HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

INSTITUTE FOR
SOFTWARE

Bachelor Thesis, Department of Computer Science

Extended Static Race Detection for Visual Studio

University of Applied Sciences Rapperswil (HSR)

Springsemester 2015
12. June 2015

Author:	Thomas Charrière
Supervisor:	Prof. Dr. Luc Bläser, HSR
Reviser:	Prof. Dr. Farhad Mehta, HSR
Expert:	Dr. Felix Friedrich, ETH Zürich
Project partner:	Institute for Software (IFS)
Timeperiod:	16.02.2015 - 12.06.2015
Workload:	360 Hours, 12 ECTS

Contents

Chapter 1

Abstract

Multi-threaded programs are known to be prone to data races that are hard to find due to their non-deterministic occurrence. This thesis implements a novel static analysis mechanism for detecting low-level data races in the C# programming language. The tool works on the fly within the Visual Studio integrated development environment using the Roslyn framework. Based on a novel algorithm, the analysis employs conservative inter-procedural data-flow analysis considering thread dependency graphs with start/join relations. The result is working prototype with a simple architecture that efficiently detects potential data race issues by conveniently highlighting them in the program source code.

Chapter 2

Management Summary

2.1 Introduction

Multi-threaded programs are known to be prone to concurrency errors, in particular to data races. These issues are hard to find due to a non-deterministic occurrence. Static analysis offers an approach for the systematic detection of such errors, conservatively finding all issues with possible false positives. Most effectiveness can be achieved through an on-the-fly detection, working directly within the programmer's integrated development environment. Unfortunately, no such tool currently exists for prominent programming languages such as C#.

2.2 Approach / Technologies

This thesis implements a static detection of low-level data races for the C# programming language. This is realised as a Visual Studio plugin based on Roslyn, the new .NET compiler framework. The tool is designed into abstraction layers: a generic inter-procedural data-flow analysis and a data race checker on top. The data race detection employs a novel algorithm considering start/join relations among threads.

2.3 Result

The result proves to be an efficient static checker that detects potential data races in C# program code within Visual Studio, highlighting and marking such issues during code writing. In contrast to other existing solutions, this checker performs a real static analysis (not only simple local bug pattern location) as a practical Visual Studio plugin. Due to the generic architecture, the tool could be extended to also support deadlock detection in the future.

Chapter 3

Technical Report

3.1 Introduction

3.1.1 Problem

In today's programming languages, such as C#, multiple threads potentially interact (explicitly or implicitly) via shared resources, being variable state within the object heap. In contrast to purely sequential programs, multi-threaded programs are prone to new archetypes of issues, so called concurrency errors. As these errors occur non-deterministically, they are inherently hard to find during execution and testing. These problems may take a long time to completely resolve and often depend on a sequence of low-probability events, this increases the risk that errors will elude tests yet make grand entrances when software is released to thousand of users. After fixing a defect, it is difficult to ensure that the defect was truly rectified and not simply masked.

Concurrency errors can be classified into three categories: **race conditions**, **deadlocks** and **starvations**.

race condition

Data races, or its synonym race conditions, are erroneous program behaviour due to insufficiently synchronised concurrent accesses on shared resources.

deadlock

Deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does. This usually occurs due to false mutual exclusion (lock) ordering.

starvation

Thread starvation is a situation in which a thread is perpetually denied necessary resources to proceed its work.

Low-level data races are characterised with regard to the memory model: They become manifest in concurrent unsynchronised accesses on the same memory location, involving at least one write

access. High-level data races may occur even despite the absence of low-level data races if the critical sections are not synchronised as a whole but only governed by multiple smaller atomic regions, i.e. the blocks of mutual exclusion are not sufficiently large.

Low-level data races can be formally determined, while high-level data races depend on the program semantics. Hence, only low-level data races can be detected by machine tool, while high-level data races can, in practice, only be determined by the programmer.

The .NET Framework offers a broad set of concurrency programming concepts, ranging from explicit multi-threading with Threads and Task Schedulers or implicit multi-threading with libraries such as Task Parallel Library (TPL) and Parallel Language-Integrated Query (PLINQ).[3] Each concurrency concept is susceptible to the aforementioned errors, as all concepts are based on multi-threaded execution. Due to the vast number of concurrency possibilities this thesis concentrates on low-level data races in the C# programming language.

Verifying a concurrent application involves testing for correctness, performance, reliability and scalability. The correctness of the application is determined using techniques such as **dynamic analysis**, **static analysis**, and **model checking**.

dynamic analysis

In dynamic analysis, bugs are detected by looking at the execution behaviour, i.e monitoring the effective execution. Online dynamic analysis inspects a program while it is executing and offline dynamic analysis record traces and analyses them at a later time to detect bugs. The analysis can only inspect taken execution paths, this lead to an incomplete analysis and false negatives. Most dynamic analysis tools rely on a layer between the program and operating system (i.e. custom virtual machine or code instrumentation, the latter being the most common) that could modify the behaviour at run-time or lead to performance problems, which is another drawback.

static analysis

Static analysis, used in this thesis, inspects code without actually executing the program, i.e. at compile time. Static analysis can be complete (instead of merely thorough/detailed), finding all potential issues but also false positives. Static analysis is also often very slow because of the exponential state explosion problem.

model checking

Model checking is special case of static analysis and suffers from this exponential state explosion problem. Model checking explores all states that can occur with concurrent execution and checks them for concurrency correctness criteria. The extraction of the model - a simplified formal language faraway from a practically usable programming language - is complicated in practice. Model checking may prove that the design is error free, but the implementation may still be incorrect. In practice, model checking is useful only for small, critical sections of a product. [10]

3.1.2 Example

Listing 3.1: A very simple program that writes and reads a boolean (true/false) value concurrently in several threads but in the process causes data races.

```
1   private static bool flag;
2   private static object myLock = new object();
3
4   public static void Main()
5   {
6       var t1 = new Thread(() =>
7       {
8           lock (myLock)
9           {
10              flag = true;
11          }
12      });
13      var t2 = new Thread(() =>
14      {
15          lock (myLock)
16          {
17              var x = flag;
18          }
19          var t3 = new Thread(() => flag = false);
20          t3.Start();
21      });
22      t1.Start();
23      t2.Start();
24
25      Console.WriteLine(flag); // output could be true/false
26
27      t1.Join();
28      t2.Join();
29
30      Console.WriteLine(!flag); // output could be true/false
31  }
```

The programmer has made an effort towards synchronised access (mutual exclusion) on *flag*:

- Line 10,17: Read/Write access to *flag* is protected through the lock statements (common monitor lock).
- (Line 30: *t1* and *t2* have been joined. Therefore, *flag* should only be accessed by the Main thread)

There are, however, several problems that have not been taken into account:

- Line 25: Output maybe true or false (read access *flag*) as *t1*/*t3* (write access *flag*) is running concurrently to the main thread and the mutual-exclusion lock is only supplied in *t1* and *t2*.

- Line 30: *t1*, *t2* have been joined, *t3* is however still potentially running. Output (read access *flag*) could be either true or false.

3.1.3 Related Work

Existing Prototype

The existing concurrency checker prototype developed by the Institute for Software, HSR Hochschule für Technik (IFS) engaged a novel conservative static analysis algorithm developed by Prof. Dr. Luc Bläser. It considers inter-procedural thread dependency graphs with start/join relations and supports various explicit and implicit thread entrance points, including the TPL. However, the implementation of the prototype is currently quite limited as it is only based on a flow-insensitive analysis.

Listing 3.2 would be considered race-condition-free by the prototype. The analysis is carried out in a flow-insensitive manner - the syntax tree is analysed in a linear fashion, not considering program paths or flows leading to inaccurate results due to both incomplete as well as inadequate analysis.

Listing 3.2: Example of recursive thread creation that requires data-flow analysis to detect data races.

```
1   static bool terminated = false;
2
3   static void Main(string[] args)
4   {
5       int i = 5;
6       Thread t1 = new Thread(() => { });
7       while (--i > 0)
8       {
9           t1.Start();
10          t1 = new Thread(() => terminated = !terminated);
11      }
12      t1.Join();
13      Console.WriteLine(terminated);
14  }
```

As *t1* is started and the reference replaced by a new *Thread* four times, only the fourth (and last) created thread is joined with the Main thread, there are therefore potentially three other threads running with data-accesses on *terminated*.

Other Related Work

There are a number of concurrency testing tools, mostly based on either dynamic or static analysis. Each tool researched concentrates on a particular concurrency error detecting area or programming language.

Static analysis

RacerX

RacerX is a static analysis tool that uses flow-sensitive, inter-procedural analysis to detect both race conditions and deadlocks. [1] It is explicitly designed to find errors in large, complex multi-threaded systems. It is designed for the C programming language and can be re-targeted to system-specific locking functions. RacerX does not accurately determine concurrent program code pieces. It heuristically classifies code in sequential and concurrent, by also neglecting thread start/join dependencies.

Chord

Checker of races and deadlocks (Chord) is a static flow-insensitive framework for Java. Chord is intended to work on a variety of platforms, including Linux, Windows/Cygwin, and MacOS. Chord's race detection algorithm is context-sensitive but flow-insensitive. Its lack of flow-sensitivity helps scalability but undermines the precision of the analysis causing many false positives. It also does not consider start/join relations. [9]

Dynamic analysis

CHESS

CHESS created by Microsoft Research detects concurrency errors by systematically exploring thread schedules and interleaving. It is a dynamic analysis tool, i.e. samples only selected program paths and monitors their execution for correctness. It may inherently miss errors unless all paths and interleavings are explored. The latter, however, takes (super-)exponential time and is therefore practically impossible for larger programs. It was designed for the C/C++ programming languages and the Windows operating system. [6]

Eraser

Eraser is a dynamic analysis tool that detects data races by dynamically tracking the set of locks held during program execution. [11] Eraser then uses these lock-sets to compute the intersection of all locks held when accessing shared state. Shared locations that have an empty intersection are flagged as not being consistently protected. As a dynamic analysis tool, it only inspects taken execution paths thus potentially missing issues, it therefore does not provide a complete analysis. It was designed in 1997 for the UNIX operating system.

On-the-fly Object Race detection

Object Race detection by Christoph von Praun, ETHZ and Thomas R. Gross, ETHZ have developed an on-the-fly mechanism that detects access conflicts in executions of multi-threaded Java programs and has been implemented in collaboration with an "ahead-of-time" Java compiler. It uses dynamic analysis to detect access conflicts but it only inspects taken execution paths thus potentially missing issues, it therefore does not provide a complete analysis. [12]

On-the-fly Detection of Data Races

John Mellor-Crummey describes a dynamic on-the-fly techniques involving augmenting a program to detect and report data races as they occur during its execution. These techniques maintain additional information at run-time to determine when conflicting accesses to a shared variable have occurred. It reports only feasible races. A prototype was developed in parallel FORTRAN with limited support for simple programs only supporting intra-procedural scoping. [8]

3.1.4 Goals

Flow-Sensitivity

Redesign the algorithm to become flow-sensitive and thus become complete and more precise than the existing IFS prototype.

Simple Design

Introduce a simple abstraction layer based on generic data-flow analysis to simplify concurrency checker and abandon unreadable visitor pattern code

Roslyn

Integrating the newest version of the .NET compiler platform ("Roslyn") to be compatible with the newest version of Visual Studio (VS2015).

Visual Studio Integration Package

Create a Visual Studio Integration Package (VSIX) according to the newest best practises to enable deployment of the concurrency checker to other installations.

3.2 Data-Flow Analysis

3.2.1 Control Flow Graph (CFG)

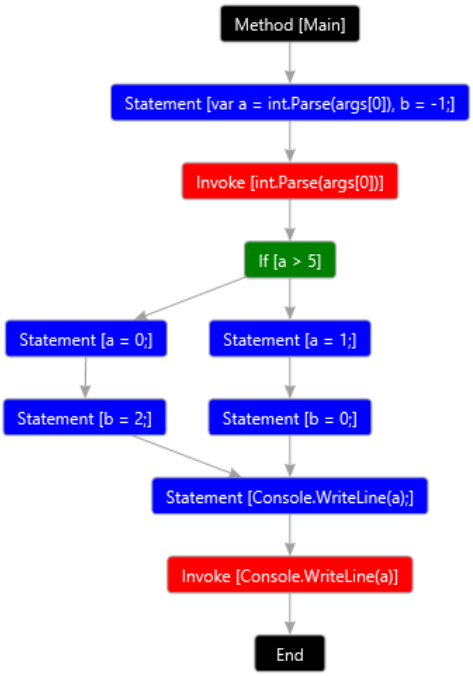
A control flow graph (CFG) is a directed graph that models all paths that might be traversed through a program during its execution [13], and is created for every method, property, constructor and lambda expression. Each node in the graph represents a statement of code, directed edges are used to represent jumps in the control flow. Statements are explicitly not condensed into a basic block, i.e. a straight-line piece of code without any jumps or jump targets, in the CFG as it allows for a more fine-grained analysis, i.e. access/concurrency errors per statement.

There are two specially designated nodes: the entry node, through which control enters and the exit node, through which all control flow ends. Calls to other procedures (such as *int.Parse(args[0])* in fig. 3.1) are always represented in their own nodes.

Listing 3.3: Excerpt of a simple program with an if statement.

