HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

ins
INSTITUTE FOR
NETWORKED SOLUTIONS

C# Checker Frontend
Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2019/2020

Autoren: Simon Styger, Daniel Bucher

Betreuer: Prof. Dr. Luc Bläser

Projektpartner: Institute for Networked Solutions

Legal Disclaimer
Alle in dieser Arbeit genannten Marken-, Produkt- und Servicenamen stehen im Besitz des jeweiligen Eigentümers. Sie werden innerhalb dieser Arbeit rein zur Information angegeben und werden somit ohne Kennzeichnung aufgeführt. Die Verfasser der Arbeit stehen in keinerlei Verbindung mit den Markeninhaber.

1 Abstract

Prof. Dr. Luc Bläser hat den HSR Parallel Checker entwickelt, mit dem Source Code statisch auf Nebenläufigkeitsfehler untersucht werden kann. Der Checker stand bisher nur als Visual Studio Plugin zur Verfügung. Im Rahmen dieser Arbeit ist eine Desktop Applikation realisiert worden, die komfortablere und umfangreichere Analysen ermöglicht.

Die Benutzerfreundlichkeit und einfache Bedienbarkeit standen bei der Entwicklung im Zentrum. Personen ohne tiefere Informatik-Kenntnisse sollte es ermöglicht werden, Programmcode auf Nebenläufigkeitsfehler zu untersuchen. Zudem sollten mehrere Parameter eingestellt werden können, um den Code noch tiefer auf Fehler zu analysieren. Im Endresultat werden dem User Fehler grafisch angezeigt und es stehen mehrere Reportingmöglichkeiten wie z.B. Word zur Verfügung.

Zusätzlich zur Desktop Applikation ist eine Konsolen-Applikation entstanden. Diese bietet weitere Anwendungsmöglichkeiten wie z.B. die Integration in einer CI/CD Pipeline. So könnten DevOps-Teams Source Code in Form eines Nightly Builds überprüfen.

Die Applikation wurde anhand populären C# Projekten getestet. Aufgrund der Resultate kann eine ungefähre Empfehlung für die Analyseeinstellungen gegeben werden.

Inhaltsverzeichnis

1 Abstract **2**

2 Management Summary **5**
2.1 Ausgangslage . . . 5
2.2 Umsetzung . . . 5
2.3 Auswertung . . . 5

3 Einleitung und Übersicht **6**
3.1 Ausgangslage . . . 6
3.2 Eigenschaften des HSR Parallel Checkers . . . 6
3.3 Gängige Nebenläufigkeitsfehler . . . 6
3.3.1 Data Race . . . 6
3.3.2 Race Condition . . . 6
3.3.3 Beispiele für Race Conditions und Data Races . . . 6
3.3.4 Deadlock . . . 7

4 Analyse von ähnlichen Programmen **9**
4.1 Kriterien . . . 9
4.1.1 Funktionalität . . . 9
4.1.2 Berichte . . . 9
4.1.3 Usability . . . 9
4.2 Die untersuchten Tools . . . 9
4.2.1 Polyspace . . . 9
4.2.2 ThreadSanitizer . . . 9
4.2.3 Inspector XE . . . 9
4.2.4 FindBugs . . . 9
4.2.5 VSTools . . . 10
4.2.6 PathFinder . . . 10
4.3 Resultate . . . 10
4.3.1 Funktionalität . . . 10
4.3.2 Berichte . . . 12
4.3.3 Usability . . . 12
4.4 Gewonnene Erkenntnisse . . . 15

5 Anforderungen **16**
5.1 Ziel des C# Parallel Checker Frontends . . . 16
5.1.1 Zielgruppe . . . 16
5.2 Funktionale Anforderungen aus der Aufgabenstellung . . . 16
5.2.1 Analyseprojekt . . . 16
5.2.2 Navigation zur Fehler-Stelle . . . 16
5.2.3 Generierung von grafischen Report-Files . . . 16
5.3 Funktionale Anforderungen aus den Erkenntnissen . . . 16
5.3.1 Funktionalität . . . 16
5.3.2 Berichte . . . 17
5.3.3 Repetierbarkeit . . . 17
5.3.4 CLI-Optionen . . . 17
5.4 Nicht-funktionale Anforderungen . . . 17
5.4.1 Statusinformationen . . . 17
5.4.2 Übersichtlichkeit . . . 17
5.4.3 Plattformunabhängigkeit . . . 17

6 Umsetzung **18**
6.1 Architekturentscheidungen . . . 18
6.1.1 Grundsätzliche Überlegungen . . . 18
6.1.2 Prototyp mit Electron . . . 18
6.1.3 Prototyp mit WPF . . . 19
6.1.4 Umgesetzte Architektur . . . 20
6.2 UI Design und umgesetzte Features . . . 21
6.2.1 Startview . . . 22
6.2.2 Durchführungs-View . . . 23
6.2.3 Resultatview . . . 24
6.2.4 Graphen . . . 25
6.2.5 Reports . . . 25
6.3 Analyseablauf . . . 26
6.4 Concurrency Design . . . 27

7 Auswertung **28**
7.1 Auswahl an Projekten . . . 28
7.2 Setup . . . 28
7.3 Ergebnisse . . . 28
7.3.1 MSBuild Probleme . . . 29
7.4 Mögliche Lösungsansätze . . . 29
7.4.1 Parallel Checker als nuGet-Paket . . . 30
7.4.2 SonarQube Ansatz . . . 30
7.5 Tests mit verschiedenen Parameter . . . 30
7.5.1 Geändertes Setup . . . 30
7.5.2 Idealer Faktor . . . 31
7.5.3 Akkumulierte Werte . . . 31
7.5.4 Asynchrone Werte . . . 32
7.5.5 Asynchrone Werte, 2. Versuch . . . 32
7.5.6 Erkenntnisse . . . 33

8 Schlussfolgerungen **34**
8.1 Ausblick . . . 34
8.1.1 Betriebsmittel-Graph . . . 34
8.1.2 Mehrere Projekte gleichzeitig parameterisierbar . . . 34
8.1.3 Selbstständige Optimierung . . . 34
8.1.4 Cloud-Service . . . 34

Glossar **35**

Abbildungsverzeichnis **36**

Tabellenverzeichnis **36**

Codebeispiele **36**

Literaturverzeichnis **36**

Anhang **39**
Testresultate SignalR . . . 39
Testresultate Orleans . . . 41
Testresultate NLog . . . 43
Testresultate MSBuild . . . 45
Fehlermeldung SonarQube Roslyn SDK . . . 47

2 Management Summary

2.1 Ausgangslage

Nebenläufigkeitsfehler sind tückisch und können auch dem erfahrensten Fachmann passieren. Zudem sind sie durch ihren Nicht-Determinismus nur schwierig testbar.

Der HSR Parallel Checker von Prof. Dr. Luc Bläser analysiert C# Code auf solche Fehler mit Hilfe eines neuartigen Verfahrens. Bisher stand dieser Checker nur als Visual Studio Plugin zur Verfügung.

2.2 Umsetzung

In dieser Arbeit wurde eine Windows Desktop Applikation entwickelt, die den bestehenden Checker Kern einbindet. Mit Hilfe der Applikation lassen sich Projekte bequem per UI auswählen und umfangreichere Analysen werden durch einstellbare Parameter ermöglicht.

Die Resultate werden mit Hilfe von Graphen dargestellt, was das Verstehen der Fehler vereinfacht. Weiter kann der Anwender die Resultate als Word oder JSON Datei exportieren.

Bei der Entwicklung der Applikation haben wir auf einfache Bedienbarkeit geachtet, damit auch Personen ohne tiefere Informatik-Kenntnisse Analysen durchführen können.

Zusätzlich zur Desktop-Applikation wurde eine Konsolenapplikation entwickelt. Diese bietet weitere Anwendungsbereiche wie eine automatisierte Einbindung in den Entwicklungsprozess (z.B. in Form von regelmässigen Analysen in Nightly Builds).

