to be an area of research where very few practical tools [26, 36] are on hand. For newer C# versions, there even exists no static checker for data races or deadlocks at all. Previous tools such as CHESS [24] have been discontinued. The situation is discussed in more detail in Section 6.

In this work, we aim to provide a practical tool that detects common concurrency errors in a slightly different setting than other work in this area. This tool should interactively support software developers when working in an integrated development environment (IDE): It should directly highlight problematic program sections with regard to concurrency during the coding. For this purpose, the following checker properties were considered essential:

- **Static**: The source code as displayed in the IDE needs to be analyzed. The code being written can be incomplete or contain erroneous fragments, making a program execution and thus a dynamic analysis impossible.
- **Fast**: Short analysis times are desired to provide quick error feedbacks to the programmers when editing or viewing source code. Ideally, the analysis should only take a few seconds or less, even for large projects.
- **Precise**: Reported warnings should indicate most likely real concurrency errors and infrequent false positives. In the case of many false positives, software developers may quickly ignore the messages and turn off the tool.

Apparently, these requirements imply a decisive compromise: It is impossible to create a static checker that is sound (detects all issues, no false negatives) and precise (detects only real issues, no false positives) at the same time, since this problem is undecidable (similar to the halting problem). This means that when regarding the postulated properties, we must trade soundness against precision. In other words, we deliberately accept a checker that may miss certain errors, i.e. has false negatives, with the purpose that programmers receive as many real errors as possible. Another goal of this checker is to report concurrency errors in a limited short amount of time.

Given this setting, we designed a practical static checker, called HSR Parallel Checker, for the detection of data races and deadlocks in an IDE. Our experiments with an extensive code base of million lines of code of popular open-source projects shows that the tool performs effectively in terms of precision and speed. The technique is deliberately simple and inspired by dynamic analysis, although it is integrated in a static checker. Briefly summarized, we perform a mostly concrete interpretation of the program by simulating randomized thread scheduling. Only for the cases where exact program states are not known, e.g. for external inputs, abstract states are engaged. The interpretation involves multiple runs and

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5699-2/18/07.
https://doi.org/10.1145/3213846.3213853

is bound in interpretation steps and memory, in order to remain responsive in the IDE, regardless of the project size. Seeded pseudo-randomization guarantees reproducibility of the analysis.

The analysis has been implemented as a checker tool that operates on the C# 7 programming language and can be plugged in the Visual Studio IDE. Based on the .NET Compiler Platform (project Roslyn), the checker performs interactive analysis and error highlighting in the code within the IDE. While this serves as proof of concept and as a practical tool at the same time, the technique could just as well be applied to other programming languages.

The checker uses well-known verification techniques, although with differences: Our checker applies mechanisms of dynamic checkers, such as ThreadSanitizer [34] or ConTest [7], however, it remains static, i.e. does not run the program and can deal with unknown program states. We do not aim for a complete (sound) analysis in contrast to model checking, abstract interpretation [5], or concrete interpretation, such as Java Pathfinder [26] or CHESS [24]. Although various counter-measures to the state explosion problem exist, the huge exploration spaces inherent to these approaches are often impractical for the analysis of large projects, especially if the analysis should only take a few seconds. Similar to symbolic execution [1], the presented checker simulates the execution by maintaining as many concrete states as possible and by using symbolic values for unknown external inputs. However, for speed reasons, we do not compute symbol conditions or path constraints, and also omit model checking. A detailed discussion is provided in Section 6.

In summary, we make the following contributions in this paper:

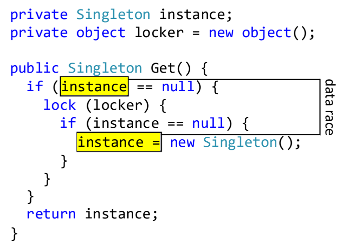
- The description of a heuristic checker method that is practically suited for the interactive static detection of data races and deadlocks in an IDE, a use case where very short analysis times and precision are considered essential.
- The report on the design, implementation, and experimental evaluation of this method in the form of a full-fledged checker tool for C#.
- The discussion of several concurrency errors found in popular C# open-source projects with this tool.

The remainder of this paper is structured as follows: Section 2 defines the detected concurrency errors. Section 3 presents the checker algorithm in general. Section 4 describes its implementation for C#. Section 5 reports on the experimental evaluation of this checker. Section 6 discusses related work. Section 7 finally draws a conclusion.

2 DETECTED ERRORS

We focus on concurrency errors that can be defined at the level of the programming language without understanding program semantics, namely data races and deadlocks.

Data races are pairs of *unsynchronized concurrent accesses on the same memory location, involving at least a write access.* The granularity of a memory location depends on the programming language: In C# and Java, this is the same variable or the same array element. Unsynchronized concurrent accesses mean that two threads each effect an access, without there being a happened-before relation [19] between the accesses. A happened-before relation is established by mutually synchronizing instructions between the



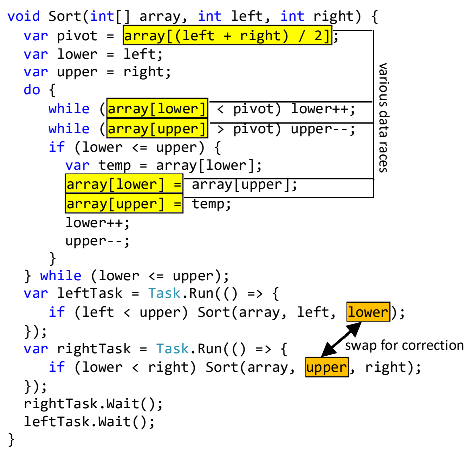
```
private Singleton instance;
private object locker = new object();

public Singleton Get() {
  if (instance == null) {
    lock (locker) {
      if (instance == null) {
        instance = new Singleton();
      }
    }
  }
  return instance;
}
```

Figure 1: Data race in a broken double-checked locking pattern: The write on the field `instance` inside the lock section can happen concurrently with the read outside the lock. Lacking a happened-before relation and corresponding memory fences, instructions in the constructor of `Singleton` could be reordered after the assignment of the created object, such that a not properly initialized object could be returned. Adding the `volatile` keyword to `instance` would eliminate the data race and correct the solution.

two threads. In C# and Java, this is between the release and acquisition of the same lock, the notification and waiting on the same synchronization instance, accesses to the same volatile variable, atomic accesses to the same memory location, on thread starts and joins, on task starts and awaits etc. As mentioned, only read-write, write-read, or write-write accesses are candidates for data races. Data races are a formal error, because the programming language does not specify the program behavior in such a case on purpose, to give freedom for program optimizations. Data races are often involved in race conditions, but not necessarily. For example, race conditions can occur despite the absence of data races: Accesses to shared resources may be all individually synchronized (e.g. a read per se and a write per se), but the granularity of synchronization could be too small (e.g. the read and write should be atomic together). Our checker cannot detect race conditions that do not effect data races, as explained below. Figure 1 shows an obvious case of a data race in a broken double-checked locking pattern, where the volatile keyword on the field `instance` would correct the sample. Figure 2 depicts another more hidden situation of data races in an erroneous parallel quicksort, accompanied by a fix. Our checker is designed to precisely detect both data race cases. As desired, the checker warnings disappear when the corresponding code is fixed.