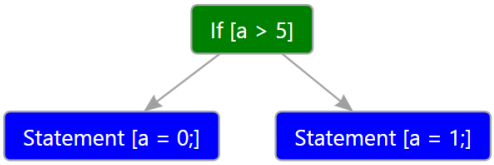
```
1  public static void Main(string[] ↩
       args)
2  {
3    var a = int.Parse(args[0]), b = ↩
         -1;
4    if (a > 5)
5    {
6      a = 0;
7      b = 2;
8    }
9    else
10   {
11     a = 1;
12     b = 0;
13   }
14   Console.WriteLine(a);
15 }
```

Figure 3.1: Resulting CFG of listing 3.3.



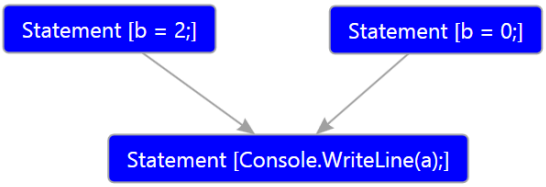
If a statement has more than one outgoing edge, this indicates that the program will, at any one point, take only one of the execution paths. This will generally occur when a condition must be fulfilled either through an if, while or switch statement.

Figure 3.2: An excerpt of fig. 3.1 - multiple outgoing edges from an if statement indicating two possible execution paths.



Multiple incoming edges on the other hand indicate two or more execution paths merging back to the same execution path.

Figure 3.3: An excerpt of fig. 3.1 - multiple incoming edges merging to the same execution path.



3.2.2 Intra-Procedural Data-Flow Analysis

Intra-procedural data-flow analysis focuses on gathering information about possible states per point within a single procedure (method, property or lambda expression), characterised as a node in the CFG. The generated CFG allows the analysis to take the order of statements (flow-sensitive) in a procedure into account.

To perform data-flow analysis, functions are set-up for each node of the CFG and solved by repeatedly calculating the output state from the input state locally at each node until the whole system stabilises and reaches its fix-point. In each iteration, a statement is removed from the work list/queue. Its out-state is computed. If the out-state changed, the statement's successors are added to the work list (as they require recalculation).

Each data-flow analysis type defines a configuration with its own **transfer** and **join** function as well as providing an initial value for the input state. Due to this generic approach, the data-flow analysis algorithm can be used for any kind of flow-sensitive inference of state (e.g. "potentially taken locks per program point", "potentially started threads", "guaranteed joined threads", "accessed locks up to program point" etc.).

As the data-flow analysis uses forward flow analysis, the output state of a block b (a single node in the CFG) is a function of the block's input state. The **transfer** function works on the input state (in_b), returning the output state (out_b).

$$out_b = transfer_b(in_b)$$

Figure 3.4: Data-flow analysis transfer function

The **join** function combines the exit states of the predecessors yielding the input state. To improve performance the **join** function is only applied if there is more than one incoming edge.

$$in_b = join_{p \in predecessors_b}(out_p)$$

Figure 3.5: Data-flow analysis join function

Algorithm 1: Intra-procedural data-flow analysis algorithm. (Previous incremental method)

```
Data: A set of statements from CFG
Result: Intra-procedural data-flow analysis fix-point
OUT[ENTRY] = ∅;
for each statement S other than ENTRY do
    OUT[S] = ∅;
end
while changes to any OUT occur do
    for each statement S other than ENTRY do
        IN[S] = join(OUT[predecessor statement]);
        OUT[S] = transfer (IN[S]);
    end
end
```

Algorithm 2: Intra-procedural data-flow analysis algorithm. (Worker list/queue method)

```
Data: A set of statements from CFG
Result: Intra-procedural data-flow analysis fix-point
for each statement S do
    OUT[S] = init(S); WORKQUEUE.Enqueue(S);
end
while WORKLIST not EMPTY do
    S = WORKLIST.Dequeue(); IN[S] = join(OUT[predecessor statement]);
    OUT[S] = transfer (IN[S]);
    if OUT[S] changed then
        WORLIST.Enqueue(sucessor statement]);
    end
end
```

Example

The data-flow analysis is configured to return a **set of active threads** through listing 3.4's execution path and is defined as followed:

state

The state is the set of potentially started threads. The initial state of every statement is defined as an empty set.

transfer function

Returns the input state and if the *Start* method is called on a thread, the started thread is added.

join function

If there is more than one input state, the set of all distinct elements (set union) in all the preceding output states (sets of active threads) is taken.

Listing 3.4: Example code for data-flow analysis.

```
1  static void A(int a, int b)
2  {
3      if (a > b)
4      {
5          Thread t1 = new Thread(() => { });
6          t1.Start();
7      }
8
9      while(a < 5)
10     {
11         Thread t2 = new Thread(() => { });
12         t2.Start();
13     }
14
15 }
```

Figure 3.6: Resulting CFG of listing 3.4.

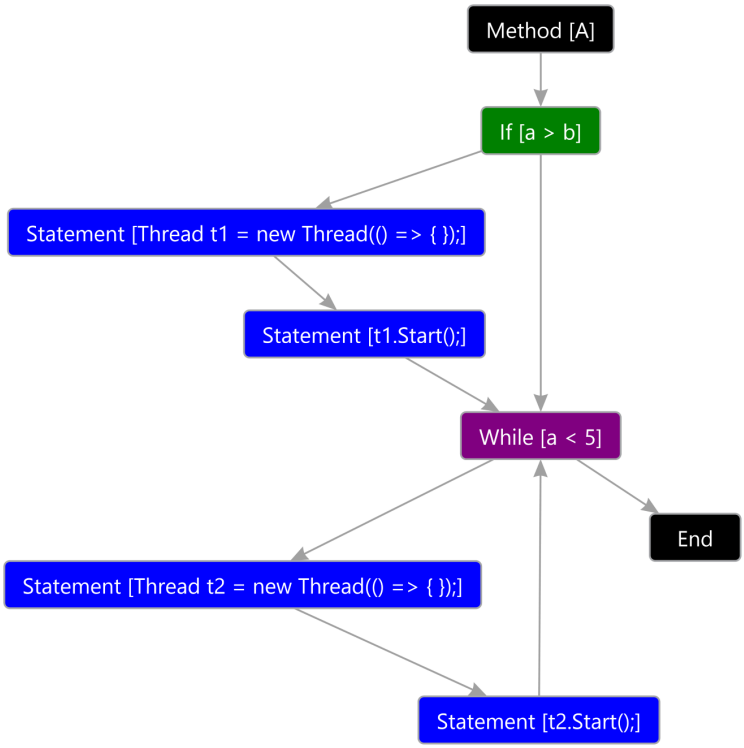
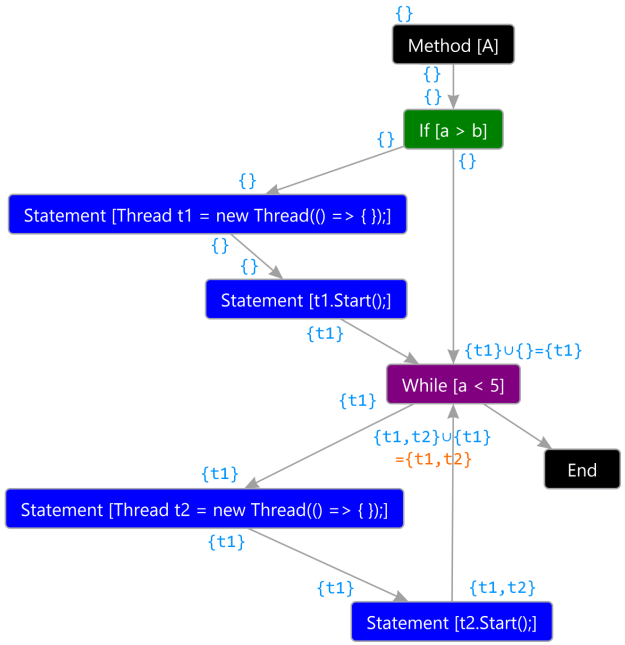
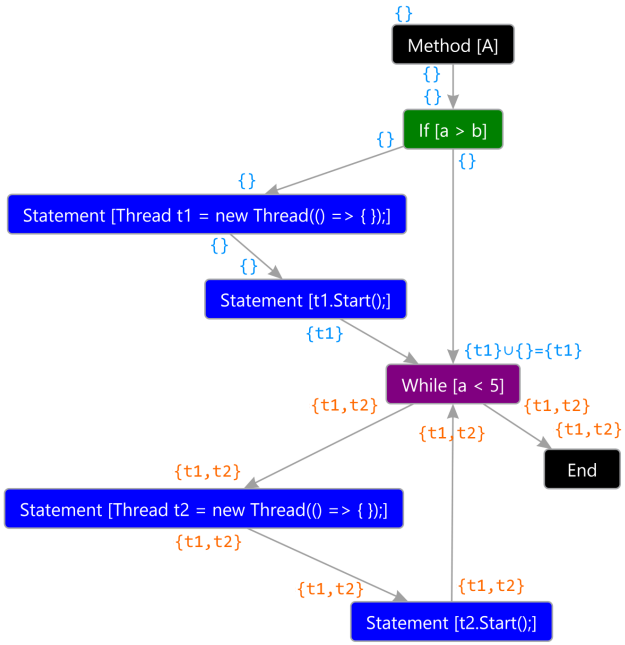


Figure 3.7: First pass through data-flow analysis.



The input state can be seen above a statement, the output state directly below a statement after it has been through the transfer function.
After the first data-flow analysis pass, the system is unstable as the output states differ after the *t1.Start* statement from their initial value (**{}** -> **{t1}**).

Figure 3.8: Second pass through data-flow analysis.

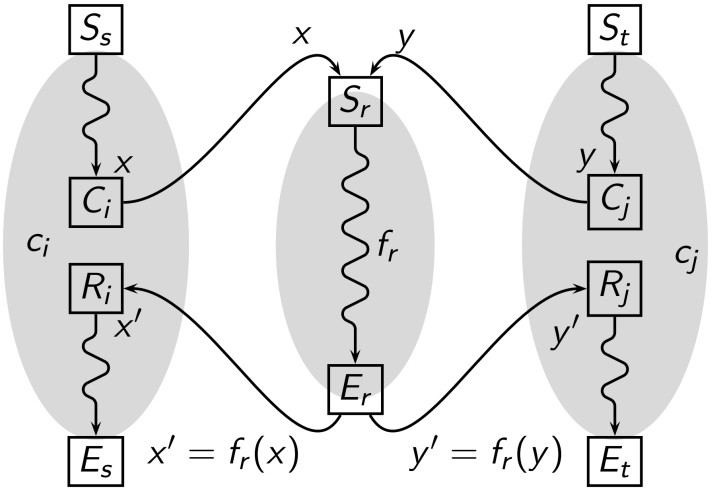


During the second data-flow analysis pass, all statements up to the **while** statement produce the same output state. However, as the result of the **while** loop ({t1,t2}) is joined with that of the if statement edges ($\{t1\} \cup \{\}$), a new output state is produced: **{t1,t2}**. This causes the subsequent statements to also produces different output states than during the first pass. When processed again, the output states will be the same, signalling that the intra-procedural data-flow analysis has stabilised and has reached its fix-point.

3.2.3 Inter-procedural Data-Flow Analysis

Inter-procedural data-flow analysis extends the scope of analysis across procedure boundaries and incorporates the effects of procedure calls in the caller procedures, and calling contexts in the callee procedures. Data-flow information is inherited through different procedures to obtain a complete inter-procedural data-flow analysis. The inter-procedural analysis is confined per thread (not intra-thread analysis), i.e. *new Thread(lambda)*, the lambda will not be considered as executed code of the data-flow.

Figure 3.9: Inherited and Synthesised Data-Flow Information. Figure taken from [5].



x	Inherited by procedure S_r from call site c_i in procedure S_s
y	Inherited by procedure S_r from call site c_j in procedure S_t
x'	Synthesised by procedure S_r in S_s at call site procedure C_i
y'	Synthesised by procedure S_r in S_t at call site procedure C_j

In fig. 3.9 c_i is the invocation node for a procedure. Node C and R represent the call (C) and return (R) of the procedure. These two nodes are implemented in the callee invocation node of the CFG.

Inter-procedural analysis uses intra-procedural analysis of multiple CFGs. To perform inter-procedural analysis, nodes in the CFG that reference other procedures join the current input state of the node with the input state of referenced procedure node, residing in a different CFG. When reaching an exit node the inherent output state is joined as the output state of the referenced callee node. Alike the intra-procedural data-flow analysis, this is solved by repeatedly calculating the output from the input for each callee node until the whole system stabilises and reaches its fix-point.

Algorithm 3: Inter-procedural data-flow analysis algorithm.

```
Data: A set of CFG
Result: Inter-procedural data-flow analysis fix-point
while changes to any OUT occur do
    for each cfg C do
        for each basic statement S do
            if S is 'Invoke' then
                IN[ C[S] ] = join(OUT[predecessor statement]);
            end
            if S is 'Exit' then
                OUT[ REF[S] ] = join(OUT[predecessor statement]);
            end
        end
    end
end
```

Example

The inter-procedural data-flow analysis example builds upon that of the intra-procedural data-flow analysis example. Taking the following inter-procedural functions into account:

- When an *Invoke* node is reached in the CFG the current input state is joined with the input state of the callee procedure.
- When the *End* node is reached in the CFG the output state is transferred (and joined) to the originating *Invoke* node.