2.3 Auswertung

Beim Testen unserer Applikation mit populären Open Source Projekten sind wir leider auf einige Hindernisse gestossen. Zusammenfassend lässt sich sagen, dass Projekte mit älteren Framework Versionen Komplikationen verursachen. Dies liegt an grundlegenden Problemen im MSBuild System.

3 Einleitung und Übersicht

3.1 Ausgangslage

Prof. Dr. Luc Bläser hat an der Hochschule für Technik, Rapperswil ein neuartiges Verfahren für die Entdeckung von Nebenläufigkeitsfehler entwickelt. Mittels beschränkter randomisierter fast-konkreter Interpretation werden C# Programme statisch analysiert. [1]
Bisher steht dieser Checker erst als Plugin für die Entwicklungsumgebung Visual Studio zur Verfügung. [2]
Ziel dieser Arbeit ist es den Checker in eine Standalone Applikation einzubinden, um das Untersuchen von Quellcode komfortabler und umfangreicher zu gestalten. Die Analyse kann so mit längeren Zeiten und verschiedenen Parameter erfolgen. Weiter sollen diverse Reports erstellt werden können. Die genauen Anforderungen werden in Abschnitt 5 erläutert.

3.2 Eigenschaften des HSR Parallel Checkers

Da der Checker direkt in einer Entwicklungsumgebung läuft und der Code somit unvollständig oder fehlerhaft sein kann, ist eine dynamische Analyse nicht möglich. Der Checker wurde daher als statischer Analyzer implementiert. Weiter wurde auf schnelle Feedbacks auch bei grossen Projekten Wert gelegt, damit der Programmierer direkt eine Rückmeldung sieht. Zwischen Vollständigkeit (Alle Fehler werden entdeckt, keine False Negatives) und Präzision (nur echte Fehler werden angezeigt, keine False Positives) musste ein Kompromiss gefunden werden. Ein statischer Checker kann nicht beides gleichzeitig sein, da dies nicht entscheidbar ist (analog dem Halteproblem). Um dem Programmierer möglichst viele echte Fehler anzuzeigen, hat man sich für Präzision entschieden und nimmt somit in Kauf, dass die Fehlererkennung nicht vollständig ist. [1]

3.3 Gängige Nebenläufigkeitsfehler

Der HSR Parallel Checker fokussiert sich auf das Auffinden von Nebenläufigkeitsfehlern auf Sprachebene. Somit werden Data Races und Deadlocks gefunden. Weiter werden Race Conditions in Zusammenhang mit Aufrufen der Collection API erkannt. Für Race Conditions im allgemeinen sowie für Starvation, Livelocks etc. müsste man eine semantische Analyse durchführen, weshalb darauf verzichtet wurde. [1]

3.3.1 Data Race

Bei einem Data Race handelt es sich um einen formalen Fehler, da Programmiersprachen kein Verhalten in diesem Fall spezifizieren. Data Races treten auf, wenn zwei nebenläufige, nicht synchronisierte Zugriffe auf dieselbe Speicherstelle (Variable oder Array-Element in Java und C#) erfolgen. Mindestens ein Zugriff ist schreibend. [1] [3]

3.3.2 Race Condition

Bei einer Race Condition greifen mehrere Threads ohne ausreichende Synchronisation auf gemeinsame Ressourcen zu. Dies führt je nach Thread Verzahnung und zeitlicher Ausführung zu falschen Resultaten oder unerwartetem Verhalten. Race Conditions treten häufig in Zusammenhang mit Data Races auf. [3]

Wenn Race Conditions bei der Verwendung von Collections enstehen (z.B. unsychronisierte, nebenläufige Ausführung von `collection.Add(..)` und `collection.GetEnumerator()`) , wird dies vom Checker als `Thread-unsafe call` ausgewiesen.

3.3.3 Beispiele für Race Conditions und Data Races

Im unten aufgeführten Beispiel sind die Zugriffe auf die Variable `balance` nicht synchronisiert, was zu mehreren Data Races führt. Da möglicherweise mehr Geld überwiesen werden kann, als Guthaben vorhanden ist, können auch Race Conditions auftreten. [4]

```
class BankAccount  {
    private int balance = 1000;

    public void Transfer(BankAccount other, int amount)  {
        if (balance >= amount) {
            balance -= amount;
            other.balance += amount;
        }
    }
}

new Thread(() => accountA.Transfer(accountB, 100)).Start();
new Thread(() => accountB.Transfer(accountC, 50)).Start();
```

Beispiel 1: Race Condition mit Data Races [4]

Die `balance` Zugriffe sind nun synchronisiert, was die Data Races eliminiert. Da die Überprüfung und Änderungen der Kontostände nicht atomar sind, ist immer noch eine Race Condition vorhanden. Somit kann wieder Geld ohne ausreichendes Guthaben transferiert werden. [4] Solch ein Fehler wird vom Checker, wie oben erwähnt, nicht erkannt.

```
class BankAccount  {
    private int balance = 1000;

    public void Transfer(BankAccount other, int amount) {
        if (Volatile.Read(ref balance) >= amount) {
            Interlocked.Add(ref balance, -amount);
            Interlocked.Add(ref other.balance, amount);
        }
    }
}

new Thread(() => accountA.Transfer(accountB, 100)).Start();
new Thread(() => accountA.Transfer(accountC, 50)).Start();
```

Beispiel 2: Race Condition ohne Data Races [4]

3.3.4 Deadlock

Deadlocks enstehen, wenn sich mehrere Threads gegenseitig in zyklischen Warteabhängigkeiten blockieren.[1]

Im unten aufgeführten Beispiel werden zwei nebenläufige Banküberweisungen getätigt. Thread 1 und Thread 2 halten die Locks für `accountA` und `accountB`. Nun möchten beide jeweils das andere Konto sperren, was zu einem Deadlock führt. Dies kann auch in einem Betriebsmittelgraph dargestellt werden. [5]

Solche Arten von Fehler werden vom Checker erkannt.

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

    public void Transfer(BankAccount other, int amount)  {
        lock (sync) {
            lock (other.sync) {
                if(balance >= amount) {
                    balance -= amount;
                    other.balance += amount;
                }
            }
        }
    }
}