Deadlocks are constellations where *multiple threads block each other forever in a circular lock dependency.* Each involved thread holds a lock, while awaiting a lock that is held by another thread in this circle. At runtime, deadlocks are equivalent to a cycle in the resource allocation graph [14]. Figure 3 outlines the deadlock in a bank transfer sample, as detected by the checker.

Other concurrency errors, such as race conditions, livelocks, and starvation generally require an understanding of the program semantics: As for race conditions, the knowledge of the granularity of critical sections and semantic state synchronizations would be essential. As for livelocks and starvation, a notion of semantic program progress despite running threads would be necessary. Therefore, we ignore these classes of errors in the checker, and focus on data races and deadlocks.



```
void Sort(int[] array, int left, int right) {
  var pivot = array[(left + right) / 2];
  var lower = left;
  var upper = right;
  do {
    while (array[lower] < pivot) lower++;
    while (array[upper] > pivot) upper--;
    if (lower <= upper) {
      var temp = array[lower];
      array[lower] = array[upper];
      array[upper] = temp;
      lower++;
      upper--;
    }
  } while (lower <= upper);
  var leftTask = Task.Run(() => {
    if (left < upper) Sort(array, left, lower);
  });
  var rightTask = Task.Run(() => {
    if (lower < right) Sort(array, upper, right);
  });
  rightTask.Wait();
  leftTask.Wait();
}
```

Figure 2: An erroneous implementation of the parallel quicksort, with incorrect overlapping boundaries in the recursive sort calls of nested parallel tasks. The code provokes various data races at the level of array elements as it is detected by our checker. By swapping the boundaries as indicated, the code is fixed and the checker no longer reports data races.

```
class BankAccount {
  private int balance;
  private object sync = new object();

  public void Deposit(int amount) {
    lock (sync) {
      balance += amount;
    }
  }

  public void Transfer(BankAccount other, int amount) {
    lock (sync) {
      balance -= amount;
      other.Deposit(amount); // implicit nested lock
    }
  }
}

var account1 = new BankAccount();
var account2 = new BankAccount();
var account3 = new BankAccount();
new Thread(() => account1.Transfer(account2, 5)).Start(); // T1
new Thread(() => account2.Transfer(account3, 5)).Start(); // T2
new Thread(() => account3.Transfer(account1, 5)).Start(); // T3
```

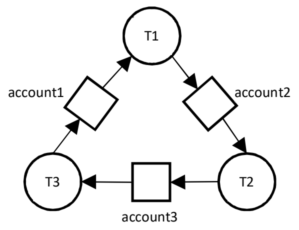


Figure 3: A deadlock situation detected in a bank transfer example. The Transfer-method implicitly acquires nested locks, the first on `this.sync` and a second on `other.sync` through the Deposit-call. The concurrent cyclic transfer may then end up in a deadlock, as depicted in the resource allocation graph.

3 ALGORITHM

Our checker aims to statically identify data races and deadlocks, as defined in the previous section, in a short time and with a high precision, but not necessarily all of them. Similar to concrete or abstract interpretation, we thereby incorporate a dynamic checker technique in a static analysis.

The rough idea is to map the source code to an internal representation and then repeatedly simulate the concurrent execution of this representation on an internal isolated runtime model, with random scheduling and configurable bounds. Thereby, exact program states are maintained, except for unknown external input or incomplete program parts. Concurrency errors happening on the internal model during the simulation are detected and reported as issues. Although the approach is oriented toward a dynamic checker, it is important to note that the analysis is indeed static, i.e. the program is not really executed. The simulation only runs inside the static checker, and does not provoke any side effects. Moreover, it can deal with unknown I/O or incomplete/incorrect program fragments. It comes close to a concrete interpretation except that it engages abstract (symbolic) states in a few necessary cases, uses randomization for thread scheduling, and implies bounds on the exploration. Seeded pseudo-randomization guarantees reproducibility of the checker results. It is also similar to symbolic execution with bounded analysis, however without path constraints and model checking due to performance reasons.

The following sections describe the algorithm in more detail.

3.1 Memory Model

We assume *weak consistency* as the memory model of the analyzed code, as is the case for the C# programming language. This means that memory accesses can be reordered as long as the serial semantics from the perspective of each thread is retained, and as long as memory fences imposed by synchronization operations are not violated. C# offers half fences with acquire- and release-semantics by volatile variable accesses or locks/unlocks, as well as, full fences by atomic operations or dedicated memory barrier instructions.

The checker only interprets instructions in the order of the source code, without simulating any reordering allowed by the weak consistency of the analyzed language. Therefore, the checker's internal interpretation follows *sequential consistency*. As for precision, this is a valid approach since each sequentially consistent schedule also represents a valid schedule on the corresponding weak consistent model. However, the checker may miss data races that only occur with certain reordering due to weak consistency. In our case, this is acceptable as we do not strive for completeness but for precision.

3.2 Interpretation

The source code in the IDE is first transformed to an intermediate representation. In our case, this is a self-designed virtual stack machine code, resembling the .NET intermediate language, although extended with concurrency-specific instructions, such as start a thread, join a thread, acquire a lock, release a lock etc. Already compiled code could also be analyzed and mapped to this intermediate representation. For each method (and constructor etc.), a control-flow graph (CFG) is created, wherein each node reflects an instruction.

This intermediate representation is then interpreted in a sandbox. This maintains a model that represents the entire simulated state of the program as it would be in a real runtime system, although without performing I/O side effects or time consuming operations (no sleeps etc.). The runtime model comprises a set of runnable threads and the object heap. Analogous to a real runtime system, each thread carries its call stack with the method incarnations, holding their local variables and instruction pointer.

Figure 4 shows the steps of the analysis algorithm. Multiple runs are simulated. Per run, a program entry is set up on the empty runtime model depending on the program type. Repeatedly, a random runnable thread is selected and its active instruction according to the CFGs simulated. It is a simple random scheduler, picking one of the runnable threads at random in each step. The checker then interprets the current instruction of the selected thread, effecting the analogous effect on the internal runtime model like in a real runtime system, except for later discussed abstract states. Depending on the state and instruction, this can also block the current thread, i.e. put the thread into a waiting queue, or wake other threads, i.e. set them as runnable again. As described in the subsequent sections, extra information is maintained to detect issues. The step of random thread selection and instruction interpretation is then repeated until a per-run bound is reached. This bound is defined by a limit on the model memory and a maximum number of simulated logical steps. The per-run bound is also reached if there is no runnable thread, i.e. the program is finished or stuck. Once the per-run bound is exceeded, the simulation model is reset and a next run simulated. This continues until an overall simulation bound is reached, defined by a total number of simulated steps of all runs. Finally, the detected issues are reported, by sorting out potential duplicate findings.