To simplify the example, only the thread names are placed in the set of active threads. In a proper data-flow analysis, semantic information would be used to differentiate threads by their scope and location, so that *t1* in one procedure would **not** be considered equal to a different *t1* in a different scope.

Listing 3.5: Example code for inter-procedural data-flow analysis.

```
1  static void A(int a, int b)
2  {
3      while (a > b)
4      {
5          B();
6          Thread t1 = new Thread(() => { });
7          t1.Start();
8      }
9  }
10
11 static void B()
12 {
13     Thread t2 = new Thread(() => {});
14     t2.Start();
15 }
```

Figure 3.10: Resulting CFG of listing 3.5 (Method A).

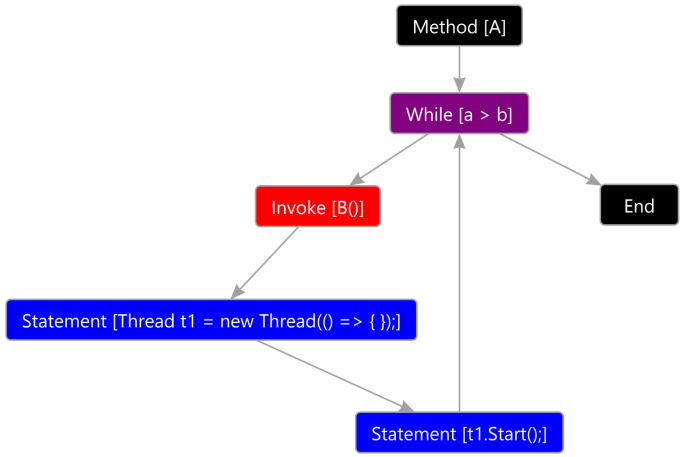


Figure 3.11: Resulting CFG of listing 3.5 (Method B).

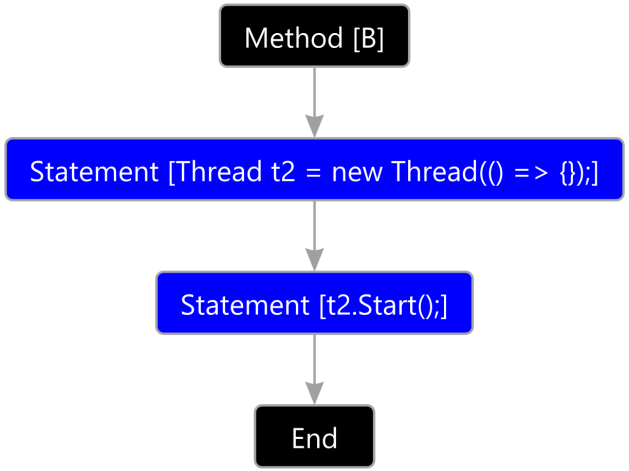
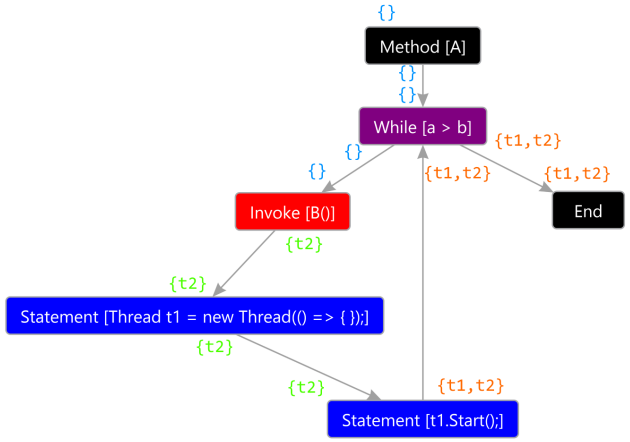
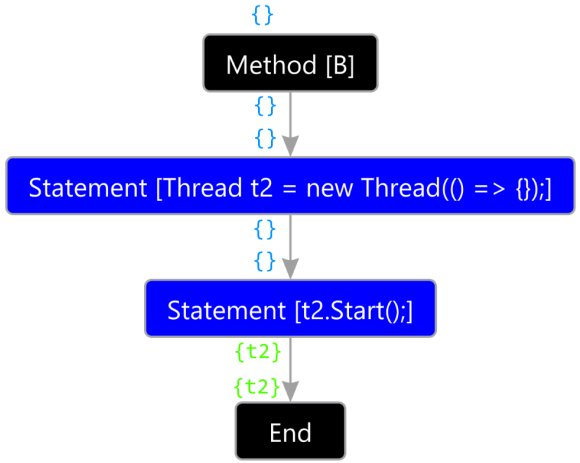


Figure 3.12: First pass through data-flow analysis (Method A).



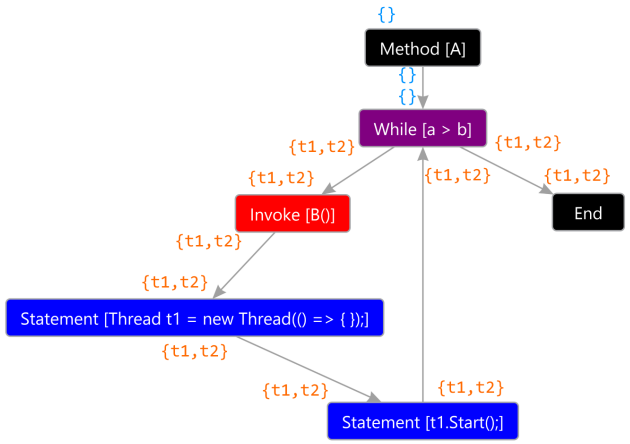
The input state can be seen above a statement, the output state directly below a statement after it has been through the transfer function.
After the first data-flow analysis pass the intra-procedural data-flow analysis of **A** is unstable as the output states differ after the *Invoke(B())* statement (**{}** -> **{t2}**) and *t1.Start* statement (**{t2}** -> **{t1,t2}**).

Figure 3.13: First pass through data-flow analysis (Method B).



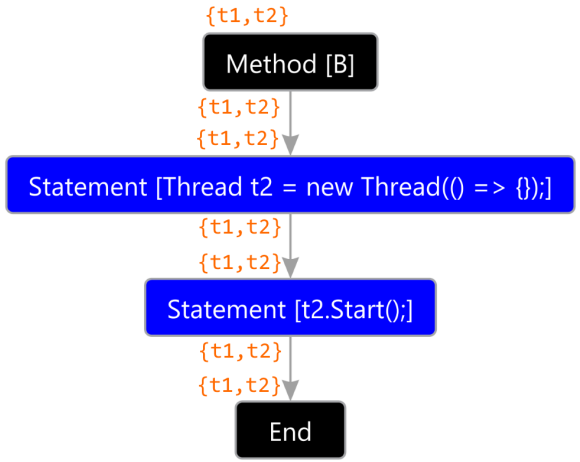
Notice how the input state of *Method(B)* is inherited from *Invoke(B())*.
After the first data-flow analysis pass the intra-procedural data-flow analysis of **B** is unstable as the output states differ after the *t2.Start* statement from the initial value.

Figure 3.14: Second pass through data-flow analysis (Method A).



During the second data-flow analysis pass, all statements up to the **while** statement produce the same output state. However, as the result of the **while** loop ({t1},{t2}) is joined with that of the predecessor output state ({}, {t1}) a new output state is produced: **{t1,t2}**. This causes the subsequent statements to also produce different output states than during the first pass. When processed again, the output states will be the same, signalling that the intra-procedural data-flow analysis for method **A** has stabilised and has reached its fix-point.

Figure 3.15: Second pass through data-flow analysis (Method B).



Notice how the new input state of *Method(B)* is inherited from *Invoke(B())*. This causes the **transfer** function to produce different output states than during the first pass. When processed again, the output states will be the same, signalling that the intra-procedural data-flow analysis for method **B** has stabilised and reached its fix-point.

3.3 Thread Analysis

The concurrency checker uses a novel static thread-start/join sensitive detecting for concurrency errors. This algorithm has been invented and developed by Prof. Dr. Luc Bläser, and a patent has been submitted[1]. In this chapter, the algorithm used for detecting low-level data races is explained. The algorithm engages the aforementioned data-flow analysis. The data race detection is split into three stages and is designed to perform a real static concurrency analysis that considers inter-procedural thread dependency graphs with start/join relations, a proper flow-sensitive analysis considering control statement as well as supporting various explicit and implicit thread entrance point.

Stage 1

Gather all static threads and their start/join relations.

Result

Graph with static threads as nodes and edges representing mutual start/join dependencies and collects access sets incl. lock sets.

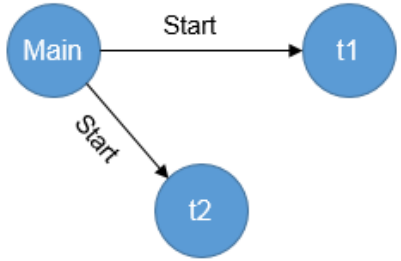


Figure 3.16: Static thread graph.

Stage 2

Data-flow analysis per thread

Result

Race conditions between threads and their child threads and pairs of fully parallel threads.

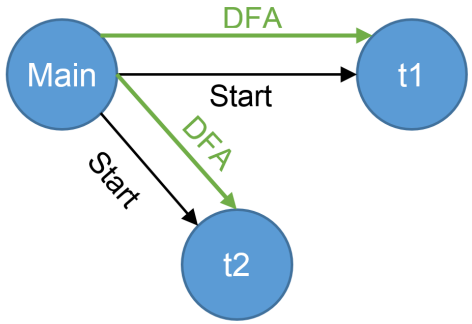


Figure 3.17: Intra-Thread data-flow analysis per thread and child threads.

Stage 3

Fully parallel thread analysis

Result

Race conditions between fully parallel thread pairs.

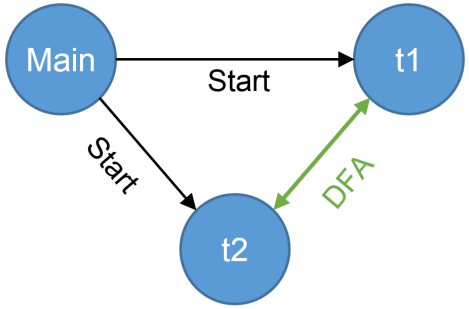


Figure 3.18: Intra-Thread data-flow analysis for fully parallel pairs.

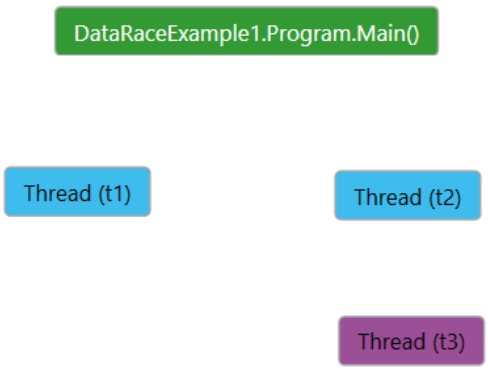
[1] Patent submitted, Swiss Federal Institute of Intellectual Property, 2015-05-04

3.3.1 Stage 1: Transitive Child Thread Analysis

A thread graph is a directed graph representing threads in a program connected by edges containing information about start/join states. The thread graph contains all **potential starts** and only **guaranteed joins** guaranteeing a conservative analysis, these are defined later. Stage 1 also collects all accesses with their corresponding lock sets (through inter-procedural data-flow analysis).

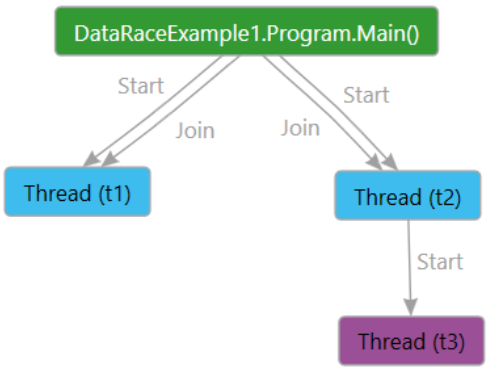
The first step in creating the thread graph is by resolving all thread graph nodes, such as the program entry (i.e. Main) and lambda expressions from new thread instances, with an inter-procedural data-flow analysis. These are "static threads", multiple instances of each of them may be created at run-time (except for the program entry, i.e. Main).

Figure 3.19: Result of static thread resolution of listing 3.1.



There are currently no edges between the different node as the data-flow must first be analysed. The analysis is done for each thread through inter-procedural data-flow analysis and resolves which threads potentially start and are guaranteed to join.

Figure 3.20: Thread graph including start and join edges.



In our example Thread *t2* starts Thread *t3*, therefore as Main starts Thread *t2* it also implicitly starts Thread *t3*. This is important as the Thread *t3* may have data accesses conflicting with those of the Main thread. All threads that are started directly or indirectly are considered child threads.

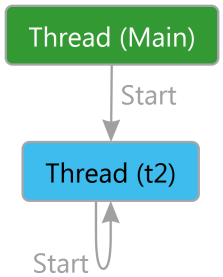
Static Threads

During analysis threads are static, i.e. "places of thread instantiation in code". The same "static thread" could lead to multiple "thread instances" at run-time. This implies that the thread graph can have cycles.