new Thread(() => accountA.Transfer(accountB, 100)).Start();//1
new Thread(() => accountB.Transfer(accountA, 50)).Start();//2
```

Beispiel 3: Deadlock [1]

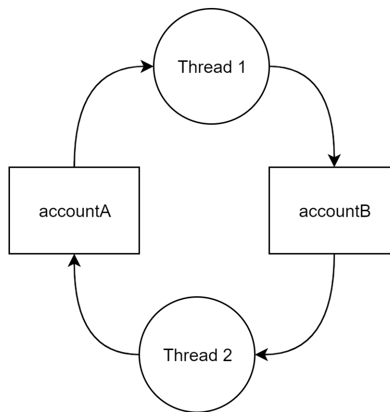

Abbildung 1: Deadlock als Graph [3]

4 Analyse von ähnlichen Programmen

Prof. Dr. Bläser hat in seinem Paper bereits mehrere Analysetools miteinander verglichen. [1] Um den Funktionsumfang unseres Projektes detaillierter festzulegen, haben wir einige davon ausgewählt und einen eigenen Vergleich erstellt. Dadurch soll einerseits ein Überblick über die Möglichkeiten der Tools und andererseits Design-Ideen gewonnen werden.

4.1 Kriterien

4.1.1 Funktionalität

Bei der Überprüfung der Funktionalität wurden die folgenden Punkte berücksichtigt:

- **Genereller Umfang:** Welche Funktionen stehen dem User zur Verfügung?
- **Repetierbarkeit:** Lassen sich bestimmte Analysen von Projekten reproduzieren? Kann man einzelne Durchführungen wiederholen? Diese Funktionalität ist wichtig, da sich Code laufend ändert und man diesen beispielsweise während der Entwicklung mehrmals mit gleichbleibenden Parameter testen möchte. Durch Änderungen am Code können neue Nebenläufigkeitsfehler entstehen oder alte behoben werden.

4.1.2 Berichte

Bei unserem Projekt wird Wert auf verschiedene Report Files gelegt. Deshalb ist dies ein eigenes Kriterium.

4.1.3 Usability

- **Einfachheit:** Wie gestaltet sich die Nutzung des Analyse-Tools? Lässt es sich einfach bedienen und versteht der Nutzer, was er machen muss, um sein Ziel zu erreichen?
- **Design:** Wie ist die Benutzeroberfläche designmässig gestaltet?

4.2 Die untersuchten Tools

4.2.1 Polyspace

Polyspace ist eine Produktfamilie von Code-Analyse-Tools. Für unsere Analyse wurde der "Polyspace Code Prover" verwendet. Der Code Prover führt eine semantische Analyse von C und C++-Quellcode durch und überprüft mittels einer abstrakten Interpretation das Verhalten der Software. [6]
Leider wurde eine Anfrage zum Erhalt der Testsoftware nicht rechtzeitig behandelt, wodurch unsere Analyse bei diesem Tools nur sehr oberflächlich durchgeführt werden konnte. Dennoch konnten einige Erkenntnisse gewonnen werden.

4.2.2 ThreadSanitizer

ThreadSanitizer ist Teil des LLVM Projektes und zur Zeit noch im Beta Stadium. Via Flag lässt sich das Tool in den Kompilier-Vorgang von C++ und C-Programmen einbinden. Die gefundenen Issues werden nach der Kompilierung aufgelistet. [7]

4.2.3 Inspector XE

Inspector XE ist ein Tool von Intel. Mit Inspector XE lassen sich Fehler in Multi-Threading-Applikationen mittels dynamischer Analyse finden. Das Tool lässt sich sowohl auf Windows wie auch auf Linux ausgeführen. Unterstützt werden Analysen für Programm-Code in C und C++. [8]

4.2.4 FindBugs

FindBugs ist ein statischer Analyser, der von der Universität Maryland entwickelt wird. Mittels Pattern Matching werden Fehler in Java Code entdeckt. [9]
Anzumerken ist, dass der Support für FindBugs mittlerweile eingestellt wurde. Das SpotBugs-Projekt führt die Entwicklung unter neuem Namen weiter. [10]

4.2.5 VSTools

Die Tools für das Visual Studio bieten einige Möglichkeiten um Code zu analysieren. Unterstützt werden dabei die Sprachen C/C++ und C#. [11]

4.2.6 PathFinder

Java PathFinder ist ein Framework zur Überprüfung von Java Bytecode. Es bietet detaillierte Konfigurationsmöglichkeiten. Mit Hilfe von JPF lassen sich unter anderem Fehler wie Deadlocks oder unbehandelte Exceptions entdecken. [12]

4.3 Resultate

4.3.1 Funktionalität

Polyspace
Der Polyspace Code Prover bietet folgende Funktionalitäten:

- Die Analyse von Code kann man sowohl im GUI, als auch in der Konsole (mittels Einbindung in eigenen Skripten) und in der Eclipse-IDE durchführen.
- Die Resultate der Analyse können entweder über das GUI oder direkt in der IDE mittels Plugin angezeigt werden.

Nebenläufigkeits-Fehler lassen sich mit dem Code Prover nicht entdecken. Der Umfang der auffindbaren Fehler beschränkt sich auf bewiesene Laufzeitfehler, potenziell fehlerhafte Operationen wie Overflows oder Divison-by-Zero und unerreichbarer Code. [13]

Mit der Code Prover Server Applikation haben Software-Firmen die Möglichkeit die Analyse des Codes in die Build-Pipeline zu integrieren. Die daraus entstehenden Resultate können dann über ein Web-Dashboard visualisiert werden.

Sowohl mit dem Server-Tool als auch mit den Dekstop-Tools lassen sich einzelne Analysen wiederholen. Die dafür benötigten Konfigurationen lassen sich als Analyse-Projekt abspeichern. Die dadurch mögliche Repetierbarkeit erlaubt es mehrere Analyse-Resultate zu gewinnen und miteinander zu vergleichen.

ThreadSanitizer