Of course, the technique of random thread scheduling and bounded interpretation is heuristic and results in an incomplete analysis, i.e. existing issues may be overseen. The rationale is to only invest a limited amount of time, such that only a subset of schedules can be analyzed. More sophisticated random scheduling [33] would require analysis of runnable threads and thus extra computation time, that is too costly for us. Interestingly, naive random scheduling also shows to be quite effective according to an empirical study [37]. Therefore, we choose the approach of a simple random scheduler. The chance of repeated, and thus unnecessary, schedules could be eliminated by remembering the past schedules in a tree but this effort is not justified as the probability exponentially decreases with the simulated program depths.

3.3 Data Race Detection

A vector clock [19] is used in the simulation model to represent the happened-before relations among the instructions performed by the threads. This information serves to infer data races in the simulation. The use of vector clocks is a well-known technique for this purpose. We only briefly revisit the algorithm here with certain technical details to support a more detailed understanding of our checker design.

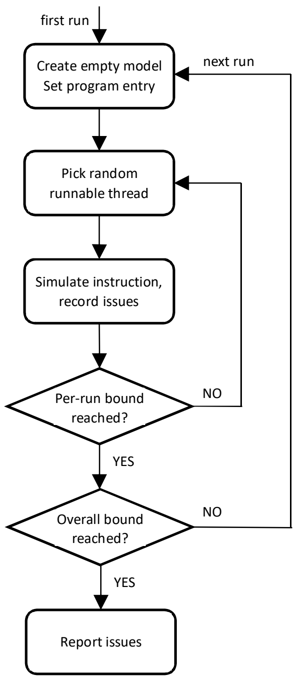


Figure 4: Flow chart of the checker algorithm.

Each thread in the model carries its vector time, defining its own current logical time and the points in time when it has been synchronized for the last time with other threads. Apart from the thread's current vector time, a thread also records all its performed accesses, each with the time and target, i.e. of whether it is a variable or array element. Moreover, synchronization objects, e.g. monitors, semaphores, read-write locks etc., carry the vector time of their last unlock.

Regarding the vector times, the interpreter now operates as follows: On thread starts, the started thread is synchronized with the time of the current thread and the current thread's time advances. On thread joins, the current thread's time advances and it is synchronized with the time of the terminated target thread. On unlock instructions, the current thread's time advances and is recorded in the synchronization object. On lock instructions, the current thread's time is synchronized with the recorded time of the synchronization object and the current thread's time advances. On accesses, the access is recorded together with the current time in an access event of the accessing thread.

Finally, data races can be identified with this information: After accessing a variable or array element, the access is compared against all other concurrent events of other threads, i.e. events that have no happened-before relation between each other regarding the vector times of the events. We only consider accesses with an exactly interpreted target, excluding accesses with "uninterpreted" addresses (cf. Section 3.5). If two compared events access the same exactly interpreted variable or array element, and at least one write is involved, it is recorded as a data race issue. The call stacks of the involved threads and transitive parent threads can be used to compose an error trace information for the issue.

3.4 Deadlock Detection

Deadlocks can be identified by analyzing the lock hold-and-wait dependencies among threads and looking for a cycle therein [14]. As this is also a standard approach, we only briefly explain its realization in our checker: Synchronization objects in our checker model carry the information about their lock owner (or multiple owners in the case of reader-writer locks), as well as about the threads awaiting the lock. When interpreting a lock instruction that blocks the actual thread, it is checked whether there has been a cyclic wait dependency established by transitively traversing over the awaited lock and the corresponding lock holders. If a cycle has been found, a deadlock issue is recorded. The involved threads are examined to remember an error trace for the issue.

3.5 Abstract States

The interpretation is mostly concrete, i.e. maintains exact states in the simulation model whenever possible. However, in certain cases, an exact state is not defined, unless invented values are introduced. Such cases are external input (user input, files, network etc.) or results of missing or incorrect program code that cannot be simulated in the sandbox. For such an undefined input, we assume an abstract state, called "uninterpreted value" that stands for any possible value of that type. At this point, the simulation is no longer fully exact, but takes potentially imprecise assumptions. If instructions involve uninterpreted operands, different actions are taken depending on the instruction type: For a branch instruction with an uninterpreted branch condition, a random branch is taken. If a lock, unlock, thread start or thread join refers to an uninterpreted object, the checker ignores the corresponding instruction. Moreover, a data race is exempt from the issue report, if it involves an access to an uninterpreted address. For all other instructions involving uninterpreted values as operands, uninterpreted values result from that instruction. The assumptions for uninterpreted branches, locks, and thread joins reduce precision and may lead to false positives due to infeasible paths or missing synchronization. Conversely, the handling of uninterpreted unlocks or thread starts, as well as the exclusion of data races with uninterpreted addresses, could result in overlooked issues, i.e. false negatives. Our approach is deliberately much simpler than in symbolic execution: Due to performance reasons, we avoid model checking and do not maintain potentially complex path constraints or symbolic expressions.

4 IMPLEMENTATION

We implemented the checker algorithm for the C# programming language. Based on the .NET Compiler Platform (also known as Roslyn), a plugin for the Visual Studio IDE has been developed that performs on-the-fly analysis of the source code and reports the detected concurrency issues and highlights them in the code. The .NET Compiler Platform already provides the abstract syntax trees and semantic symbol information of the C# source code. The framework also enables custom diagnosis plugins that directly run in the IDE. We implemented such a diagnosis class and derived the call graphs with intermediate language code from the available syntax trees and semantic information. Consequently, the diagnosis runs the static analysis with the randomized interpretation. As for the thresholds, the total number of logical steps was bound to 10,000,000 and the per-run limit was set to 1,000,000 logical steps. The heap size was limited to 8MB. If none of the program entries involves multi-threading in their first run, we skip any subsequent runs. The interpretation-based analysis eventually feeds back the identified data race and deadlock issues with corresponding error traces. The error displaying is again part of the inbuilt .NET Compiler Platform. Our diagnosis is a so-called full-solution analysis that considers the entire loaded code in the IDE. The checker implementation, excluding automated tests, comprises around 11,000 lines of code.

Specific aspects had to be addressed in the checker algorithm that are related to C# and the .NET framework, namely the selection of program entries and C#-specific concurrency features.

4.1 Program Entry Selection

Each interpretation round in the checker needs to start from a program entry. Two different strategies are used for the entry selection, an initial entry or an in-the-middle start.