Listing 3.6: Example code with cycle.

```
1  public static void Main(string[] args)
2  {
3      Func();
4  }
5
6  public static void Func()
7  {
8      new Thread(() => Func()).Start();
9  }
```

Figure 3.21: Thread graph with cycle



In listing 3.6 *new Thread()* defines a "static thread" this leads to a graph with a node for this static thread - the static thread starting itself. This becomes a chain of thread instances at run-time.

Potential Start

If a thread is started during any of the program's execution paths, this will be considered as a potential thread start. This conservative approach is taken in order not to miss any potential concurrency.

Listing 3.7: An example of a potential thread start.

```
1  var t1 = new Thread(() => { });
2  bool start = Random.Next()%2 == 0;
3
4  if (start)
5    t1.Start();
```

Guarenteed Joins

If a thread joins all started instances of a static thread in all execution paths, this will be considered a guaranteed thread join. This conservative approach is taken in order not to miss any remaining potential concurrency.

Listing 3.8: An example of a thread join that cannot be guaranteed.

```
1  var t1 = new Thread(() => { });
2  t1.Start();
3
4  if (a > b)
5  {
6    t1.Join();
7  }
8  // t1 join not guaranteed
```

Multiple Start

If a thread is started several times, the thread becomes unjoinable and any subsequent joins to the thread will be ignored as joining cannot be guaranteed. This can occur if a reference to a thread is overridden before it has been joined.

Listing 3.9: An example of a thread started multiple times. An undefined number of threads are produced, due to a data race and only the last thread can be joined.

```
1  bool terminated = false;
2  Thread t2;
3
4  while (!terminated)
5  {
6    t2 = new Thread(() => terminated = !terminated);
7    t2.Start();
8  }
9
10 t2.Join();
```

3.3.2 Stage 2: Inter-Thread Data-Flow Analysis per Thread

This stage finds all data races among parent-child threads, being sensitive to thread starts and joins. For each thread, the child threads are computed by data-flow analysis. Active threads are directly started threads and indirectly started threads (extrapolated from the thread graph). Again, potential starts and guaranteed joins are used as concepts. As depicted in listing 3.1 thread *t1* and *t2* are child threads of the Main thread, and thread *t3* is a child thread of *t2*. Thread *t3* is therefore started indirectly from the Main thread.

Each read/write access in a statement, is compared against the active child threads defined for that statement. If there is any data race conflict - access to the same memory location (i.e. variable), involving at least one write access - it is reported.

Result

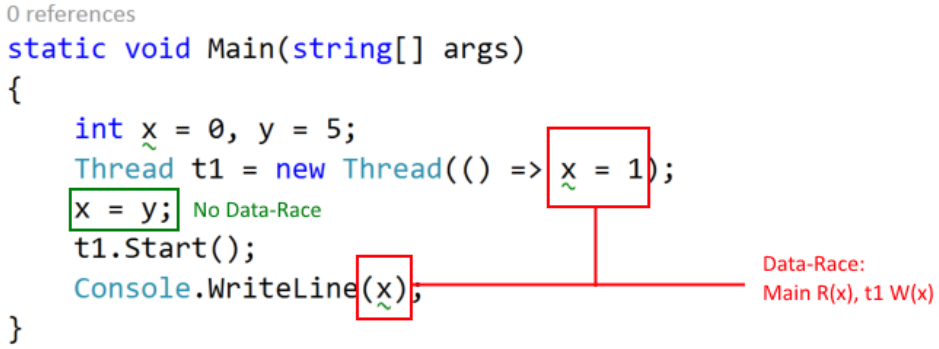
As depicted in Listing 3.10 the variable *x* is modified (line: 5) before the child thread is started - which therefore does not cause a data race. The read-access (line: 7) on the other hand is a clear data race with the child thread *t1* (line: 4).

Listing 3.10: An example indicating the importance of flow-sensitive data-flow analysis.

```
1        static void Main(string[] args)
2        {
3            int x = 0, y = 5;
4            Thread t1 = new Thread(() => x = 1);
5            x = y;
6            t1.Start();
7            Console.WriteLine(x);
8        }
```

The result in Visual Studio depicts a data race warning for *x* through the inter-thread data-flow analysis.

Figure 3.22: Data races from listing 3.10 depicted in Visual Studio 2015.



Inference of fully parallel threads

"Child threads" before and after statements are compared. For each new child thread after the statement that does not yet exist before the statement, a pair is added between this new thread and all previous child threads.

As thread *t1* and thread *t2* are both child threads of the Main thread and active at the same time in listing 3.1 they become a fully parallel pair. Thread *t3* is a child thread of *t2*, this results in thread *t1* and thread *t3* becoming a fully parallel pair.

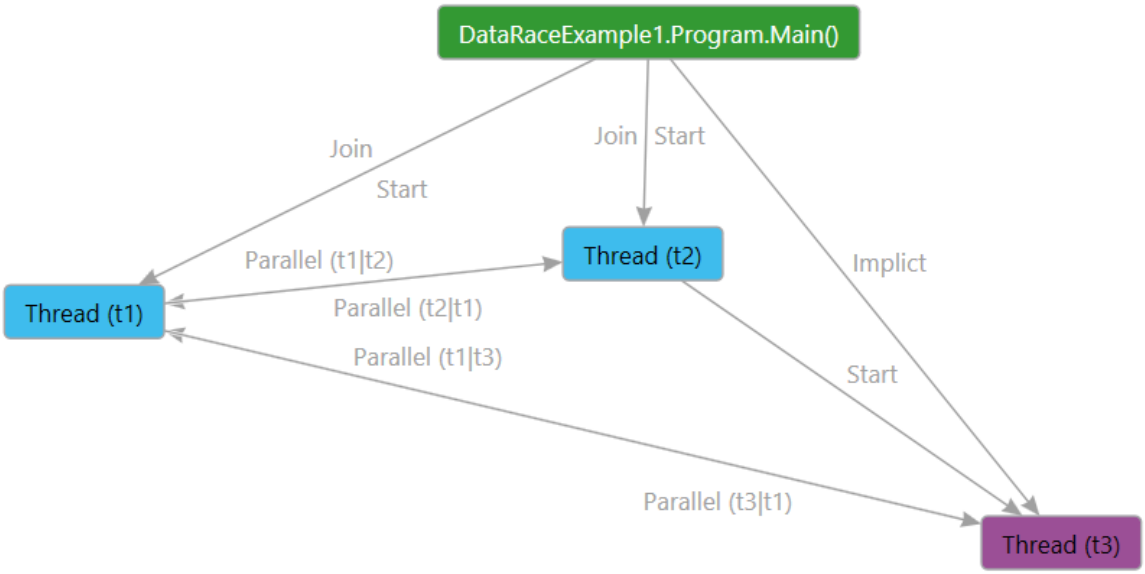


Figure 3.23: Complete thread graph with start/join states, implicit starts and fully parallel references.

3.3.3 Stage 3: Fully-Parallel Analysis

With the new information in the thread graph won in stage 2, it is now known which thread pairs run unconditionally in parallel, i.e. without any parent/child relation. Each fully-parallel-pair's entire access sets are compared for conflicts.

Result

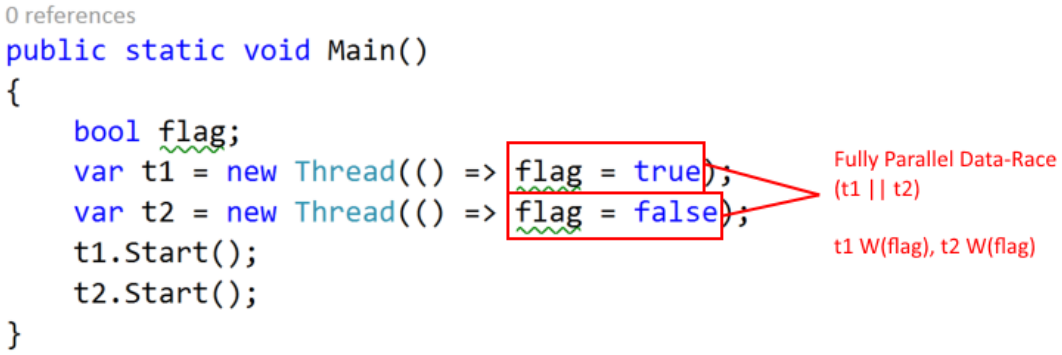
As depicted in listing 3.11 there are two data-accesses on *x* from *t1* and *t2*. As there is no data-access for *x* on the main thread stage 2 does not detect a conflict with its child threads. Only by comparing the access sets of the fully parallel pair (*t1* and *t2*) is the data race found.

Listing 3.11: An example of two fully-parallel threads.

```
1  public static void Main()
2  {
3      bool flag;
4      var t1 = new Thread(() => flag = true);
5      var t2 = new Thread(() => flag = false);
6      t1.Start();
7      t2.Start();
8  }
```

The result in Visual Studio depicts a data race warning for *x* through the fully-parallel analysis.

Figure 3.24: Fully-Parallel data races depicted in Visual Studio 2015.

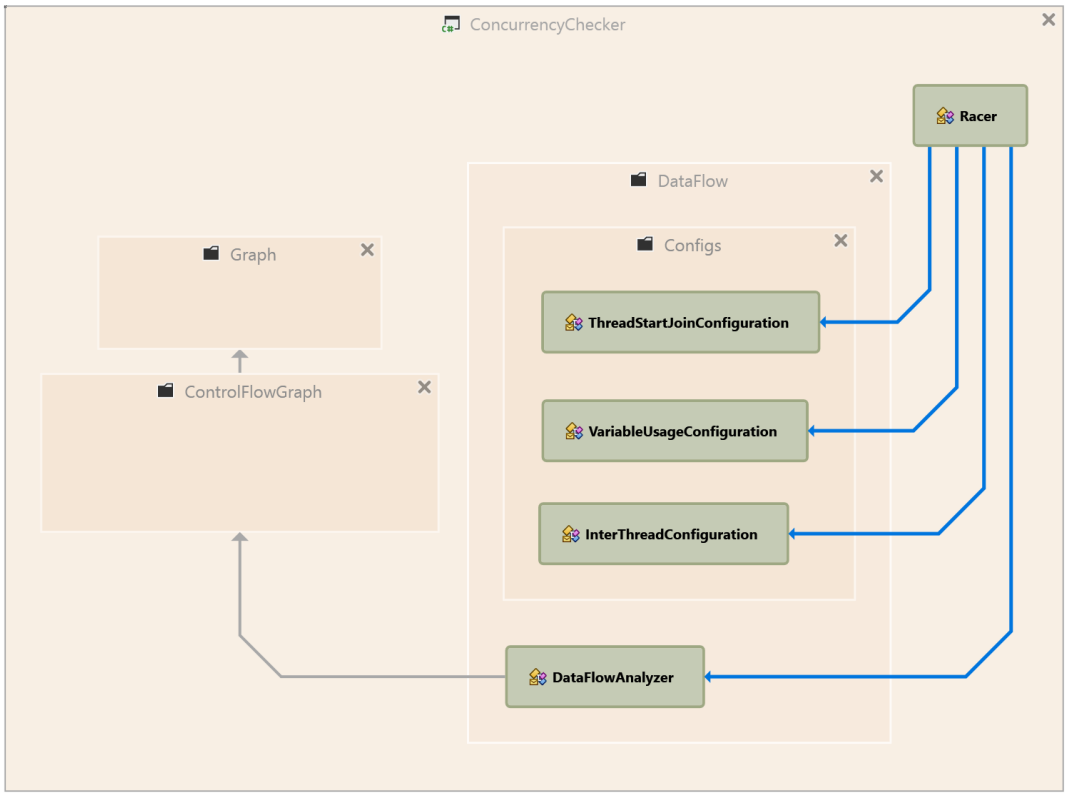


3.4 Implementation

3.4.1 Architecture

The concurrency checker has an architecture in which the data race analyser, data-analysis, corresponding configurations, control graph generator and graph utilities are segregated into different logical layers.

Figure 3.25: Dependency diagram of the core project generated with ReSharper 9.1



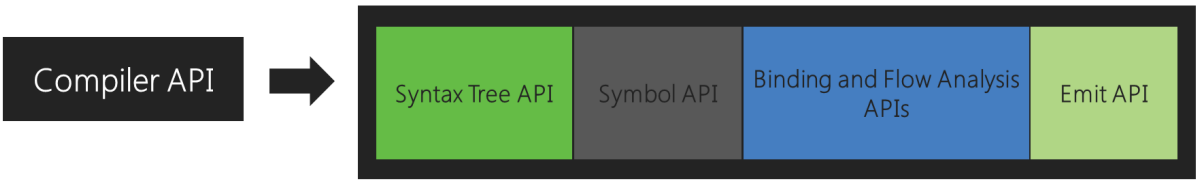
3.4.2 Roslyn

The .NET Compiler Platform ("Roslyn") includes self-hosting versions of the C# and VB.NET compilers – compilers written in the languages themselves. The compilers are available via the traditional command-line programs but also as APIs available natively from within .NET code. Roslyn exposes modules for syntactic (lexical) analysis of code, semantic analysis, dynamic compilation to IL, and code emission [7].

At the time of writing, the concurrency checker uses the release candidate 2 (RC2) version of Roslyn. RC2 is to be considered stable and no public API changes should occur until v1 has been released and fully integrated into Visual Studio 2015 as the default compiler.