Mittels Flag lässt sich der Analyzer direkt in den Kompilierungsvorgang einbinden. Diese Einbindung kann mit einzelnen Parametern zusätzlich spezifiziert werden. Die Möglichkeit eine Art Präzisions-Faktor anzugeben ist dabei besonders nennenswert, da man damit die Möglichkeit hat auf Kosten einer höheren Laufzeit mehr Nebenläufigkeitsfehler zu entdecken.

Eine Desktop-Applikation mit GUI wird vom ThreadSanitizer nicht geboten.

Bezüglich Repetierbarkeit stellten wir fest, dass man durch eigens erstellte Skripts die Möglichkeit hat, die Kompilierung mitsamt Analyse beliebig oft zu wiederholen, indem man die Kompilierung jeweils mit den selben Parameter aufruft. Diese vorgenommene Konfigurationen können allerdings nicht als Analyse-Projekt gespeichert werden. [7]

Inspector XE
Der Intel Inspector kann über drei Arten verwendet werden. Er lässt sich als Plugin in die Visual Studio IDE einbinden. Zudem bietet er eine CLI sowie ein eigenständiges GUI.
Neben Data Races und Deadlocks lassen sich mit dem Intel Inspector auch Memory Leaks und ungültige Zugriffe (Invalid Accesses) finden.

Der Intel Inspector bietet die Möglichkeit, die Analyse auf drei unterschiedlichen Detailstufen durchzuführen:

- **Deadlocks erkennen**: Die Analyse erkennt lediglich, ob sich im Code Deadlocks befinden oder nicht. Wo sie sich befinden, wird bei dieser Detailstufe nicht angegeben.

- **Data Races erkennen**: Die Analyse erkennt neben Deadlocks auch Data Races. Wo sich die Fehler im Code befinden, wird mit dieser Detailstufe nicht festgestellt.
- **Nebenläufigkeitsfehler lokalisieren**: Neben dem Erkennen von Deadlocks und Data Races werden sie mit dieser Detailstufe auch im Code lokalisiert und angezeigt.

Der Grund für diese drei Levels ist gemäss Hersteller der massive Memory- und Laufzeit-Overhead der exponentiell zur Level-Zahl ansteigt. [14]
Mit Intel Inspector kann man die verursachende Code-Zeile mit einem einfachen Klick auf den Fehler in einem Editor öffnen. So besteht die Möglichkeit, die Ursache dieses Nebenläufigkeitsfehlers gleich zu beheben.

Die Benutzer von Inspector XE können die Konfigurationen für Analysen als Analyse-Projekt speichern. Damit sind sehr spezifische Analysen sowohl aus dem GUI wie auch aus dem CLI aufrufbar.

FindBugs
FindBugs bietet neben einer Desktop-Applikation mit einem GUI auch eine CLI zur Fehlersuche. Diese Fehlersuche beschränkt sich jedoch auf einfache Fehlermuster und kann keine Nebenläufigkeitsfehler erkennen. Die Funktionalität von FindBugs lässt sich, wenn gewünscht und nötig, mit eigenen Detektoren erweitern.

FindBugs bietet sowohl für das GUI als auch die CLI eine sehr umfangreiche Funktionalität und erlaubt sehr viele Optionen mit denen man das Verhalten während der Analyse genauer spezifizieren kann. Beispielsweise lässt sich mit dem Flag `-effort:[min/max]` die Genauigkeit der Analyse variieren. Weiter besteht mit den Flags `-low`, `-medium` und `-high` die Möglichkeit, sich nur Fehler eines bestimmten Schweregrads anzeigen zu lassen. Besonders interessant ist auch das Flag `-relaxed`, welches die Heuristiken zur Vermeidung von False Positives unterdrückt.

Mit dem GUI kann man zudem ein Analyse-Projekt anlegen. Dort können die gewünschten Dateien angegeben werden, welche analysiert werden sollen.

VSTools
Im Visual Studio hat man einige Möglichkeiten die Code Qualität zu analysieren. Die Funktionalität beläuft sich dabei aber lediglich auf bekannte Fehlermuster und man kann keine Nebenläufigkeitsfehler erkennen lassen. Da diese Tools direkt in der Visual Studio-IDE genutzt werden, ist ihre Verwendung recht simpel und man findet sich schnell zurecht.
Die Code-Analysen lassen sich mit sehr vielen Optionen konfigurieren, wodurch man spezialisierte Analysen durchführen und wiederholen kann. Die Konfigurationen werden dabei intern im Visual Studio gespeichert.

Dadurch dass die Tools bereits in der IDE integriert sind, bieten sie eine sehr vorteilhafte Funktionalität. Die gefundenen Fehler können so nämlich direkt behoben werden. Die VS-Tools bieten hierfür sogenannte Quick-Fixes an. Das sind Aktionen die dem Entwickler aufgrund des erkannten Fehlermusters vorgeschlagen werden. Bestätigt der Entwickler einen Quick-Fix, wird die Änderung direkt im Quellcode vorgenommen.

PathFinder
Mittels Veränderung des Inputs versucht der PathFinder das Verhalten des analysierten Programms zu überprüfen. So können unter anderem auch Data Races und Deadlocks entdeckt werden. Zusätzlich kann der PathFinder ein Model Checking durchführen. Das bedeutet es werden alle möglichen Programmzustände abgearbeitet und untersucht, bis entweder keine weiteren Zustände mehr möglich sind, oder ein Fehler entdeckt wurde.