An initial entry denotes a real starting point into the program. It depends on the type of the analyzed assembly: For a console application, this is the main method. The uninterpreted value is assigned to the parameter of the main method, see Section 3.5. For graphic user interfaces (UIs), we create and initialize the user interfaces and simulate a feasible random UI event sequence. For unit tests, the entries are the test methods after having instantiated and initialized the test classes. For libraries, we conservatively consider all publicly accessible members (methods, constructors, properties etc.) as entry point and select randomly. It would be possible to also test concurrent accesses of the library from outside but lacking a formal specification of the library's thread safety, we consider sequential library accesses to minimize false positives. Again, uninterpreted values are assigned to the parameters of library entry methods. In the case of a non-static method, the corresponding object is first instantiated.

Since the interpreter is bounded in the number of steps, we aim to increase the coverage by also selecting random in-the-middle starts. For this purpose, we collect methods that directly or indirectly start threads and consider them as potential in-the-middle starting points. The checker then randomly selects such starting points for the interpretation. The uninterpreted values are also helpful here to set up the parameters or related object states. While this strategy increases the potential of finding more issues, it also relaxes precision, and is thus a further reason why false positives are possible.

We select the program entry per assembly (.NET project) of the program (.NET solution) loaded in the IDE, although we perform a full-program (full-solution) analysis starting from this entry point.

4.2 Language-Specific Concurrency

This section discusses practical details of the checker with regard to C#. We support C# version 7 with most language concepts. Some limitations apply, as described in the next section. Specific concurrency features in .NET and C# thereby require special handling:

- Tasks of a thread pool, as offered by the .NET Task Parallel Library [20], can be mapped to simple threads in the interpretation. This is because parallel tasks can be scheduled on different worker threads but not necessarily. We conservatively consider the highest possible concurrency degree.
- Parallel loops and invokes are mapped to a multi-task start and join pattern. Task continuations are also imitated.
- The UI dispatcher loop is simulated by having a dedicated UI thread that sequentially schedules the UI events. Proper synchronization across the events by the submitting threads is realized.
- The C# async and await [2] programming model is mapped to equivalent task continuations by considering the two different scheduling options for the post-await continuation, i.e. with dispatcher and without dispatcher.
- Monitor synchronization with wait and pulsing, as well as all synchronization primitives, semaphores, reader-writer locks, barriers, count-down events, auto-reset-event etc. in the API are simulated with the corresponding vector time synchronizations.
- Accesses to volatile variables or dedicated volatile memory accesses are flagged in the corresponding access event of the threads. Data races are exempt if the two involved accesses are both volatile.
- Atomic accesses are also implemented and count as volatile in the access history and the data race detection. Explicit memory barriers (fences) advance the vector time of the effecting thread.
- The collection API, both non-thread-safe and the concurrent versions, are supported. Concurrent unsynchronized accesses to same non-thread-safe collection instance are also reported as an error by the checker.
- Garbage collection is also simulated to identify pending finalizers at regular intervals. Finalizers often are an overlooked source of concurrency errors and thus simulated by a finalizer thread in our checker.

4.3 Limitations

The current checker version has a few implementation restrictions that may reduce precision and coverage in specific areas:

- Only the C# source code of the active program (.NET solution) is analyzed, but not the referenced external libraries. Results from external libraries are treated as uninterpreted abstract states.
- A few language features are not yet exactly interpreted but simplified by using abstract states: language-integrated query (LINQ), parallel LINQ, yield statements, weak references, unsafe/unmanaged code, and certain standard API functionality.
- The program entry selection for web applications and web services with ASP.NET and WebApi are not yet specifically implemented but the assembly is only treated as a conventional library.

These limitations are not for conceptual reasons but a question of engineering time. We plan to extend the corresponding analysis in future versions.

5 EVALUATION

We take two approaches to assess the checker: Firstly, we apply it on larger open-source projects to see whether it is practically effective. Secondly, we compare it to other concurrency checkers with smaller tests.

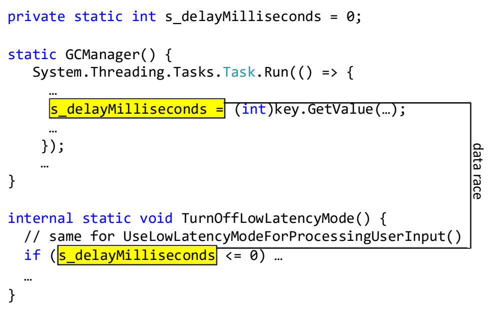
5.1 Practical Evaluation

As for the evaluation with realistic larger projects, we selected the top ten .NET C# projects from GitHub by the number of stars awarded in the user ranking. We excluded projects that do not include thread or task starts in productive code, i.e. code that may run at a user and is not unit-test or integration-test code. Moreover, we also skipped the .NET runtime implementations themselves, as they implement threading in their own classes and do not use it from an imported standard library, as it is expected in our checker. The selection hence comprised Roslyn, SignalR, Nancy, ILSpy, CefSharp, ReactiveUI, MsBuild, Hangfire, Polly, and NLog. We also searched for .NET projects with keywords on concurrency and parallelization and picked the GitHub projects ranked with at least 2000 stars, being Orleans, Akka.Net, and Rx.NET. With this approach, a benchmark of 13 .NET projects was assembled. For each project (.NET solution), we used the latest stable version (selected on 28 June 2017). Table 1 summarizes the different projects, providing the version, a brief description and code metrics. We counted the source lines of code (SLOC), the number of classes, the number of methods (including constructors, property accessors, lambdas, inner functions), as well as the code locations with explicit or implicit thread starts. The "thread starts" metric includes task launches in a thread pool, but excludes finalizers, task continuations, and await continuations. In total, the benchmark comprises a substantial code base of about 3.5 million SLOC, much more than has been reported in previous literature on concurrency checkers.

Our checker was applied to the benchmark of Table 1 inside Visual Studio 2017 on an Intel i7-4712MQ CPU with 2.3 GHz and 8GB main memory. Each GitHub project (a .NET solution) of the benchmark consisted of multiple C# assemblies (also called .NET projects) that were analyzed. Table 2 shows the results: We measured the total analysis time (including Roslyn compilation), and the corresponding average time per assembly in that .NET solution. The analysis proved to be fast, taking 1.71 seconds on average per assembly, and requiring 12.93 minutes in total for the entire code base. We also edited the code to see that the checker responded in similar short period as is desired for an interactive IDE code analysis. The tool reported 121 issues, all of them data races or thread-unsafe usage of API, that we eventually studied and classified. We performed manual code inspection for all reported issues. Many of the 121 detected issues stem from related patterns or have common root causes. We figured out that these issues had 13 different causes/patterns. Code review also revealed that 14 of the 121 issues are false positives, and only 107 are real issues, true positives, that can theoretically occur at runtime, although some may be unlikely. Out of these 107 issues, 89 reside in productive assemblies. The remainder belongs to sample or unit test code. We reported all issues in productive code that we identified as true positives. Deadlocks have not been found in the code base. However, to see that the deadlock detection principally works, we