As Roslyn does not provide a native CFG implementation, nor a public inter- or intra-data-flow analysis, the concurrency checker must take advantage of the Roslyn's Compiler API to generate its own CFG and implement its own data-flow analysis.

Figure 3.26: An overview of available API's from Roslyn's code analysis (Compiler API).



Syntax Tree API

The Syntax Tree API exposes the syntax tree of the source code, containing every bit of information in a code file, including elements like comments or whitespace. Writing a syntax tree to text will reproduce the exact original text that was parsed. In Visual Studio the syntax tree is regenerated after nearly every new character.

Symbol API

The Symbol API provides the semantic model for a syntax tree and is essential for enriching the CFG and data-flow analysis as it is possible to query the following:

- What names are in scope at this location?
- What members are accessible from this method?
- What variables are used in this block of text?
- What does this name/expression refer to?

Visual Studio refreshes the semantic model after (nearly) every statement change. The syntax tree is updated more often that the semantic model, it is however guaranteed due to Roslyn's immutable data structures that semantic model and syntax tree correlate.

The concurrency checker has been implemented so that it only relies on the Roslyn API (and not the Visual Studio APIs). It is, therefore, possible for the concurrency checker to be used as a

library and is not bound to Visual Studio. This could be useful, for example, in creating an online service that checks code for concurrency errors.

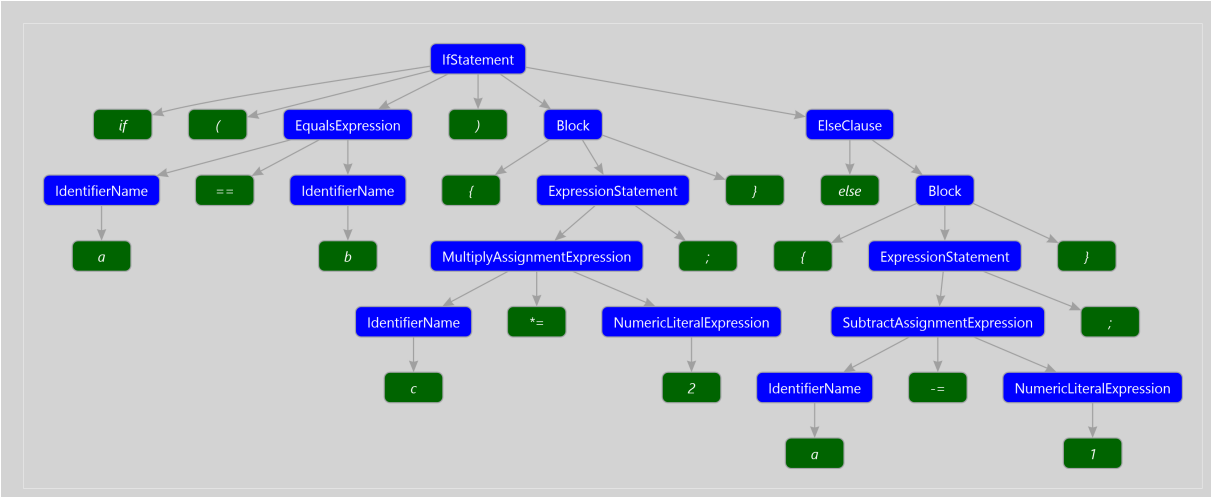
3.4.3 Control Flow Graph

As Roslyn does not provide a CFG implementation, it has to be generated from the AST (Abstract Syntax Tree) using visitors. After creation of the CFG, visitors are no longer required in the analysis.

Syntax Tree

The Syntax Tree API provides several methods of accessing/traversing the abstract syntax tree generated by Roslyn. Each node of the tree denotes a construct occurring in the source code. Roslyn offers several ways to traverse the abstract syntax tree, either through a visitor (SyntaxVisitor), walker (SyntaxWalker) or LINQ.

Figure 3.27: Visual representation of the syntax tree for listing 3.12 generated in Visual Studio, Trivia has been omitted for better readability



The SyntaxWalker represents a SyntaxVisitor that descends an entire SyntaxNode graph visiting each SyntaxNode and its child SyntaxNodes and SyntaxTokens in depth-first order. It however explicitly processes and includes TriviaNodes (comments, whitespaces, etc.) that are not necessary in the generation of the CFG.

By implementing a custom SyntaxVisitor that visits the entire SyntaxNode graph instead of just a single SyntaxNode, execution time was reduced by ca 50% compared to utilising a SyntaxWalker (this was mainly part to Roslyn not having to re-parse the source for the TriviaNodes).

Statements

Each C# statement is represented by its own statement class inside the syntax tree. Statements such as foreach, switch, try, catch have been not been implemented in this thesis due to their complexity. The following statements, defined in the C# language specification [2], can be processed by the CFG.

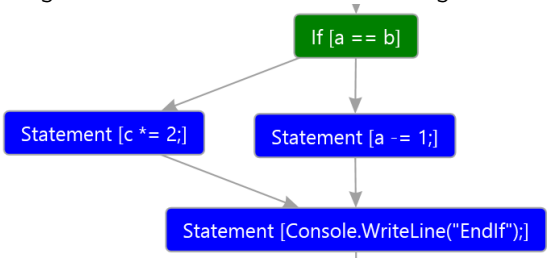
If-Else, Else-If

An if statement identifies which statement to run based on the value of a Boolean expression. Because the condition cannot be simultaneously true and false, the then-statement and the else-statement of an if-else statement can never both run.

Listing 3.12: If-Else statement

```
1  if (a == b)
2  {
3      c *= 2;
4  }
5  else
6  {
7      a -= 1;
8  }
```

Figure 3.28: Partial CFG for listing 3.12

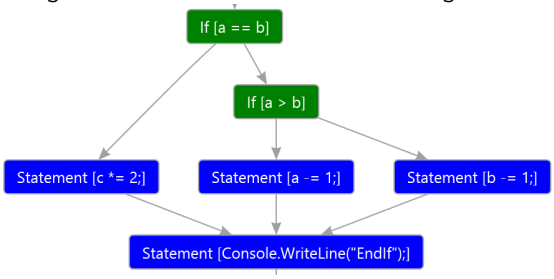


As an if statement can only have a single then- and else-statement 'else if' is transformed to a nested if-else statement.

Listing 3.13: If-Else If-Else statement

```
1  if (a == b)
2  {
3      c *= 2;
4  }
5  else if (a > b)
6  {
7      a -= 1;
8  }
9  else
10 {
11     b -= 1;
12 }
```

Figure 3.29: Partial CFG for listing 3.13



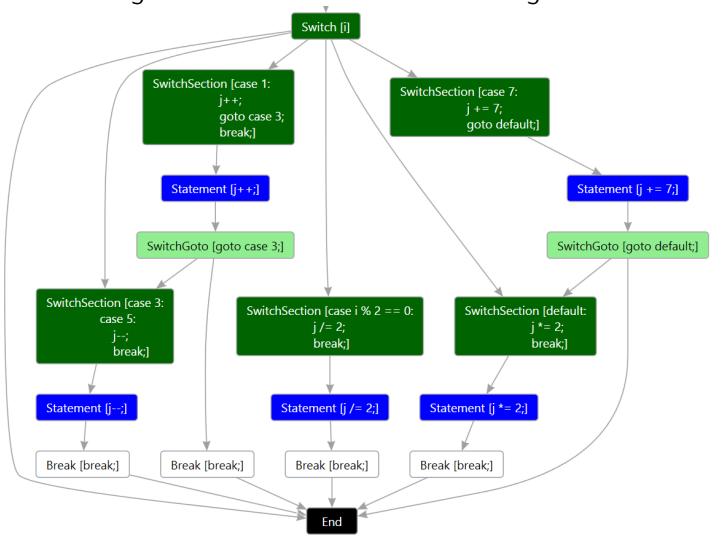
Switch

A switch statement includes one or more switch sections. Each switch section contains one or more case labels followed statements. By the use of goto statments the exection can jump to another section. When reaching break statements the switch section is exited.

Listing 3.14: Switch statement

```
1   switch (i)
2     {
3         case 1:
4             j++;
5             goto case 3;
6             break;
7         case i % 2 == 0:
8             j /= 2;
9             break;
10        case 3:
11        case 5:
12            j--;
13            break;
14        case 7:
15            j += 7;
16            goto default;
17        default:
18            j *= 2;
19            break;
20    }
```

Figure 3.30: Partial CFG for listing 3.14



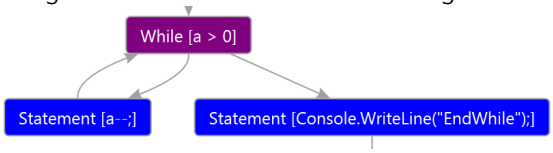
While

The while statement executes a statement or a block of statements until a specified expression evaluates to false.

Listing 3.15: While statement

```
1    while (a > 0)
2    {
3        a--;
4    }
```

Figure 3.31: Partial CFG for listing 3.15



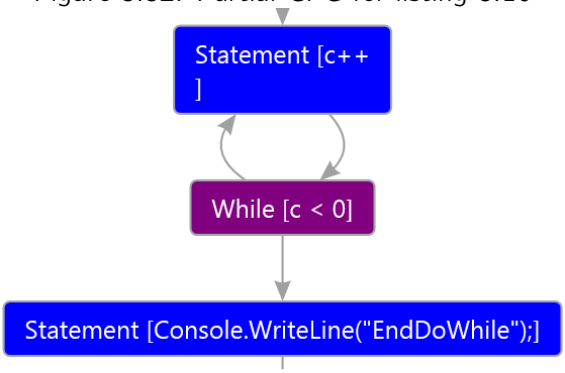
Do-While

The do statement executes a statement or a block of statements repeatedly until a specified expression evaluates to false. A do loop executes one or more times.

Listing 3.16: Do statement

```
1    do
2    {
3        c++
4    }
5    while(c < 0)
```

Figure 3.32: Partial CFG for listing 3.16



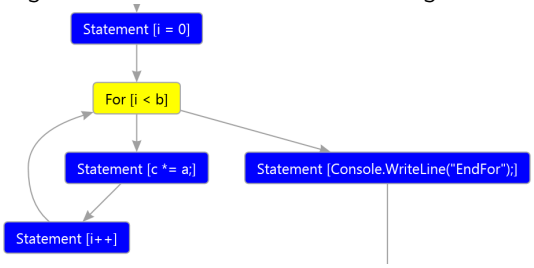
For

The for statement executes a statement or a block of statements repeatedly until a specified expression evaluates to false. Unlike a do or while statement, a for statement has a initialiser section that sets the initial conditions and an iterator section that defines what happens after each iteration of the body of the loop.

Listing 3.17: For statement

```
1    for (var i = 0; i < b; i++)
2    {
3        c *= a;
4    }
```

Figure 3.33: Partial CFG for listing 3.17



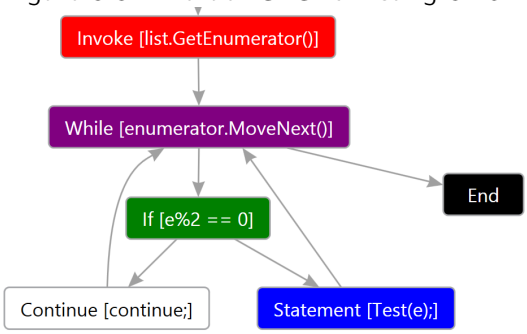
Foreach

The foreach statement repeats a group of embedded statements for each element in an array or an object collection that implements the IEnumerable interface. The foreach statement is used to iterate through a collection but can not be used to add or remove items from the source collection to avoid unpredictable side effects. Each foreach statement is transformed into while-statement enclosed by a try-finally statement. For simplicity the try-finally was omitted.

Listing 3.18: Foreach statement

```
1   foreach (var e in list.↩
        ToArray())
2   {
3       if (e%2 == 0)
4       {
5           Test(e);
6       }
7       else
8       {
9           continue;
10      }
11  }
```

Figure 3.34: Partial CFG for listing 3.18



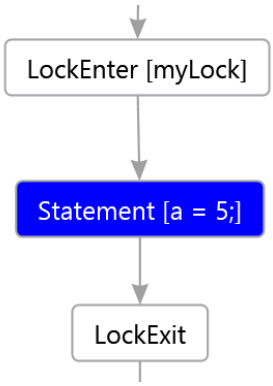
Lock

The lock keyword marks a statement block as a critical section by obtaining the mutual-exclusion lock for a given object, executing a statement, and then releasing the lock.

Listing 3.19: Lock statement

```
1   lock (myLock)
2   {
3       a = 5;
4   }
```

Figure 3.35: Partial CFG for listing 3.19



Jump Statements

Jump statements cause an immediate transfer of the program control. As the syntax tree visitor is depth-first, there is a stacked scoping to handle jump statements. The stack is manually pushed and popped depending on the current syntax node, this requires that the first node passed to the CFG generator be a scoping (method, lambda, property) node. Failure to pass a scoping node may cause jump statements not to be processed correctly/completely, due to missing scopes.