Den Java PathFinder kann man direkt in der Konsole, sowie als IDE-Plugin für die Umgebungen NetBeans und Eclipse nutzen. Zusätzlich kann der Java PathFinder sogar direkt aus einer eigenen Java-Applikation genutzt werden. Das bietet unzählige Erweiterungsmöglichkeiten, die man auf Basis des PathFinders realisieren kann.

Der Java PathFinder bietet sehr detaillierte Konfigurationsmöglichkeiten, die gespeichert werden können. Somit lassen sich Analysen auch sehr gut repetieren.

4.3.2 Berichte

Polyspace
Wie bereits erwähnt, bietet der Code Prover von Polyspace die Möglichkeit die Resultate einer Code Analyse zu exportieren. Die generierbaren Berichte zeigen übersichtlich, in welchen Dateien welche Art von Fehlern gefunden wurde.
Das Web-Dashboard wurde bereits unter dem Punkt Funktionalität erwähnt. Durch dieses Dashboard lassen sich viele Analysen über eine grössere Zeitspanne vergleichen. Dadurch lässt sich eine Reihe an Berichten aufbauen, welche die Verbesserung der Code Qualität über eine Zeitspanne aufzeigen können.

ThreadSanitizer
Leider hat man beim ThreadSanitizer nicht die Möglichkeit solche Berichte zu erstellen, wie wir sie beim Parallel Checker bieten wollen. Die gefundenen Issues werden nach der Durchführung lediglich auf der Konsole ausgegeben. Man hat dadurch die Möglichkeit, die Ausgabe in ein File zu schreiben, erhält davon aber keinen übersichtlichen Bericht.

Inspector XE
Leider hat man Intel Inspector nicht die Möglichkeit, die gefundenen Issues in einem Bericht zu exportieren. Somit lässt sich auch keine Übersicht über Analysen erstellen, welche mehrere Male durchgeführt wurden.

FindBugs
Man hat mit FindBugs zwar die Möglichkeit, die Resultate einer Analyse zu exportieren, dies allerdings nur in eine XML-Datei. Um die Issues in einem schönen Bericht darzustellen oder in ein Dashboard einzubinden, müsste der Benutzer selbst einen Weg finden, wie er das erreichen kann.

VSTools
Die Visual Studio Tools bieten keine Möglichkeit, die Resultate einer Analyse zu exportieren oder in einen Bericht einfliessen zu lassen. Das liegt wahrscheinlich daran, dass die Tools vor allem für die Verwendung während der Entwicklung innerhalb der IDE gedacht sind.

PathFinder
Mit dem PathFinder lassen sich zwar Reports erstellen, diese sind jedoch nicht sehr umfangreich. Man hat nur die Möglichkeit, die gefundenen Issues in eine Text oder XML Datei zu exportieren.

4.3.3 Usability

Polyspace
Das Design des Code Provers ist stark an die Eclipse IDE gebunden. Dadurch ist es sehr schwer, die gesamte Funktionalität in teilweise sehr kleinen Unterfenstern darzustellen. Der Code Prover ist dennoch einfach und verständlich aufgebaut, wodurch eine gute User Experience erreicht wird. Das Web-Dashboard ist durchaus ansprechend gestaltet und bietet ebenfalls eine gute Usability.

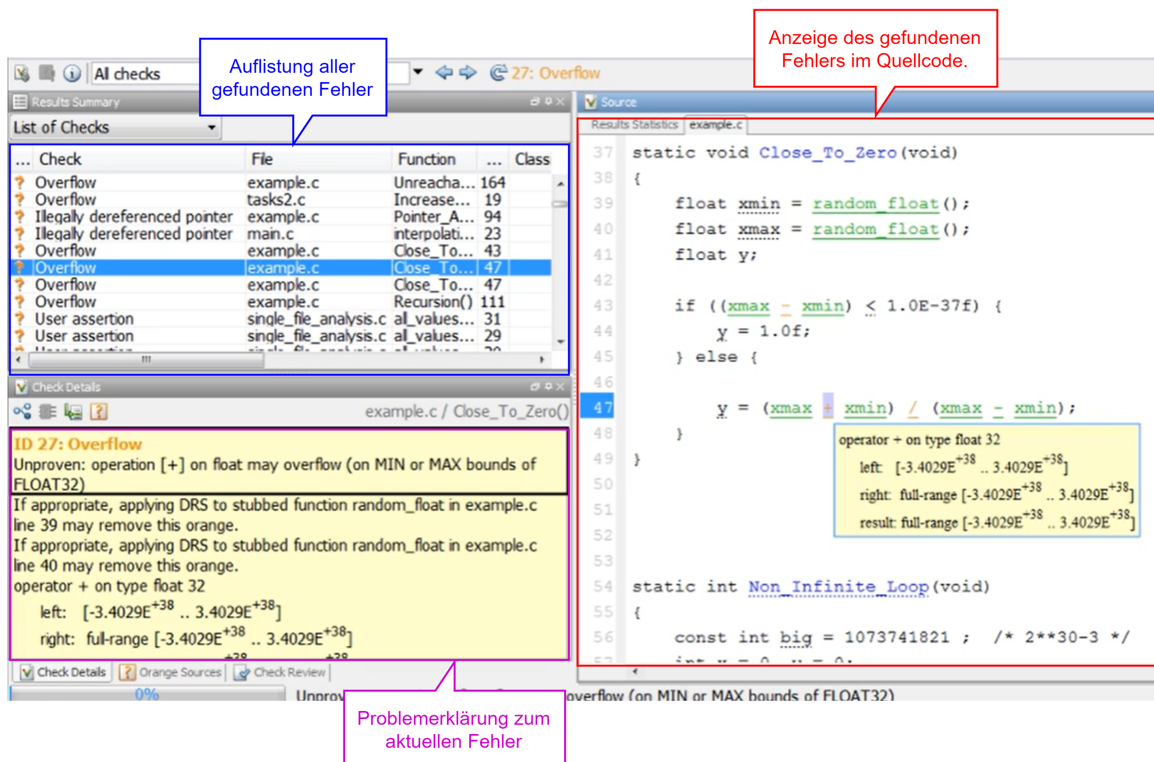

Abbildung 2: Screenshot vom GUI des Polyspace Code Provers. [15]

ThreadSanitizer
Da der ThreadSanitizer lediglich in der Konsole ausgeführt werden kann, lässt sich das Design dieses Tools nicht wirklich analysieren. Durch die Einschränkung auf die Konsole, spricht das Tool grundsätzlich Software-Entwickler an. Somit entsteht dennoch eine hohe Usability, da sich das Zielpublikum gut mit solchen Tools auskennen sollte.

Inspector XE
Das Design des Intel Inspectors ist sehr überzeugend. Beim IDE Plugin wurde eine sehr übersichtliche Ansicht aufgebaut, welche alle Issues in einer Liste und einen einzelnen, ausgewählten Issue im Detail anzeigen kann. Zudem lassen sich die angezeigten Issues filtern. Inspector XE bietet somit eine hohe Usability.

FindBugs
Das Design des GUI ist sehr übersichtlich. Unserer Meinung nach ist es allerdings nicht sehr intuitiv und bietet eine eher geringe User Experience. Man hat zwar die Möglichkeit, direkt zur entsprechenden Stelle im Code zu navigieren, die Fehler werden jedoch nicht zusammenfassend aufgelistet. Wie man im folgenden Bild erkennen kann, wird einfach die package-Struktur angezeigt. Der Benutzer muss dann durch die Struktur navigieren, um zur Detailansicht zu gelangen. Positiv sind die Buttons zur Navigation, sowie eine dargestellte Zusammenfassung des ausgewählten Fehlers.

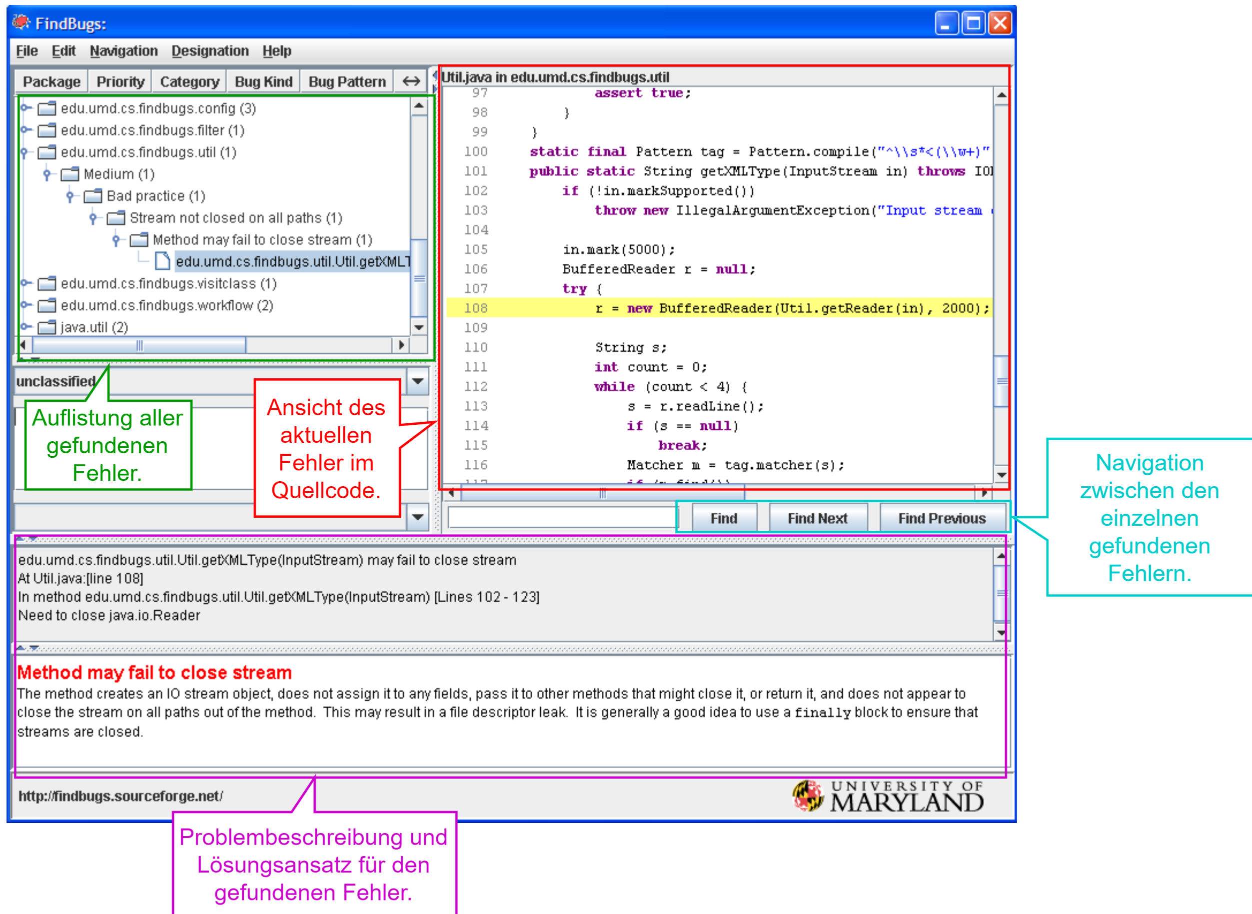

Abbildung 3: GUI von FindBugs zeigt die gefundenen Fehler. [16]

VSTools

Obwohl die Tools direkt im Visual Studio integriert sind, ist das Design der Tools sehr überzeugend. Es bietet eine übersichtliche Liste von gefundenen Fehlern, welche auch nach dem Schweregrad gefiltert werden können. Die Verwendung des IDE-Plugins gestaltet sich sehr einfach, wodurch eine hohe Usability geboten wird.

PathFinder