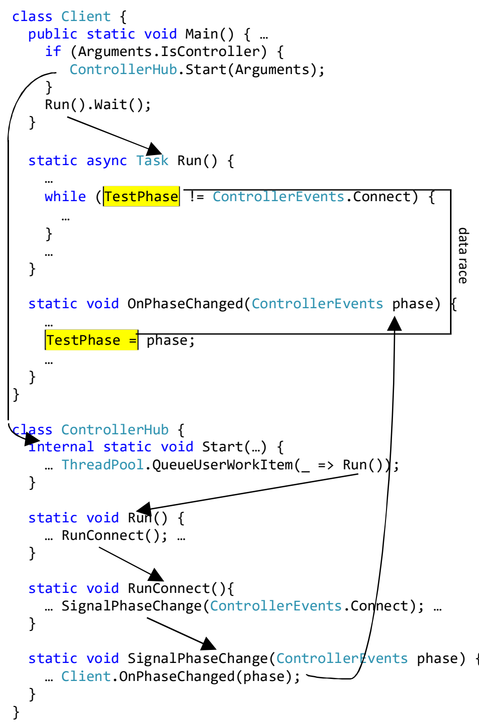
Table 1: Benchmark of popular open-source C# projects (.NET solutions) with code metrics, as used for the checker evaluation.

Project	Version	Description	SLOC	Classes	Methods	Thread starts
Roslyn	15.2	.NET compiler platform for C# and VB	1,851,645	9,880	128,608	258
SignalR	2.2.2	Real-time web functionality for .NET	86,574	1,057	11,316	198
Nancy	2.0.0-ce	Framework for HTTP services	72,345	1,106	10,881	3
ILSpy	2.4	.NET decompiler	279,432	2,434	28,565	36
CefSharp	57.0.0	Chromium Embedded Framework bindings	14,116	174	2,464	24
ReactiveUI	7.4.0	Reactive model-view-viewmodel framework	33,381	515	6,145	6
MsBuild	15.1.1012	Build engine for .NET and Visual Studio	397,281	1,866	26,244	33
Hangfire	1.6.14	Background job processing in .NET	73,986	1,048	9,345	3
Polly	5.2.0	Fault-handling library for .NET	91,363	586	18,440	231
NLog	4.4.11	Logging framework for .NET	63,381	645	6,477	129
Orleans	1.4.2	Distributed actor model	137,695	1,698	16,875	333
Akka.NET	1.2.2	Port of Akka actor model for .NET	225,744	4,555	36,589	314
Rx.NET	3.1.1	Reactive programming extension for .NET	155,358	984	21,053	297
Total			3,482,301	26,548	323,002	1,865



```
private static int s_delayMilliseconds = 0;

static GCManager() {
   System.Threading.Tasks.Task.Run(() => {
     …
     s_delayMilliseconds = (int)key.GetValue(…);
     …
    });
    …
}

internal static void TurnOffLowLatencyMode() {
  // same for UseLowLatencyModeForProcessingUserInput()
  if (s_delayMilliseconds <= 0) …
  …
}
```

Figure 5: Data race excerpt in Roslyn.



```
class Client {
  public static void Main() { …
    if (Arguments.IsController) {
       ControllerHub.Start(Arguments);
    }
    Run().Wait();
  }

  static async Task Run() {
    …
    while (TestPhase != ControllerEvents.Connect) {
      …
    }
    …
  }

  static void OnPhaseChanged(ControllerEvents phase) {
    …
    TestPhase = phase;
    …
  }
}

class ControllerHub {
  internal static void Start(…) {
    … ThreadPool.QueueUserWorkItem(_ => Run());
  }

  static void Run() {
    … RunConnect(); …
  }

  static void RunConnect(){
    … SignalPhaseChange(ControllerEvents.Connect); …
  }

  static void SignalPhaseChange(ControllerEvents phase) {
    … Client.OnPhaseChanged(phase);
  }
}
```

Figure 6: Data race excerpt in SignalR. `ThreadPool.QueueUserWorkItem` queues `Run()` as a thread pool task.

manually inserted deadlock-prone code at program entries (Roslyn `VBCSCompiler.Main` and NLog `InstallNLogConfig.Main`) and noticed that these were correctly identified.

We found the following data races in productive code: In Roslyn, the checker revealed a case in the language service implementation where a static constructor launches a task that writes a static field `s_delayMilliseconds`, which can be read by methods called `TurnOffLowLatencyMode` and `UseLowLatencyModeForProcessingUserInput`, see Figure 5. The risk of occurrence and impact may be considered relatively low but it is formal data race. In SignalR, we found several related races (data races and thread-unsafe calls) in a load test utility called Crank. Since Crank is used as a tool for testing web applications in the programmer community, we also consider it as productive code, not as sample or unit test code. Figure 6 sketches a data race in SignalR that leads to consequent races. NLog contains a data race pattern involving a timer, see Figure 7. Our tool detected similar repeated data races in Rx.NET, see Figure 8. In one project, Nancy, we also found a false negative, a data race that remained undetected by our tool. The reason is that it requires a specific series of external calls to the library that is not simulated by our tool. We also reported this issue.

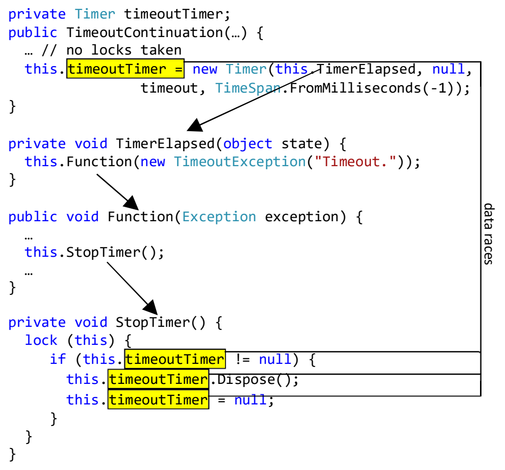
We identified the following reasons for false positives: (1) For SignalR, the checker missed a happened-before relation for a specific continuation API call, an issue that we corrected after the evaluation. (2) For CefSharp, specific ordering of a seldom UI initialization event (`HandleCreated` before window initialization) is not implemented. (3) For MsBuild, the `WaitHandle.WaitAny` API call was not yet specifically supported in the interpretation. (4) For NLog, the timer period was set to an infinite value, meaning it is a single occurrence that is not yet handled by the checker. These false positives could be avoided by introducing a broader and more precise support of the .NET API, which is eventually a question of effort. The false positive rate is relatively low with 12%, nevertheless.

None of the observed false positives is related to the imprecise interpretation with abstract states. We see three reasons for this: Firstly, uninterpreted values only stem from a few external sources (e.g. user inputs) since we analyze the entire project source code with a limited number of external libraries that cannot be analyzed. Secondly, none of the detected races depends on execution traces

Table 2: Evaluation results of the checker with the total and per-assembly analysis times and the detected issues, their underlying causes, the grouping into false and true positives, as well as the fraction of issues in productive code.