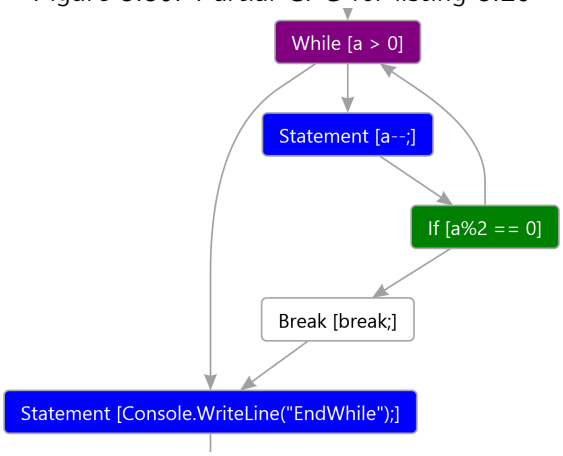
Break

The break statement terminates the closest enclosing loop or switch statement in which it appears. Control is passed to the statement that follows the terminated statement, if any.

Listing 3.20: While statement with break

```
1  while (a > 0)
2  {
3      a--;
4      if (a%2 == 0)
5          break;
6  }
```

Figure 3.36: Partial CFG for listing 3.20



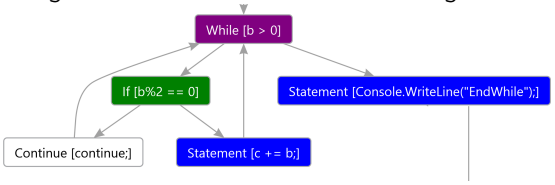
Continue

The continue statement passes control to the next iteration of the enclosing while, do, for, or foreach statement in which it appears.

Listing 3.21: While statement with continue

```
1  while (b > 0)
2  {
3      if (b%2 == 0)
4          continue;
5      c += b;
6  }
```

Figure 3.37: Partial CFG for listing 3.21



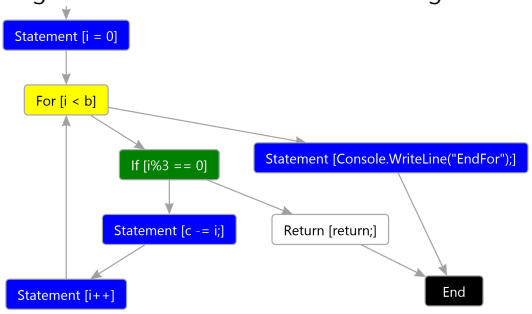
Return

The return statement terminates execution of the method in which it appears and returns control to the calling method.

Listing 3.22: For statement with return

```
1     for (var i = 0; i < b; i++)
2     {
3         if (i%3 == 0) return;
4         c -= i;
5     }
```

Figure 3.38: Partial CFG for listing 3.22



Intra-Procedural CFG

To extend the scope of the data-flow analysis across procedure boundaries a Control Flow Graph Container separates each procedure into its own control flow graph (CFG). A map is created to reference caller with the callee instead of inserting the procedure's CFG directly into the main CFG. This methodology was preferred as recursive procedures would cause an infinitely-long and indeterminate CFG. A typical example of recursive procedural calls is the fibonacci sequence:

Figure 3.39: CFG of main method.

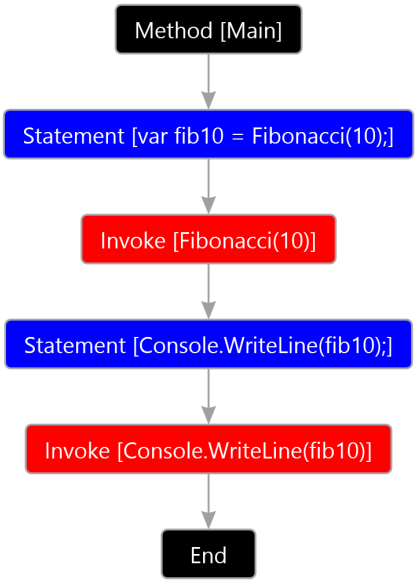
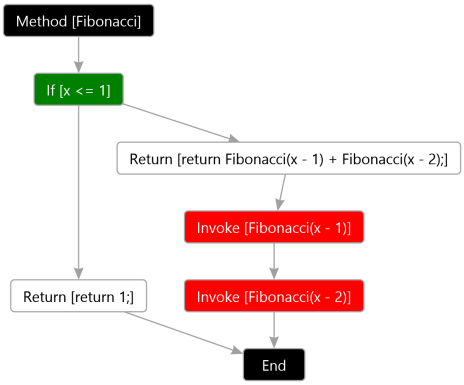


Figure 3.40: CFG of the recursive fibonacci method.



Invoke nodes (in red) are reference nodes that map with their method node (in black) in a corresponding CFG. Currently only procedures resolvable in the current abstract syntax tree are supported, neglecting precompiled IL-binaries (such as *Console.WriteLine(fib10)* in fig. 3.39).

As a performance optimisation CFGs are built only when needed. This reduces the memory foot-print to a minimum and would also allow for better caching.

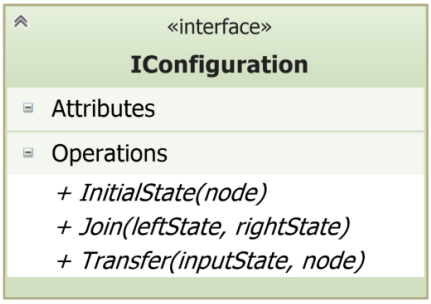
3.4.4 Data-Flow

Configuration

As the data-flow analysis is generic, allowing it to be used for any kind of flow-sensitive inference of state, it accepts a configuration to process a specific type of data-flow analysis.

The configuration is defined for each data-flow analysis with its own transfer and join function - each function returning the state defined by the configuration - as well as providing an initial value for the input state.

Figure 3.41: UML class diagram of configuration interface.



Any class with the configuration interface can be used for the data-flow analysis, this has the advantage that the class can be set up before being used, i.e. providing context information such as the current thread or the thread graph.

There currently exists three configurations used in the concurrency checker:

Name	Purpose	Used in
ThreadStartJoinConfiguration	Calculates thread state (start/join)	Stage 1
VariableUsageConfiguration	Calculates what variables are read / written and their locks	Stage 1
InterThreadConfiguration	Calculates race conditions between child-threads	Stage 2

Further configurations such as for pointer analysis or deadlock analysis can easily be added in the future.

State

As each configuration calculates different values, the values composing the (input/output) state of each configuration need to be definable by the configuration but understandable for the data-flow analysis (i.e. checking to see if the output states have changed), otherwise the fix-point cannot be reached. This is accomplished by implementing the generic **IEquatable(T)** interface that defines a type-specific method for determining equality of instances.

3.4.5 Visual Studio Integration

Diagnostic Analyser

Diagnostic analysers are available as of Visual Studio 2015 and allow for interaction with Roslyn's Compiler API. Developers can ship domain-specific code analysis as part of their NuGet packages, providing warnings or even errors for their package, assuring correct usage of their code. Code-fixes can also be provided, helping the developer even further. Diagnostic analysers can react to different Compiler API events such as when the syntax tree has been built, the semantic model is available or when compilation has occurred. Important to understand is that warnings and errors are not passed directly to Visual Studio but are added to Roslyn's API pipeline allowing for other diagnostic analysers to react on further information.

As the concurrency checker requires semantic information to build the CFGs an event is registered to be notified when the semantic model is available. This event may be triggered every few seconds. Once notified that the semantic model is available a CancellationToken is provided. If the provided semantic model and syntax tree become obsolete due to a large quantity of changes, a cancellation request is issued via the CancellationToken to stop further analysis. As Visual Studio queues events until the previous event has been processed, respecting cancellation requests causes diagnostic information (such as warnings or errors) to appear faster and more accurate to the user.

Error handling is very important as Roslyn uses best guesses in the abstract syntax tree if code is incomplete or contains syntactical errors. This causes exceptions in the CFG generation as the syntax tree is incomplete/erroneous. A global try-catch clause in the diagnostic analyser makes sure that no exceptions are thrown, as this could cause the diagnostic analyser, and even the plug-in, to be unloaded by Visual Studio.

VSIX Package

A Visual Studio Integration Extension (VSIX) package is a compressed file that follows the Open Packaging Conventions (OPC) standard [4]. The package contains binaries and supporting files, together with metadata required to classify and install the extension. VSIX packages can be distributed manually, through private galleries or published to Microsoft's Visual Studio Gallery.

By placing NuGet packages inside a VSIX package it becomes globally available in Visual Studio (instead of just on a project to project basis). These two components must be implemented as two separate projects (best practice).

3.5 Results

3.5.1 Precision

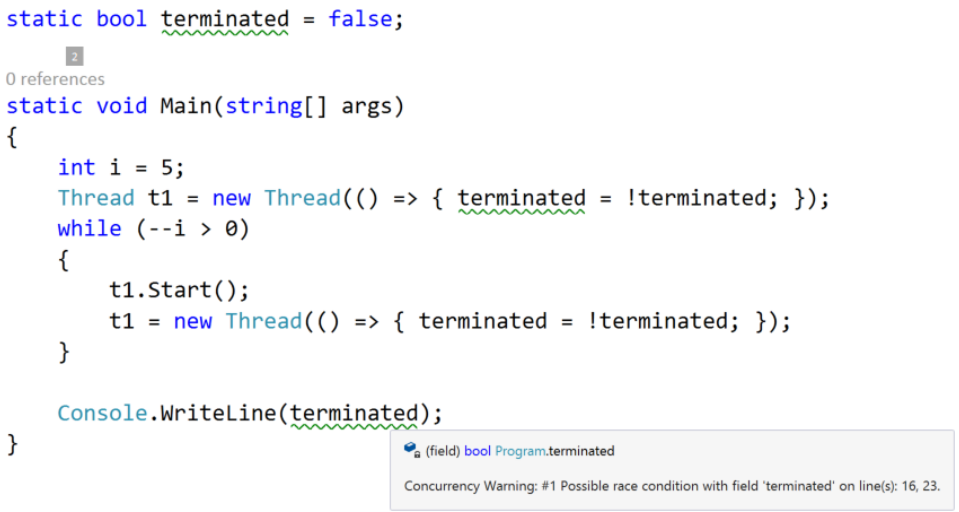
Due to the flow-sensitive redesign, the algorithm has become more precise than the existing IFS flow-insensitive prototype. The existing prototype would not be able to detect the data races in listing 3.23, this is no longer the case with the new concurrency checker as it now supports a flow-sensitive algorithm.

However, the concurrency checker does not fully account for all data races as pointer analysis would be required to detect references being replaced during execution, as can be seen in fig. 3.42.

Listing 3.23: Example code with multiple data races

```
1  static bool terminated = false;
2
3  static void Main(string[] args)
4  {
5      int i = 5;
6      Thread t1 = new Thread(() => { terminated = !terminated; });
7      while (--i > 0)
8      {
9          t1.Start();
10         t1 = new Thread(() => { terminated = !terminated; });
11     }
12
13     Console.WriteLine(terminated);
14 }
```

Figure 3.42: Data races detected in Visual Studio for listing 3.23.



3.5.2 Examples

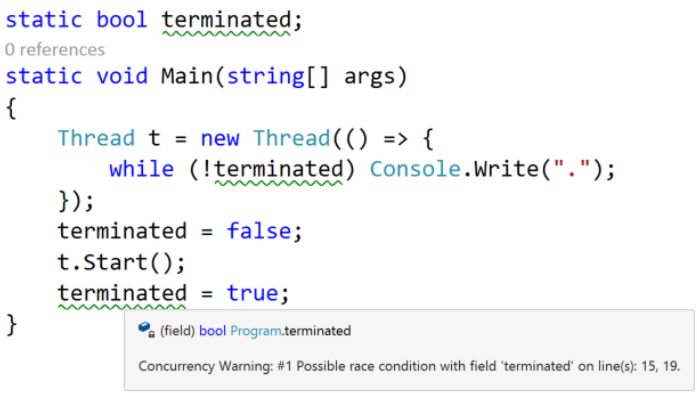
Based on the following examples the analysis in the concurrency checker worked as expected.

Simple

Listing 3.24: Example code with a single data race

```
1  static bool terminated;
2
3  static void Main(string[] args)
4  {
5      Thread t = new Thread(() => {
6          while (!terminated) Console.Write(".");
7      });
8      terminated = false;
9      t.Start();
10     terminated = true;
11 }
```

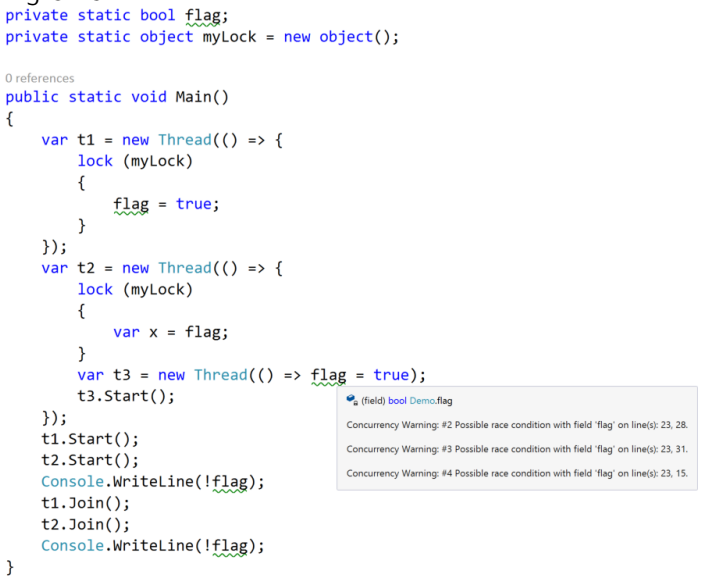
Figure 3.43: Data race detected in Visual Studio for listing 3.24.