Der PathFinder lässt sich in unterschiedlichen Java-Entwicklungsumgebungen nutzen. Hat dadurch aber keinen grossen Einfluss auf die Benutzeroberfläche. Trotz den Einschränkungen wird eine gute Usability geboten. Der Benutzer kennt sich mit der Navigation in der IDE aus und findet sich darum auch schnell in der Erweiterung des PathFinders zurecht. Die CLI des PathFinders entspricht zudem den Erwartungen eines solchen Tools.

	Repetierbarkeit	Umfang	Berichte	Usability
PolySpace	✓	∼	✓	✓
ThreadSanitizer	∼	∼	✗	✗
Inspector XE	✓	✓	✗	✓
FindBugs	✓	∼	∼	∼
Visual Studio-Tools	✓	✓	✗	∼
PathFinder	✓	∼	∼	∼

Tabelle 1: Übersicht der Analyse (✗= nicht vorhanden, ∼ = vorhanden, ✓= gut umgesetzt)

4.4 Gewonnene Erkenntnisse

Im folgenden werden die wichtigsten Erkenntnisse aus der Analyse nochmals zusammengefasst. Sie sind unserer Meinung nach deshalb wichtig, weil sie bereits als Inspiration für unsere Umsetzung dienen. Wie genau die Erkenntnisse verwendet und umgesetzt werden sollen, wird im Abschnitt 5 genauer erläutert.

- **CLI-Optionen**: Der ThreadSanitizer und der Intel Inspector bieten sehr viele Optionen um die Analysen zu konfigurieren. So wird eine breite Funktionalität angeboten. Entscheidend dabei ist, dass man bei beiden Tools angeben kann, wie detailliert die Suche durchgeführt werden soll. Beim ThreadSanitizer hat man die Möglichkeit einen Präzisions-Faktor anzugeben, beim Intel Inspector kann man das Detaillierungslevel definieren.

- **Repetierbarkeit**: Mit Code Prover von Polyspace und Intel Inspector hat man die Möglichkeit Konfigurationen als Analyse-Projekt abzuspeichern. Dies ermöglicht es, Analysen mit gleichen Parametern zu wiederholen.

- **Berichte**: Der Code Prover von Polyspace hat diese Funktionalität unserer Meinung nach am besten umgesetzt. Die Möglichkeit Berichte sowohl als PDF-Datei als auch in ein Web-Dashboard zu exportieren macht durchaus Sinn. Uns gefällt die Idee für ein Web-Dashboard zudem sehr gut und wird im Abschnitt 5 nochmals vertiefter beschrieben.

- **Design**: Bezüglich Design haben wir keine handfesten Erkenntnisse, die wir als Entscheidungsgrundlage für unsere Anforderungen nutzen könne. Wir haben aber erkannt, dass es wichtig ist, die gefundenen Issues übersichtlich und intuitiv darzustellen.

5 Anforderungen

Dieser Abschnitt beschreibt die gestellten Anforderungen genauer. Zusätzlich haben wir aus den Erkenntnissen im Abschnitt 4 weitere Anforderungen definiert, die hier besprochen werden.

5.1 Ziel des C# Parallel Checker Frontends

Das Frontend des C# Parallel Checkers soll die Verwendung des HSR Parallel Checkers vereinfachen. Dem Benutzer soll eine Applikation geboten werden, mit welcher er die gesamte Funktionalität des Parallel Checkers einfach und verständlich nutzen kann. Er muss dafür die Analyse-Parameter anpassen, die Analyse mitverfolgen und die gewonnen Resultate in einer übersichtlichen Art und Weise einsehen können.

5.1.1 Zielgruppe

Mit der Desktop Applikation sprechen wir vor allem Personen ohne tiefere Informatikkenntisse an. Dies könnten z.B. Manager oder auch Personen aus der Qualitätssicherung sein. Das CLI ist vor allem für die Automatisierung und somit für DevOps-Spezialisten, sowie für Software-Entwickler gedacht.

5.2 Funktionale Anforderungen aus der Aufgabenstellung

5.2.1 Analyseprojekt

Als Benutzer möchte man ein Analyseprojekt definieren können. Der Benutzer kann damit die genauen Analyse-Optionen für die gewünschten Projekte und/oder Solutions definieren und abspeichern. Damit hat er die Möglichkeit, diese Analyse jederzeit zu wiederholen und die Resultate über die Zeit miteinander zu vergleichen. Zu diesen Analyse-Optionen gehört die Filterung nach bestimmten Fehlertypen sowie die Festlegung von verschiedenen Zeit bzw. Genauigkeitsparametern (werden in Abschnitt 7 genauer beschrieben).

5.2.2 Navigation zur Fehler-Stelle

Es soll die Möglichkeit bestehen, aus den Resultaten zu den entsprechenden Fehler-Stellen navigieren zu können.

5.2.3 Generierung von grafischen Report-Files

Die Resultate der durchgeführten Analyse können in ein File exportiert werden. Das Anforderungsdokument nennt dafür Word und PDF als Formate. Wir entschieden uns in Absprache mit Prof. Bläser aber gegen das PDF-Format und wollen Berichte nur in Word-Dateien exportieren. So hat der Benutzer die Möglichkeit, die Dokumente zu verändern und als PDF zu speichern.

5.3 Funktionale Anforderungen aus den Erkenntnissen

Im Abschnitt 4 konnten wir einige Erkenntnisse bezüglich der Funktionalität solcher Analyse-Tools gewinnen. Diese Erkenntnisse sind vor allem ergänzend zu den bereits beschriebenen funktionalen Anforderungen.

5.3.1 Funktionalität

Vor allem der Intel Inspector XE hat uns mit seiner Funktionalität sehr überzeugt. Diese deckt sich zum grössten Teil mit den Anforderungen aus der Aufgabenstellung. Dazu gehören die Betrachtung der Fehler-Stelle im Code, die Erstellung eines Analyse-Projekts und die Filterung nach bestimmten Fehlertypen.