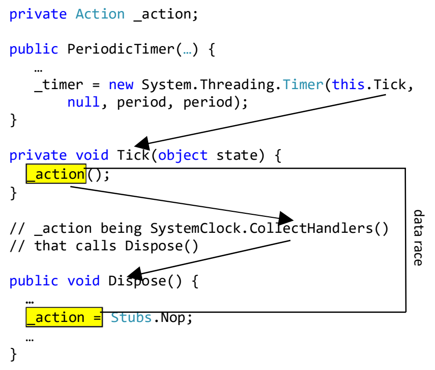
Project	Assemblies	Time [min]	Time/assem [s]	Issues	Causes	False pos.	Real issues	Productive
Roslyn	114	7.33	3.86	3	2	0	3	2
SignalR	31	0.43	0.84	80	2	1	79	79
Nancy	56	0.56	0.60	0	0	0	0	0 (1*)
ILSpy	14	1.01	4.33	0	0	0	0	0
CefSharp	9	0.17	1.14	1	1	1	0	0
ReactiveUI	10	0.17	1.04	0	0	0	0	0
MsBuild	20	0.64	1.91	4	2	4	0	0
Hangfire	12	0.20	0.98	0	0	0	0	0
Polly	6	0.28	2.85	0	0	0	0	0
NLog	6	0.14	1.37	13	4	8	5	3
Orleans	29	0.37	0.77	0	0	0	0	0
Akka.NET	82	1.29	0.94	0	0	0	0	0
Rx.NET	13	0.34	1.58	20	2	0	20	5
Total	402	12.93	avg. 1.71	121	13	14	107	89

(*) 1 false negative issue found during code review



```
private Timer timeoutTimer;
public TimeoutContinuation(…) {
  … // no locks taken
  this.timeoutTimer = new Timer(this.TimerElapsed, null,
              timeout, TimeSpan.FromMilliseconds(-1));
}

private void TimerElapsed(object state) {
  this.Function(new TimeoutException("Timeout."));
}

public void Function(Exception exception) {
  …
  this.StopTimer();
  …
}

private void StopTimer() {
  lock (this) {
     if (this.timeoutTimer != null) {
       this.timeoutTimer.Dispose();
       this.timeoutTimer = null;
     }
  }
}
```

Figure 7: Data race excerpt in NLog. Although the data race may seem to be unlikely with long timeouts, it is theoretically possible.



```
private Action _action;

public PeriodicTimer(…) {
  …
  _timer = new System.Threading.Timer(this.Tick,
      null, period, period);
}

private void Tick(object state) {
  _action();
}

// _action being SystemClock.CollectHandlers()
// that calls Dispose()

public void Dispose() {
  …
  _action = Stubs.Nop;
  …
}
```

Figure 8: Data race excerpt in Rx.NET. The timer `Dispose()` could run while a subsequent tick is active.

with mutually exclusive branch conditions inferred from uninterpreted values. The identified data races thus seem to follow simpler patterns regarding branches. Thirdly, locks do typically not target uninterpreted objects, as it is a C# design practice to lock on aggregated helper instances inside classes, such that the helper is concretely interpretable.

5.2 Tool Comparison

We also aim to compare our tool to other concurrency checkers. As mentioned, there exist only a few maintained and publicly available concurrency checkers, and unfortunately, none for C#. The predestined candidate tool CHESS [24] (discontinued in 2009) lacks 64-bit platform support and could not be applied. We therefore performed an indirect evaluation by assembling a benchmark of smaller test cases that we ported from C# to other programming languages, namely Java and C++. For the comparison, we chose Java Pathfinder (JPF) [26] as a well-known static checker for Java programs, and ThreadSanitizer [34] as a representative of a dynamic checker working on C/C++ code. We built the JPF binaries from the source repository using the latest checkin of Feb. 2017 and used ThreadSanitizer as part of the Clang 3.3 compiler. We took the same machine as in the previous section.

Our benchmark comprises the following characteristic test cases, each being implemented in C#, Java, and C++: (1) `DCL Broken`: The defect double checked locking pattern shown in Figure 1. (2) `DCL Fixed`: The correct implementation of double checked locking, using volatile in C# and Java, and applying atomic accesses in C++ (as volatile in C does not define a fence). (3) `Quicksort Broken`: The defect quicksort implementation explained in Figure 2. In JPF, we had to use threads instead of ForkJoinPool tasks due to an internal JPF exception. (4) `Quicksort Fixed`: The correct quicksort implementation with the fix indicated in Figure 2. (5) `Bank Broken`: The deadlock scenario with the bank transfer of Figure 3. (6) `Bank Fixed`: The correct bank scenario with acyclic transfers. (7) `Buffer Broken`: A producer-consumer scenario with an unsynchronized bounded buffer, concurrently accessing a non-thread-safe API collection, therefore leading to race conditions. (8) `Buffer Fixed`: The producer-consumer scenario with a properly synchronized bounded buffer, implemented as a monitor. (9) `Time`

Race: An infinite periodic timer with tick events that provoke data races. (10) Finalizer Race: A data race scenario between a GC finalizer and the mutator. This case is not applicable for C++ since C++ has no GC. We had to avoid user inputs in the mentioned test cases in order not to block JPF (concrete interpretation) and ThreadSanitizer (dynamic analysis).

Table 3 summarizes the test cases with their actual issues. It also denotes the runtimes and detected results for the three compared checkers: Our checker (C#), Java Pathfinder (Java), and ThreadSanitizer (C++). The runtimes are the average of three subsequent measurements. They are indicated in seconds and rounded to one significant figure, where short runtimes below 100 milliseconds are not further resolved.

As can be seen, most of the cases are accurately analyzed, i.e. at least one of the corresponding issues is reported for a broken test cases and no problem is indicated in a correct version. There are only a few exceptions: JPF has a false positive on DCL Fixed as it does not exempt volatile accesses from data races. Thread-unsafe collection accesses remain undetected in JPF, explaining the false negative in the Buffer Broken case. The false negative on Finalizer Race occurs because JPF does not examine GC finalizers. ThreadSanitizer does not feature a deadlock detection, and therefore misses the corresponding issue in the Bank Broken test.

Generally, short analysis times of a few seconds were measured, although with a few exceptions: Certain cases ran for a very long time in JPF due to an exponentially grown search space, and we manually stopped them after an hour. This shows how a static analysis like JPF can lead to impractically long runtimes, even for the relatively small programs of this comparison. As for ThreadSanitizer, certain analyses did not terminate because the tested program were endless themselves (Quicksort Broken due the bug explained in Figure 2, and Timer Race due to its infinite periodic ticker).

Our checker exhibits short analysis times of usually less than one second. Only the quicksort case required more time than in the other checkers. A reason for this is that our checker does not stop on the first data race encounter but continues its search. More specifically, it reports 10 data races for the Quicksort Broken case, while JPF reports one data race, and ThreadSanitizer three data races.