Listing 3.25: Example code with several data races, also includes a mutual exclusion (lock)

```
1  private static bool flag;
2  private static object myLock = new object();
3
4  public static void Main()
5  {
6      var t1 = new Thread(() => {
7          lock (myLock)
8          {
9              flag = true;
10         }
11     });
12     var t2 = new Thread(() => {
13         lock (myLock)
14         {
15             var x = flag;
16         }
17         var t3 = new Thread(() => flag = true↩
               );
18         t3.Start();
19     });
20     t1.Start();
21     t2.Start();
22     Console.WriteLine(!flag);
23     t1.Join();
24     t2.Join();
25     Console.WriteLine(!flag);
26 }
```

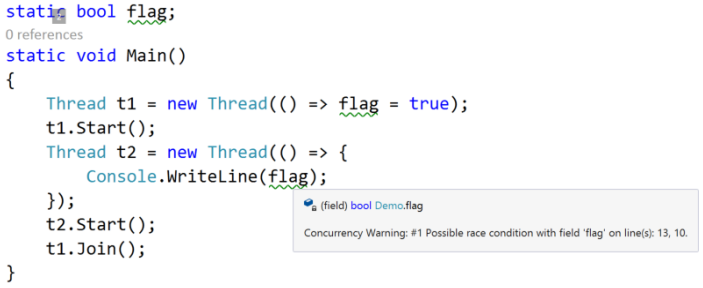
Figure 3.44: Data races detected in Visual Studio for listing 3.25.



Listing 3.26: Example code with a single data race in a fully-parallel pair

```
1  static bool flag;
2  public static void Main()
3  {
4      Thread t1 = new Thread(() => { flag = ↩
           false; });
5      t1.Start();
6
7      Thread t2 = new Thread(() => {
8          Console.WriteLine(flag);
9      });
10     t2.Start();
11     t1.Join();
12 }
```

Figure 3.45: Data races detected in Visual Studio for listing 3.26.



Inter-Procedural

Data races are found on field *balance* in listing 3.27 through inter-procedural data-flow analysis.

Listing 3.27: Example code with data races caused in a different procedure

```
1  class Bank
2  {
3      private int balance;
4      public void Deposit(int amount)
5      {
6          balance += amount;
7      }
8      public int Balance
9      {
10         get { return balance; }
11     }
12 }
13 static void Main()
14 {
15     Bank bank = new Bank();
16     Thread t1 = new Thread(() => {
17         bank.Deposit(100);
18     });
19     Thread t2 = new Thread(() => {
20         bank.Deposit(50);
21     });
22     t1.Start();
23     t2.Start();
24     Console.WriteLine(bank.Balance);
25 }
```

Figure 3.46: Data races detected in Visual Studio for listing 3.27.

```
2 references
class Bank
{
    private int balance;
    2 references
    public void Deposit(int amount)
    {
        balance += amount;
    }
    1 reference
    public int Balance
    {
        get { return balance; }
    }
}
0 references
static void Main()
{
    Bank bank = new Bank();
    Thread t1 = new Thread(() => {
        bank.Deposit(100);
    });
    Thread t2 = new Thread(() => {
        bank.Deposit(50);
    });
    t1.Start();
    t2.Start();
    Console.WriteLine(bank.Balance);
}
```

3.5.3 Performance

To test the performance of the algorithm a code example with mutual exclusions and two fully-parallel pairs was used. The computing time was calculated for each stage of the algorithm. The following aspects were not factored into the results: compile time, syntax tree and semantic model generation as well as passing diagnostic reports to Roslyn.

This thesis bases its findings for performance testing on a small example to classify the relative cost distribution of the algorithm. A more detailed evaluation on the performance and scalability of the algorithm requires further research.

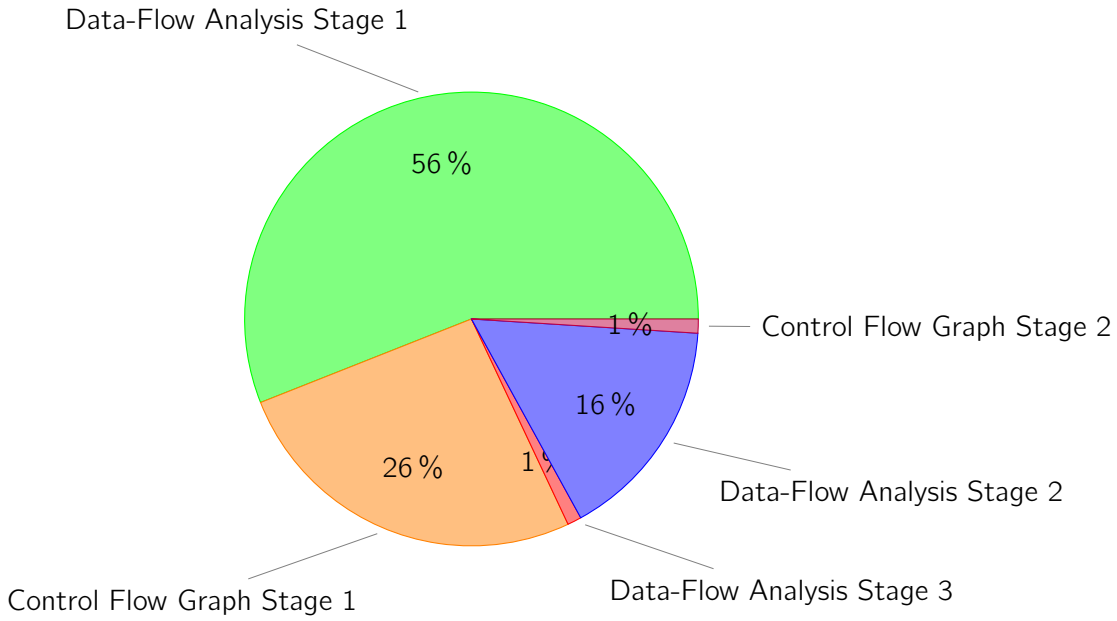
Listing 3.28: Example code used to test performance

```
    var t1 = new Thread(() => {
        lock (myLock)
        {
            flag = true;
        }
        var t4 = new Thread(() => flag = false);
        t4.Start();
    });
    var t2 = new Thread(() => {
        lock (myLock)
        {
            var x = flag;
        }
        var t3 = new Thread(() => flag = true);
        t3.Start();
    });
    t1.Start();
    t2.Start();
    Console.WriteLine(!flag);
    t1.Join();
    t2.Join();
    Console.WriteLine(!flag);
```

Iterative Algorithm

In total, the iterative algorithm completes in **0.18 s**, stage 1 being the most and stage 3 the least compute intensive. Surprisingly, after analysing execution logs, after the first CFG is generated the same CFG only takes a fraction of the time to generate, this is most likely due to a internal caching function in Roslyn.

Figure 3.47: Percentage of time used for each stage in the algorithm



As can be interpreted from fig. 3.47, the most performance gain could be achieved by optimising the CFG generation. This would entail caching a CFG once it has been generated. In total each procedure's CFG was generated 6 times during execution.

Moreover, the data-flow analysis could profit from a work list, only calculating the output states for changed predecessors as some data-flow configurations in stage 2 took 6 full iterations until it reached its fix-point.

Worker List

In total, the worker list algorithm completes in **0.01 s** - a 10x performance gain of the iterative algorithm.

3.5.4 Future Work

The concurrency checker works in its core properties. There are, however, various possibilities for further improvement, that were not addressed in this thesis.

General

- Implicit thread support (i.e. TPL)
- Support for delegates in Thread constructor
- Variable definitions are found by analysing the syntax-tree's predecessors, for a more flow-sensitive approach this should be done via the CFG
- Further scalability and performance evaluation of the algorithm

Control Flow Graph

- ~~Implement a memory or file cache, as the same CFGs are calculated multiple times during analysis~~
- Support for conditional operators (?:,??,?.)
- Support for ~~foreach,~~ try-catch ~~and switch~~ statements
- Support for constructor and field declarations

Data-Flow Analysis

- Analysis of pre-compiled IL-binaries by using reference source, meta-files or IL-parser (increase precision of data-flow analysis)
- Pointer aliasing (increase precision of data-flow analysis)
- Context-sensitive inter-procedural data-flow analysis - segregate state by call location, (increase precision of data-flow analysis)
- ~~Introduce a work list: In each iteration, a block is removed from the work list. Its out-state is computed. If the out-state changed, the block's successors are added to the work list (as they require recalculation).~~
- Implement deadlock and thread starvation detection configurations

Visual Studio Integration

- Detect program entry through default project in Visual Studio configuration
- Look into possible automatic code-fix providers for concurrency errors
- Highlighting in Visual Studio could be improved to visualise the execution path taken, by marking where a procedure was called from. As of current, it is not clear to the user which caller(s) is/are causing the data race

3.6 Conclusion

The resulting tool proves to be an effective flow-sensitive static checker that detects potential data races in C# program code within Visual Studio, highlighting and marking such issues during code writing in a non-disruptive manner.

With its flow-sensitive design taking all execution paths into account, low-level data races are detected with a higher precision than with the existing concurrency checker.

In contrast to other existing solutions, this checker performs a real static analysis (not only simple local bug pattern location) as a practical Visual Studio 2015 plugin compatible with the newest version of .NET's compiler platform "Roslyn". Due to the generic data-flow based architecture, the tool could be extended to also support deadlock detection in the future.

Appendix A

Task Description[1]

A.1 Supervisor

This bachelor thesis is conducted on an internal research project of the HSR Institute for Software.

A.1.1 Supervisor HSR

Prof. Dr. Luc Bläser, Institute for Software (IFS), lblaeser@hsr.ch

A.2 Task Setting

The HSR Institute for Software hosts the concurrency lab of Prof. Bläser that is active in research on concurrency and parallelization with a particular focus on the .NET technology. The lab has built a prototype of an efficient on-the-fly static concurrency checker for Visual Studio .NET. The checker currently detects potential race conditions in C# code within the Visual Studio IDE, highlighted and marked even during code writing. In contrast to other existing solutions, this checker performs a real static concurrency analysis (not only simple local bug patterns) as a practical Visual Studio plugin. The implementation is based on the Microsoft Roslyn compiler framework that is integrated by default in the upcoming Visual Studio 2015 edition.

The checker engages a novel conservative static analysis algorithm that considers inter-procedural thread dependency graphs with start/join relations and supports various explicit and implicit thread entrance points, including the .NET Task Parallel Library (TPL). However, the implementation of the checker is currently still strongly limited: it lacks (1) a proper dataflow-sensitive analysis considering control statements, (2) an advanced pointer- and shape-analysis, (3) an incremental analysis across multiple assemblies, (4) the detection of further errors, such as deadlocks, starvation etc., (5) the support of other .NET languages apart from C# as well as analysis of precompiled IL assemblies. Moreover, the current prototype could profit from a proper documentation, certain implementation redesigns, and GUI extensions towards concurreny problem visualizations and explanations.

[1] As defined by: Prof. Dr. Luc Bläser

A.3 Thesis Objectives

This bachelor thesis aims to improve the existing .NET Concurrency Checker prototype in some of the most important, though certainly not all, of the aforementioned limitations. The overall goal lies on extending and altering its design towards a clean support of dataflow-sensitive analysis.
The specific work packages of this thesis are:

1. Study of the existing checker analysis and prototype implementation.
2. An academic description of the underlying analysis algorithm.
3. Redesign towards a general dataflow analysis layer replacing the explicit use of visitors.
4. Improving the accuracy with regard to control statements and method on this layer.
5. Evaluation of the accuracy and impact on performance and memory for sample cases.
6. Optional: further improvements and/or redesigns as described above.

The corresponding goals that should be addressed are:

1. An improved concurrency checker with a sound and complete conservative flow-sensitive inter-procedural analysis. More advanced path- or context-sensitive analysis would be plus but are not required, neither pointer aliasing or shape analysis.
2. A technical report with an adequate description of the overall analysis including the specific enhancements made incl. an evaluation and discussion of related works.

Ideally, an academic paper intended for a research workshop and/or conference could emerge from this work.

Appendix B

Personal Report

At the time of writing I am nearing the end of my studies. I found my bachelor thesis very interesting but also very demanding. Although I have been programming with C# for many years, my thesis gave me an in-depth insight into the programming language and its inner workings.

The first several weeks were sometimes frustrating as the underlying layers are very abstract and results hard to visualise. A notable moment during development was the integration into Visual Studio, where the fruits of my labour became visible - indicating the concurrency errors of a program.

During my thesis I took part in Microsoft's dotnetConf 2015 (March 18-19) with talks on Roslyn, Visual Studio integration as well as new C# 6.0 features. I found the talks helpful in understanding cutting edge best practises advised by the developers behind Roslyn and Visual Studio, as well as understanding some of the more undocumented features.

The notion that my work could eventually result in a product used by many affected me tremendously, motivating me to produce some of my best work to date. It is however a shame that there was not enough time to develop more concurrency configurations (deadlock detection, etc) as part of my bachelor thesis.