Mit den Bounds decken wir den Präzisionsfaktor des ThreadSanitizer und das `effort`-Flag von FindBugs ab.
Die Visual Studio-Tools bieten die Möglichkeit von QuickFixes für problematische Code-Abschnitte. Diese Funktionalität betrachten wir im Rahmen dieser Studienarbeit nicht als Anforderung. Durch die Navigation zur Fehler-Stelle mit Hilfe von Visual Studio Code wird aber die Möglichkeit geboten, das Problem gleich selbst zu beheben.

5.3.2 Berichte

Wie bereits erwähnt, erlaubt beispielsweise der Code Prover die Generierung von Exports für ein Web-Dashboard. Ein solches würde den Rahmen dieser Studienarbeit klar übersteigen, weswegen wir einfach einen JSON-Export der Analyse-Resultate anbieten möchten. Diese Datei könnte dann für genau solche Dashboards verwendet werden.

5.3.3 Repetierbarkeit

Durch die Analyse der anderen Tools haben wir erkannt, dass die Repetierbarkeit eine sehr wichtige Funktionalität darstellt. Sie lässt sich gut über die bereits beschriebenen Funktionalität der Analyse-Projekte realisieren. Wir wollen dafür den Export der Konfigurations-Parameter in eine JSON-Datei implementieren. Diese Datei kann dann sowohl über die CLI als auch über das GUI importiert und genutzt werden.

5.3.4 CLI-Optionen

Aus der Analyse der anderen Programme haben wir erkannt, dass die Tools, welche eine CLI anbieten, eine sehr detaillierte Parametrisierung erlauben. Darum soll auch unsere CLI mit diversen Optionen konfigurierbar sein. Damit werden alle Funktionalitäten, welche ein Benutzer über das GUI zur Verfügung hat, auch über die CLI zur Verfügung gestellt.

5.4 Nicht-funktionale Anforderungen

Um unsere Zielgruppe bei der Analyse von Programm-Code zu unterstützen, wird bei der Realisierung des GUI stark auf eine gute Usability geachtet. Der Benutzer soll jederzeit wissen in welcher Phase der Analyse er sich gerade befindet und welche Interaktionsmöglichkeiten ihm zur Verfügung stehen.

5.4.1 Statusinformationen

Der Benutzer soll während der Analyse jederzeit nachvollziehen können, was der aktuelle Stand ist. Dafür werden viele Statusinformationen benötigt. Diese sollen einerseits textuell als auch visuell sein.

5.4.2 Übersichtlichkeit

Wir betrachten die Übersichtlichkeit als wichtige nicht-funktionale Anforderung, welche in den analysierten Programmen nicht immer erreicht wurde. Häufig fehlt eine Übersicht, da die Resultate werden auf zu wenig Raum dargestellt werden. Wir werden in unserem GUI darauf achten, dass die Resultate mit genügend Platz und übersichtlich dargestellt werden. Der Benutzer soll auf den ersten Blick sehen, was ihn interessiert und nicht nach den Informationen suchen müssen.

5.4.3 Plattformunabhängigkeit

Der Parallel Checker von Prof. Dr. Luc Bläser wurde .NET Standard konform implementiert, wodurch die Library plattformunabhängig verwendet werden kann. In den Anforderungen wird nur ein GUI für Windows gefordert. Wir haben uns mit Prof. Bläser abgesprochen und die Plattformunabhängigkeit als nichtfunktionale Anforderung hinzugefügt.

6 Umsetzung

6.1 Architekturentscheidungen

6.1.1 Grundsätzliche Überlegungen

Aktuell werden Applikationen oft als Webanwendung und nicht mehr als lokale Applikation entwickelt. Dies hat den Vorteil, dass keine Installation notwendig ist und man die Applikation unabhängig vom Betriebssystem nutzen kann.
Webanwendungen bieten jedoch eine besonders grosse Angriffsfläche, da sie über das Internet für jeden erreichbar sind. Neben den gängigen Webrisiken (OWASP Top 10 [17]) stellt der hochgeladene Code eine hohe Gefahr dar. Es ist nicht entscheidbar (kann auf das Halteproblem zurückgeführt werden), ob ein Stück Code auf einem System Schaden anrichtet. [18] Somit können wir nicht zwischen einem bösartigem und harmlosen Upload unterscheiden.

Wegen diesen Risiken haben wir uns gegen eine Weblösung entschieden. Durch die lokale Ausführung unserer Applikation hat jeder Nutzer selbst die Kontrolle, welchen Code er kompilieren lässt und muss dadurch auch mit den Konsequenzen leben, die bei schädlichem Code entstehen können.

6.1.2 Prototyp mit Electron

Der HSR Parallel Checker von Prof. Dr. Bläser wurde .Net Standard konform entwickelt, wodurch die Library sowohl mit .NET Framework, wie auch mit .NET Core verwendet werden kann. [2]
.NET Core verbreitet sich immer mehr als plattformunabhängige Alternative zu .NET Framework. Somit wäre es auch wünschenswert, dies bei der Konstruktion unserer Applikation zu berücksichtigen. Um plattformunabhängige Desktop-Applikationen zu erstellen, wird sehr häufig Electron verwendet. Electron basiert auf Chromium und kann somit einfach in Kombination mit aktuellen Webtechnologien wie React und TypeScript verwendet werden. [19] [20]