6 RELATED WORK

As concurrency analysis is a heavily studied research field, we can only discuss a selected set of prior work. The discussion is categorized into static and dynamic analysis.

6.1 Static Analysis

Bug style checkers are predominant in industrial application, e.g. checkers, e.g. FindBugs for Java [15] or Code Analysis for .NET [23]. As these checkers only statically identify specific code anti-patterns, including concurrency-related bugs, they do not perform an overall analysis, and are unable to detect data races or deadlocks in general. Their advantage and probably also the reason for the prevalence is, however, the relatively fast execution time of a few seconds, even for MLOC programs. The work of Santhiar and Kanade [31] detects a specific concurrent thread blocking pattern in the async-await programming model of C#.

The lockset algorithm is a broadly known static data race detection method. It has been introduced by Warlock [35], and applied in RacerX [9] and others [18, 25, 39, 41]. The idea is to gather the taken locks per memory access and verify that all accesses to the same location carry at least a common lock. It is a flow-sensitive analysis, except for Warlock. RacerX additionally performs a deadlock analysis by also computing the lock orders. The limitation of the basic lockset algorithm and its checkers [9, 18, 35, 39] is that it ignores that only specific code regions run effectively concurrent to each other. This makes the approach prone to false positives. For this purpose, code annotations [9, 35] or heuristic discrimination of code into concurrent or non-concurrent [9] is used. Chord [25] improves the precision by implementing extra analyses, such as of aliasing and thread starts. The static analysis of von Praun and Gross [41] additionally considers thread joins. May-Happen-in-Parallel (MHP) relations [27] give a more precise definition of which instructions may run concurrently. Static vector clocks [45] thereby offer a very efficient approach to compute these relations, achieving higher precision than other static checkers. All these mechanisms are sound, though some with specific exemptions, which is the major advantage over our checker. They, however, inherently have less precision than our approach as we only simulate a series of mostly concrete program traces.

Precision in data race analysis can also be increased by constraint-based analysis, such as in Locksmith [29], or model checking [4, 24, 26, 36]. Such analysis, however, inherently suffer from the exponential state explosion problem, making it often inapplicable to larger programs. Despite numerous techniques against state explosion [3], the search space of practical multiple KLOC programs is still too large to be analyzable in only a few seconds. Java Pathfinder (JPF) [26] performs a concrete interpretation similar to our checker but with an exhaustive analysis, thus also having exponential runtimes. The same applies to CHESS [24] that systematically explores all concurrent schedules, although with a stateless replay technique. A concrete interpretation, however, requires a program without external IO inputs to be automatically testable. Abstract interpretation [5] can reduce the state space but the analysis time still grows exponentially. This has also been applied to JPF [26]. Symbolic execution [1] maintains as many concrete states as possible but can deal with unknown external inputs, by using symbolic expressions and deriving path constraints that are eventually analyzed by a model checker. Moreover, Polyspace [36] is an industrial static checker that detects data races and deadlocks based on abstract interpretation. Some static checkers trade soundness against higher precision and higher performance: Bounded model checking [3] limits the analysis depth but still leads to exponential state spaces. Prior art on random interpretation [6, 13] does not tackle data race or deadlock analysis. Our checker uses a mostly concrete interpretation, with a certain extent of abstraction, in combination with randomized thread scheduling.

Except for the bug style checkers, JPF [26], and the Polyspace checker [36], all the mentioned static tools are either publicly unavailable or no longer maintained. The CHESS [24] project has been discontinued after 2009 and newer .NET programs can no longer be analyzed.

Table 3: Comparison of our checker (C#) to Java Pathfinder and ThreadSanitizer (C/C++).

Test Case		Our Checker		Java Pathfinder		ThreadSanitizer	
Name	Issues	Time [s]	Result	Time [s]	Result	Time [s]	Result
DCL Broken	data race	< 0.1	accurate	0.8	accurate	< 0.1	accurate
DCL Fixed	none	< 0.1	accurate	0.7	*false pos.*	< 0.1	accurate
Quicksort Broken	data races	4	accurate	0.9	accurate	(timeout[1])	accurate
Quicksort Fixed	none	2	accurate	(timeout[2])	accurate	< 0.1	accurate
Bank Broken	deadlock	< 0.1	accurate	1	accurate	< 0.1	*false neg.*[3]
Bank Fixed	none	< 0.1	accurate	2	accurate	< 0.1	accurate
Buffer Broken	race condition	< 0.1	accurate	(timeout[2])	*false neg.*	< 0.1	accurate
Buffer Fixed	none	0.8	accurate	2	accurate	< 0.1	accurate
Timer Race	data race	0.1	accurate	2	accurate	(timeout[1])	accurate
Finalizer Race	data race	< 0.1	accurate	0.5	*false neg.*	n/a	n/a

([1]) tested program is endless by design, ([2]) analysis takes more than 1 hour, ([3]) checker does not analyze deadlocks

6.2 Dynamic Analysis

Dynamic checkers have certain prevalence in practice because of their precision. They commonly instrument programs to monitor the program execution and thereby detect runtime errors that effectively occurred. Some examples of practically used and maintained dynamic concurrency checker tools are Intel Inspector [17], Thread Sanitizer [34], or Helgrind [38]. They typically analyze happened-before relations [19] which is the most precise algorithm.

Earlier dynamic checkers used weaker algorithms than happened-before analysis: The earliest known dynamic checker of data races [22] focused on start/join dependencies instead of general synchronizations. Later, Eraser [32] applied the lockset algorithm. A different approach is taken by von Praun and Gross [40] by tracking object ownership at runtime. O'Callahan and Choi [28] use a hybrid of happened-before and lockset analysis.

LiteRace [21] speeds up the happened-before-based data race analysis using sampling to a certain extent instead of fully instrumented code. FastTrack [11] and AccuLock [42] avoid costs of unnecessarily fine-grained vector clocks by adaptive representations of the happened-before relations. RaceTrack [44] applies adaptive granularities in the race detection, from the object level down to field level, to make the dynamic checker faster. DataCollider [10] introduces a light-weight random sampling techniques based on breakpoints that performs very effectively for low-level kernel code.

Predictive testing [16, 30, 43] combines static analysis with a dynamic checker: Generally, from the postmortem dynamic trace, alternative feasible schedules are statically derived to also detect issues in those traces by using a constraint solver. A similar hybrid approach is taken by concolic testing [12]: Programs are tested by concrete execution using initial random inputs, while simultaneously performing a symbolic execution to derive path constraints. The constraints eventually serve to derive new input cases for subsequent execution. The difficulty often is the computational complexity of a constraint solver.