As in previous academic work, I could always rely on the excellent care and support of Prof. Dr. Luc Bläser. As a supervisor he always took the time, even outside of the weekly meetings, to answer any unclarities or lingering questions and provided valuable feedback which I learnt a lot from.

To conclude: I feel that I have put the available time to good use and have tried my best to extensively complete the goals defined in the task description. The result of my work is a stable application, which provides a good basis for any continued research.

Appendix C

Declaration of Independent Work

I hereby declare,

- that I have carried out this work myself and unaided, except for persons which are explicitly mentioned in the assignment or have been agreed with the supervisor in writing,
- that I have mentioned all sources used and specified correctly in accordance to current scientific citation,
- that I have used any copyrighted materials (eg pictures) in this work in an unauthorized manner.

Rapperswil, 15. June 2015

Thomas Charrière

Appendix D

Meeting Records

The student and the HSR supervisor generally had weekly meetings to check and discuss the thesis progress. On three occasions meetings had to be cancelled for the following reasons:

- 17.04.2015 due to military service - Thomas Charrière
- 24.04.2015 due to Prof. Dr. Luc Bläser participation at a conference in Karlsruhe, DE
- 08.05.2015 due to military service - Thomas Charrière

D.1 2015-02-20 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- Approval of project plan
- Definition of algorithm
- Admin (project expert, signing of documents)

Protocol

Signing of bachelor thesis agreement between Prof. Dr. Luc Bläser and Thomas Charrière.

Project plan

- Increase Elaboration phase by one week (aim: more detailed prototype);
- Push "Construction I" back by one week;
- "Construction II" will be reduced by one week;
- Milestones reduced to four;

- There will be two Code-Reviews (end of Elaboration and Construction I) as per request.

Short discussion on principles of semantic and syntax analysis.

Defined an example for discussion on proposed algorithm by Prof Dr. Bläser

Open issues for next meeting

- Read through paper on RacerX, to be discussed in next meeting
- Expert for review of bachelor thesis (Prof. Dr. Bläser)
- Root Suffix Visual Studio -> Roslyn?

D.2 2015-02-27 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- RacerX, learnings of
- Visitor vs Walker

Protocol

Discussion of the RacerX paper (belief analysis, meta data usage).

Clarification of advantage SyntaxVistor compared to SyntaxWalker -> single node vs whole tree including trivia (bad).

Further reading as suggested by Prof Dr. Luc Bläser.

- Object race detection (Christoph von Praun and Thomas R. Gross);
- Effective Static Race Detection for Java (Mayur Naik, Alex Aiken, John Whaley);
- Data-flow Analysis (Wikipedia)

Open issues for next meeting

- TFS Build-Server (CI?)

D.3 2015-03-06 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- CFG status
- DFA Architecture (Configurations)

Protocol

Discussion of inter-procedural data flow analysis (reading from last week). Possible Intra-Procedural DFA implementation.

Definition of DFA Configuration Interface (Transfer, Join, NewState)

Further reading sugguested by Prof Dr. Luc Bläser

- Principles of Program Analysis (ISBN 978-3-642-08474-4);

Open issues for next meeting

- Implementation Control-Flow-Graph (CFG)
- Data-Flow Analysis in combination with CFG

D.4 2015-03-13 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- CFG status
- Data-Flow-Analyzer (DFA)
- Thesis expert and revisor expectations

Protocol

After showing Prof Dr. Luc. Bläser a stack based CFG for Do,For,Foreach,If,Switch,While we discussed how to implement jump statments such as continue, break, return.

Discussion on how to solve return edge problem for multiple calls to same procedure inside CFG. Solved by creating a mutlipe CFG (1 per Method,Lambda,Property) and create a reference node.

Data flow analysis optimization for reverse edge lookup - control flow graph requires directed graph to depict flow of execution but data-flow analysis requires predecessors.

Advise on Graph implementation (use msdn example or custom).

Defined scope of what can be expected from reviser and expert during the bachelor thesis.

Open issues for next meeting

- CFG with jump statements
- Multiple CFG in DFA

D.5 2015-03-20 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- CFG status

Protocol

CFG can be exported to the directed graph meta language (dgml) that allows for a visual representation in Visual Studio.

Discussion on the finer points of Foreach (try, finally), how to implement try,catch in CFG. Advised by Prof. Dr. Luc. Bläser to create a stack free CFG to reduce subsequent errors when parsing the syntax tree and to reduce the number of empty blocks (nodes without statement: EndIf, EndWhile, etc).

Open issues for next meeting

- CFG without stack
- CFG Node reduction

D.6 2015-03-27 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- CFG status
- Appointment for mid-term presentation

Protocol

CFG completely rewritten to recursive depth-first visitor pattern by removing stack and replacing walker with visitor. Reduction of empty statements (EndIf, EndWhile) - also in part to recursive architecture.

Prof. Dr. Luc Bläser suggests how to solve thread semantic information problem for threads not assigned to variables (fire and forget).

Open issues for next meeting

- Futher CFG optimization
- DFA

D.7 2015-04-02 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- DFA
- Mid-term presentation

Protocol

Discuss how to reduce the usage of separate in and out states: in states only to be used for statements without predecessors (method, lambda, property start/definition), other in states can be deduced from previous out states of predecessor nodes.

Prof. Dr. Luc. Bläser advises on how to solve intra-procedural data-flow analysis without going through all CFGs when a state changes.

Review of ThreadStartJoinConfiguration for supplying more information to the thread graph in stage 2 of the algorithm.

Open issues for next meeting

- Mid-term presentation

D.8 2015-04-10 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- DFA
- VariableUsage
- Recap Mid-term presentation

Protocol

Prof. Dr. Luc. Bläser advises what to improve for the next presentation at end of semester: comparison of old and new, explain data-flow analysis with simple example, live example in Visual Studio.

Review of custom variable usage (read/write on variable in statement) as Roslyn's implementation provides accesses up to statement in method not actual statement (this caused errors in detection of data-races).

Open issues for next meeting

- DFA complete

D.9 2015-05-01 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- DFA

Protocol

Prof. Dr. Luc Bläser reviews code for stage 2 of thread analysis and advises on methods for intra-thread data-flow analysis.

Next steps are defined in agreement with Prof. Dr. Luc Bläser: Visual Studio integration, code review and documentation.

Open issues for next meeting

- Visual Studio integration

D.10 2015-05-13 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- Code Review
- Visual Studio integration

Protocol

Concurrency checker now integrated into Visual Studio 2015 RC2 with Roslyn RC2. Both I and Prof. Dr. Luc Bläser are very happy with outcome, information for user needs to be polished.

Defined scope of code review (19.05.2015) and prerequisites.

Open issues for next meeting

- Submission for code review

D.11 2015-05-22 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- Code Review
- Visual Studio integration

Protocol

Review of refactorings according to suggestions provided by Prof. Dr. Luc Bläser.

Integration into Visual Studio is more polished offering the user better and more detailed information of data races.

Prof. Dr. Luc Bläser advises on implementation of Fully-Parallel thread analysis.

Open issues for next meeting

- Timeline Documentation

D.12 2015-05-29 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- Documentation

Protocol

Prof. Dr. Luc Bläser advises to split data-flow analysis and thread analysis into separate chapters for better understanding.

Discussion of several points in documentation review such as visualisations and related work.

D.13 2015-06-05 Weekly meeting

Participants

- Prof Dr. Luc Bläser
- Thomas Charrière

Talking points

- Documentation
- Abstract submission

Protocol

Developed an examples for both intra- and inter-data-flow analysis with suggestions from Prof. Dr. Luc. Bläser.

Review of final ToDos in thesis, abstract submission for the HSR and A0-Poster.

Appendix E

Project Plan

Project start: 16.02.2015
Project end: 14.06.2015
Project duration: 17 Weeks

The project phases after which work is carried out comply with the Rational Unified Process (RUP) was developed at the beginning of the thesis, to promote continuous and visible work progress. In agreement with Prof. Dr. Luc Bläser the elaboration phase will take up most of the available time.

RUP	INCEPTION	ELABORATION									
Project week	1	2	3	4	5	6	7	8	9	10	11
	16.02-22.02	23.02-01.03	02.03-08.03	09.03-15.03	16.03-22.03	23.03-29.03	30.03-05.04	06.04-12.04	13.04-19.04	20.04-26.04	27.04-03.05
	CW 8	CW 9	CW 10	CW 11	CW 12	CW 13	CW 14	CW 15	CW 16	CW 17	CW 18
Milestones								M1			

RUP		INTEGRATION			TRANSITION	
Project week	12	13	14	15	16	17
	04.05-10.05	11.05-17.05	18.05-24.05	25.05-31.05	01.06-07.06	08.06-14.06
	CW 19	CW 20	CW 21	CW 22	CW 23	CW 24
Milestones	M2			M3		M4

M1:	Prototype
M2:	Dataflow complete
M3:	Integration complete
M4:	Thesis complete

Figure E.1: Project plan for thesis

E.1 Phases / Iterations

E.1.1 Inception (W1)

- Study of the existing checker analysis and prototype implementation
- Project plan
- Setup of developer tools, build server
- Risk analysis

E.1.2 Elaboration (W2-W12)

- An academic description of the underlying analysis algorithm
- Control Flow Graph generation
- Redesign towards general data-flow analysis layer replacing the explicit use of visitors
- All technical risks eliminated

E.1.3 Integration (W13-W15)

- Visual Studio integration (VSIX)

E.1.4 Transition (W16-W17)

- Final polish
- Thesis
- Admin

E.2 Milestones

E.2.1 M1: Prototype

Deadline: 12.04.2015
Goals:

- Basic CFG generation
- Data Flow Analysis prototype
- Mid-term presentation

Deliverables:

- Mid-term presentation for reviser

E.2.2 M2: Redesign to data-flow analysis layer

Deadline: 10.05.2015
Goals:

- CFG generation for set of examples
- Data Flow Analysis (Stage 1 - 3) complete

Deliverables:

- Code Package for Code-Review

E.2.3 M3: Integration complete

Deadline: 31.05.2015
Goals:

- Integration into Visual Studio 2015 as native add-in
- Concurrency warnings are presented to the user in a understandable form.

Deliverables:

- VSIX Package

E.2.4 M4: Thesis complete

Deadline: 13.06.2015
Goals:

- TBD

Deliverables:

- Thesis (pdf), as well as eprints version
- A0-Poster
- Final Code Package

List of Figures

Bibliography

[1] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, October 2003.

[2] Microsoft Inc. C# Language Specification. `https://msdn.microsoft.com/en-us/library/aa645596.aspx`. [Online; accessed 08-June-2015].

[3] Microsoft Inc. Parallel Programming in the .NET Framework. `https://msdn.microsoft.com/en-us/library/dd460693%28v=vs.110%29.aspx`. [Online; accessed 13-March-2015].

[4] Microsoft Inc. VISX Deplyoment. `https://msdn.microsoft.com/en-us/library/ff363239.aspx`. [Online; accessed 08-June-2015].

[5] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[6] S. Qadeer M. Musuvathi and T. Ball. Chess: A systematic testing tool for concurrent software. *Microsoft Research Technical Report MSR-TR-2007-149*.

[7] Neil McAllister. Microsoft's Roslyn: Reinventing the compiler as we know it. `http://www.infoworld.com/article/2621132/microsoft-net/microsoft-s-roslyn--reinventing-the-compiler-as-we-know-it.html`. [Online; accessed 01-June-2015].

[8] John Mellor-Crummey. 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, pages 24–33, New York, NY, USA, 1991. ACM.

[9] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. *SIGPLAN Not.*, 41(6):308–319, June 2006.

[10] Boby George Rahul V. Patil. Tools and techniques to identify concurrency issues. *MSDN Magazine*.

[11] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *SIGOPS Oper. Syst. Rev.*, 31(5):27–37, October 1997.

[12] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. *SIGPLAN Not.*, 38(5):115–128, May 2003.

[13] B. Wikipedians. *Compiler Construction*. PediaPress.

Glossary

.NET

.NET (pronounced dot net) Framework is a software framework developed by Microsoft that runs primarily on Microsoft Windows. Programs written for .NET Framework execute in a software environment, known as Common Language Runtime (CLR). 7, 32, 51, 72

C#

C# (pronounced as see sharp) is a multi-paradigm programming language encompassing strong typing, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines. It is one of the programming languages designed for the Common Language Infrastructure. 6, 7, 32, 34, 54, 72

CFG

control flow graph. 12, 14, 15, 17, 19–21, 32–34, 38, 40, 42, 48–50, 69, 72

deadlock

is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does. 6, 72

IDE

integrated development environment. 72

IFS

Institute for Software, HSR Hochschule für Technik. 9, 11, 43, 72

NuGet

is the package manager for the Microsoft development platform. 42, 72

PLINQ

Parallel Language-Integrated Query. 7, 72

race condition

is a situation in which threads are dependent on the sequence or timing of other uncontrollable events. Low-level data races become manifest in concurrent synchronised accesses on the same memory location, involving at least one write access. 6, 72

starvation

is a situation in which a thread is perpetually denied necessary resources to proceed its work. 6, 72

TPL

Task Parallel Library. 7, 9, 50, 72

VSIX

Visual Studio Integration Package. 11, 72