ConTest [7] is related to our checker insofar that it also introduces random thread interleaving by introducing sleeps for data race and deadlock detection. Contrary to this, we never effectively delay the threads by sleeps in our checker, but directly simulate the scheduler. Rapos [33] proposes a more sophisticated random scheduler that experimentally performs more uniformly with regard to non-equivalent thread interleaving selects a sub-set of mutually independent runnable threads. We choose the simple scheduler, as we can then avoid the mutual thread independency analysis in each scheduler round. An empirical study [37] also shows that naive random scheduling performs relatively effectively in comparison to more sophisticated approaches. Delay-bound scheduling [8] adds a fix of variations in an otherwise-deterministic scheduler to make it non-deterministic.

As is the case with all dynamic checkers, our checker aims at precision but is unsound. However, as our checker is static and not dynamic, it does not require a running program and can deal with incomplete program parts and external inputs.

From a practical perspective, the following discussed tools are still publicly available and maintained according to our study: Intel Inspector [17], ThreadSanitizer [34], and Helgrind [38].

7 CONCLUSION

Concurrency errors should be eliminated as early as possible, ideally already at coding time in the integrated development environment. For this use case, a static analysis with short duration and high precision is considered important, even if does not necessarily detect all issues. We took the approach of mostly concrete interpretation with random scheduling and bounded exploration for this purpose. This mechanism has been implemented for C#, by covering the broad concurrency spectrum of the language, including task-based and asynchronous programming, the various synchronization primitives, as well as, atomic/volatile accesses and GC finalizers. Offered as a plugin for Visual Studio, it analyzes the source code of the IDE in the background and directly marks issues in the code. The experimental evaluation of popular open-source C# projects shows that the checker can uncover a good amount of concurrency issues. The analysis thereby takes a few seconds per C# assembly and only exhibits few false positives.

AVAILABILITY

The HSR Parallel Checker is available for free in the Visual Studio Marketplace. More information can be found on the project website: http://parallel-checker.com.

ACKNOWLEDGMENT

I thank Christoph Amrein for certain code extensions, such as the configuration dialog and several bug-fixes, as well as, for the review of experimental results.

REFERENCES

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2016. A Survey of Symbolic Execution Techniques. *CoRR* (2016).

[2] Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause 'n' Play: Formalizing Asynchronous C#, In ECOOP 2012 – Object-Oriented Programming. 7313 2012, 233–257.

[3] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design* 19, 1 (01 Jul 2001), 7–34.

[4] Edmund M. Clarke and E. Allen Emerson. 1982. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *In: Kozen, D., Ed., Logics of Programs*. Springer, 52–71.

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. 238–252.

[6] Patrick Cousot and Michael Monerau. 2012. Probabilistic Abstract Interpretation. In *Proceedings of the 21st European Conference on Programming Languages and Systems (ESOP'12)*. 169–193.

[7] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. 2003. Framework for Testing Multi-Threaded Java Programs. *Concurrency and Computation: Practice and Experience* 15, 3-5 (2003), 485–499.

[8] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2010. *Delay-Bounded Scheduling*. Technical Report. https://www.microsoft.com/en-us/research/publication/delay-bounded-scheduling/

[9] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. 237–252.

[10] John Erickson, Madan Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *Operating System Design and Implementation (OSDI'10)*. USENIX.

[11] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *In Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*.

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. 213–223.

[13] Sumit Gulwani and George C. Necula. 2005. Precise Interprocedural Analysis Using Random Interpretation. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. 324–337.

[14] Richard C. Holt. 1972. Some Deadlock Properties of Computer Systems. *ACM Comput. Surv.* 4, 3 (Sept. 1972), 179–196.

[15] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *ACM SIGPLAN Not.* 39, 12 (Dec. 2004), 92–106.

[16] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. 337–348.

[17] Intel Corp. 2017. Inspector Xe. (2017). https://software.intel.com/intel-inspector-xet

[18] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. *Fast and Accurate Static Data-Race Detection for Concurrent Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 226–239.

[19] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

[20] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. 227–242.

[21] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. 134–143.

[22] John Mellor-Crummey. 1991. On-the-fly Detection of Data Races for Programs with Nested Fork-join Parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. 24–33.

[23] Microsoft Corp. 2017. Code Analysis for Managed Code Overview. (2017). https://docs.microsoft.com/en-us/visualstudio/code-quality/code-analysis-for-managed-code-overview

[24] Madan Musuvathi, Shaz Qadeer, and Tom Ball. 2007. *CHESS: A Systematic Testing Tool for Concurrent Software*. Technical Report. Microsoft Research. MSR-TR-2007-149.

[25] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. 308–319.

[26] NASA Ames Research Center. 2007. Java Pathfinder (JPF). (2007). http://babelfish.arc.nasa.gov/trac/jpf

[27] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. 1999. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. *SIGSOFT Softw. Eng. Notes* 24, 6 (Oct. 1999), 338–354.

[28] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*. 167–178.

[29] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. In *In PLDI âĂŽ06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, 320–331.

[30] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. *Generating Data Race Witnesses by an SMT-Based Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327.

[31] Anirudh Santhiar and Aditya Kanade. 2017. Static Deadlock Detection for Asynchronous C# Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. 292–305.

[32] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. 27–37.

[33] Koushik Sen. 2007. Effective Random Testing of Concurrent Programs. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 323–332.

[34] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*. 62–71.

[35] Nicholas Sterling. 1993. Warlock: A Static Data Race Analysis Tool. In *Winter USENIX*. San Diego, California, 97–106.

[36] The MathWorks Inc. 2017. Polyspace Static Analysis – Concurrency: Race Conditions and Deadlocks. (2017). https://www.mathworks.com/products/polyspace/concurrency-race-conditions-deadlocks.html

[37] Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2014. Concurrency Testing Using Schedule Bounding: an Empirical Study. In *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. 15–28.

[38] Valgrind Project. 2005. Helgrind: A Data-Race Detector. (2005). http://valgrind.org/docs/manual/hg-manual.html

[39] Vesal Vojdani and Varmo Vene. 2009. Goblint: Path-Sensitive Data Race Analysis. *Annales Univ. Sci. Budapest., Sect. Comp* (2009).

[40] Christoph von Praun and Thomas R. Gross. 2001. Object Race Detection. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. 70–82.

[41] Christoph von Praun and Thomas R. Gross. 2003. Static Conflict Analysis for Multi-threaded Object-oriented Programs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. 115–128.

[42] Xinwei Xie, Jingling Xue, and Jie Zhang. 2013. Acculock: Accurate and Efficient Detection of Data Races. *Software: Practice and Experience* 43, 5 (2013), 543–576.

[43] Adarsh Yoga, Santosh Nagarakatte, and Aarti Gupta. 2016. Parallel Data Race Detection for Task Parallel Programs with Locks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. 833–845.

[44] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking, In ACM Symposium on Operating Systems Principles (SOSP 2005). 33.

[45] Qing Zhou, Lian Li, Lei Wang, Jingling Xue, and Xiaobing Feng. 2018. May-happen-in-parallel Analysis with Static Vector Clocks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 228–240.