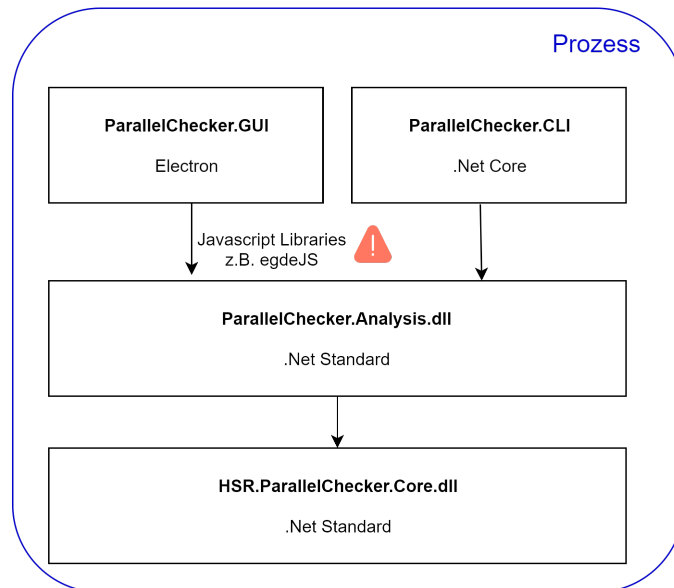

Abbildung 4: Architektur mit plattform-unabhängigem GUI

Um den HSR Parallel Checker von Prof. Bläser optimal nutzen zu können, wurde eine Wrapper Library geschrieben. Dieser Wrapper wird sowohl vom Electron-Frontend als auch vom CLI verwendet.

Leider hat sich herausgestellt, dass das Einbinden von .NET Libraries in Electron Applikationen sich zum aktuellen Zeitpunkt nicht zufriedenstellend umsetzen lässt. Javascript Libraries wie EdgeJS sind zu wenig ausgereift [21], was dazu führt, dass bei einer Kombination von Electron und C# der C# Code in einem separatem Prozess ausgeführt werden muss. Im Gegensatz zu Threads haben Prozesse keinen gemeinsamen

Adressraum. [3] Dies führt dazu, dass spezielle Mechanismen verwendet werden müssen, um zwischen zwei Prozessen zu kommunizieren. Die Objekte müssen beim Senden serialisiert und beim Empfänger wieder zu Objekten zusammengesetzt werden. Somit ensteht ein Mehraufwand beim Austausch zwischen GUI und Logik, sowie eine gewisse Fehleranfälligkeit, da Nachrichten verloren gehen können.

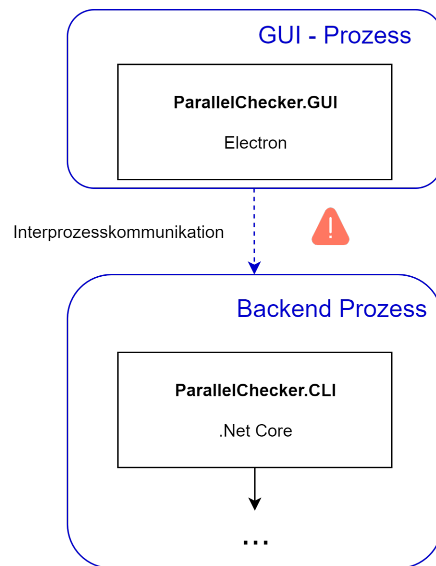

Abbildung 5: Electron in Kombination mit C#

Die CLI könnte man zwar einfach mit Interprozessmechanismen erweitern, die genannten Nachteile haben uns jedoch dazu bewogen einen neuen Prototyp mit WPF zu entwickeln.

6.1.3 Prototyp mit WPF

Durch den Wechsel zu WPF kann die Analyse wieder im selben Prozess stattfinden, in welchem auch das GUI läuft. Die Interprozess-Kommunikation entfällt und es können wieder Objekte zwischen den verschiedenen Software Komponenten ausgetauscht werden. Damit die Projekte unabhängig vom Betriebssystem gebuildet werden können, haben wir die Library Buildalyzer [22] [23] verwendet. Diese hat sich leider beim Testen des zweiten Prototyps für unsere Zwecke als unzureichend herausgestellt. Bei einigen Projekten konnten so weniger Referenzen aufgelöst werden, als mit der teilweise .NET Framework abhängigen .Net Compiler Plattform API. [24]

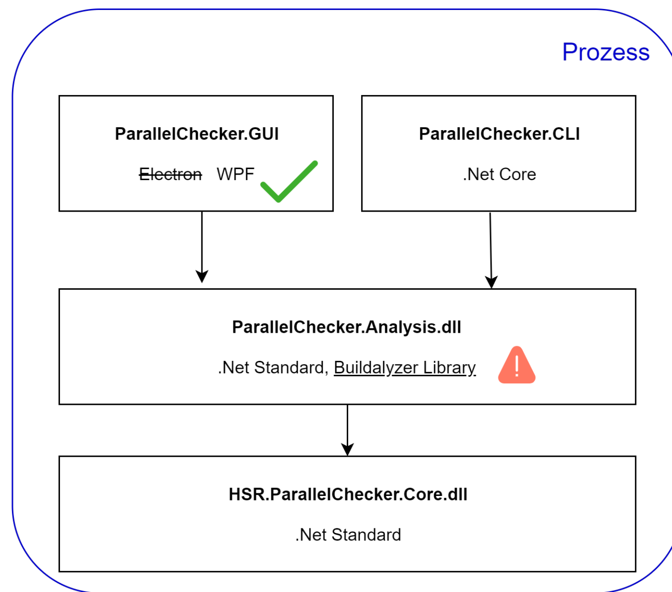

Abbildung 6: Architektur mit Windows-GUI und plattformunabhängiger CLI

6.1.4 Umgesetzte Architektur

Wie im oberen Abschnitt erwähnt, haben wir für die Umsetzung die offizielle API von der .Net Compiler Plattform verwendet. Teile daraus können leider nur mit .NET Framework verwendet werden. Somit sind nun GUI und CLI der finale Variante nur auf Windows lauffähig. Die Praxistauglichkeit unserer Lösung wird anhand von grösseren Projekten in Abschnitt 7 genauer beleuchtet.

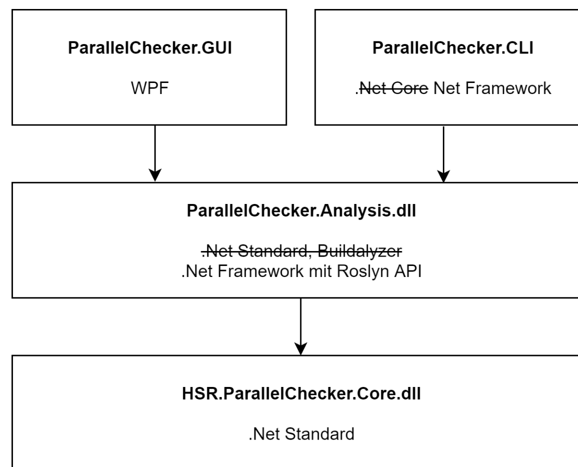

Abbildung 7: Architektur mit GUI und CLI für Windows

6.2 UI Design und umgesetzte Features

Da die Zielgruppe für unsere Desktop-Applikation vor allem aus Managern besteht, haben wir einen hohen Wert auf eine einfache Bedienbarkeit gelegt. Unser Ziel war es, nur so viele Informationen darstellen, wie notwendig. Das User Interface haben wir hierfür in drei verschiedene Views unterteilt.

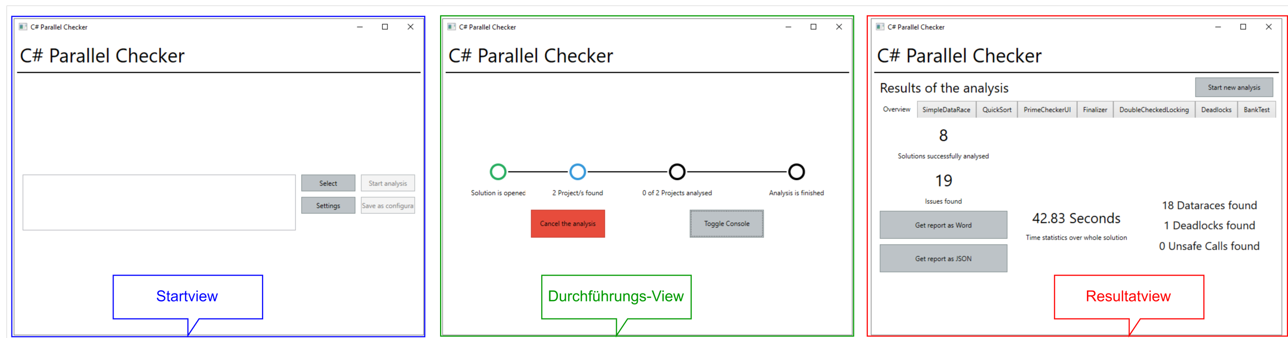

Abbildung 8: Ablauf der Views

In den folgenden Abschnitten werden die drei Views detailliert erklärt.

6.2.1 Startview

Die erste View stellt den Startpunkt der Applikation dar. Hier können Projekte und Solutions ausgewählt werden. Mittels eines Buttons werden die Analyseparameter sichtbar. Einzelne Fehlertypen können in der erweiterten Ansicht exkludiert, sowie die Genauigkeit und Dauer der Analyse beeinflusst werden (die Parameter werden im Abschnitt 7 genauer beschrieben und auf deren Einfluss untersucht).

Zusätzlich hat der Benutzer auf der Startview auch die Möglichkeit, die ausgewählten Projekte zusammen mit ihrer Konfiguration als Analyseprojekt im JSON Format zu exportieren. Diese Analyseprojekte können dann auch wieder über die Startview importiert werden.

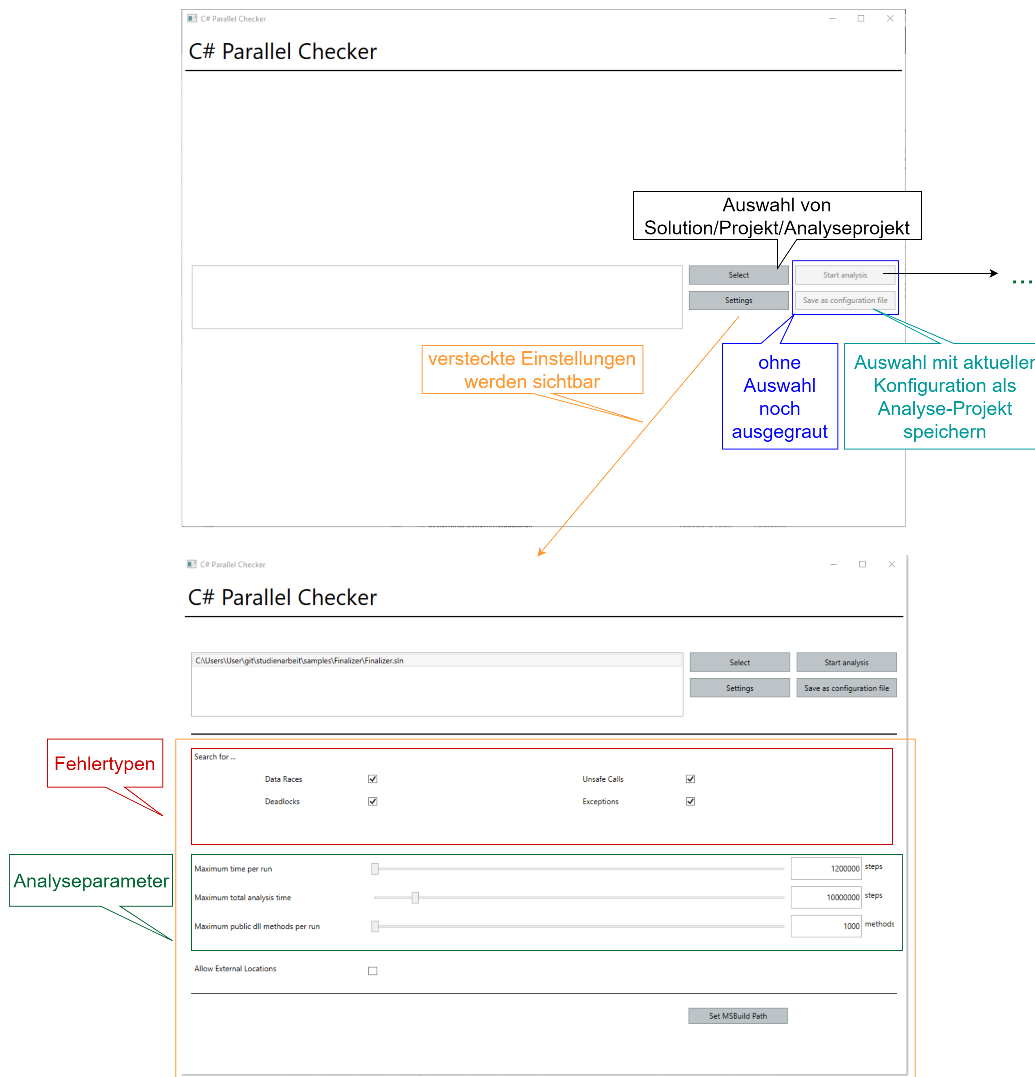

Abbildung 9: Startview

6.2.2 Durchführungs-View

Für die Durchführung der Analyse wird die zweite View geöffnet. Hier hat der Benutzer die Möglichkeit, den Fortschritt der Analyse zu verfolgen. Dafür werden ihm alle interessanten Informationen geboten. Dazu gehören:

- Konnte die Solution oder das Projekt geöffnet werden?
- Wie viele Projekte wurden gefunden?
- Wie viele Projekte wurden bisher untersucht?
- War die Analyse erfolgreich?

Über einen Button kann sich der User den Log Output ansehen. Weiter besteht die Möglichkeit die Analyse jederzeit abzubrechen und auf die Startview zurückzukehren.

Sobald die Analyse abgeschlossen ist, gelangt der Benutzer über Buttons auf die Resultatview oder zurück auf die Startview.

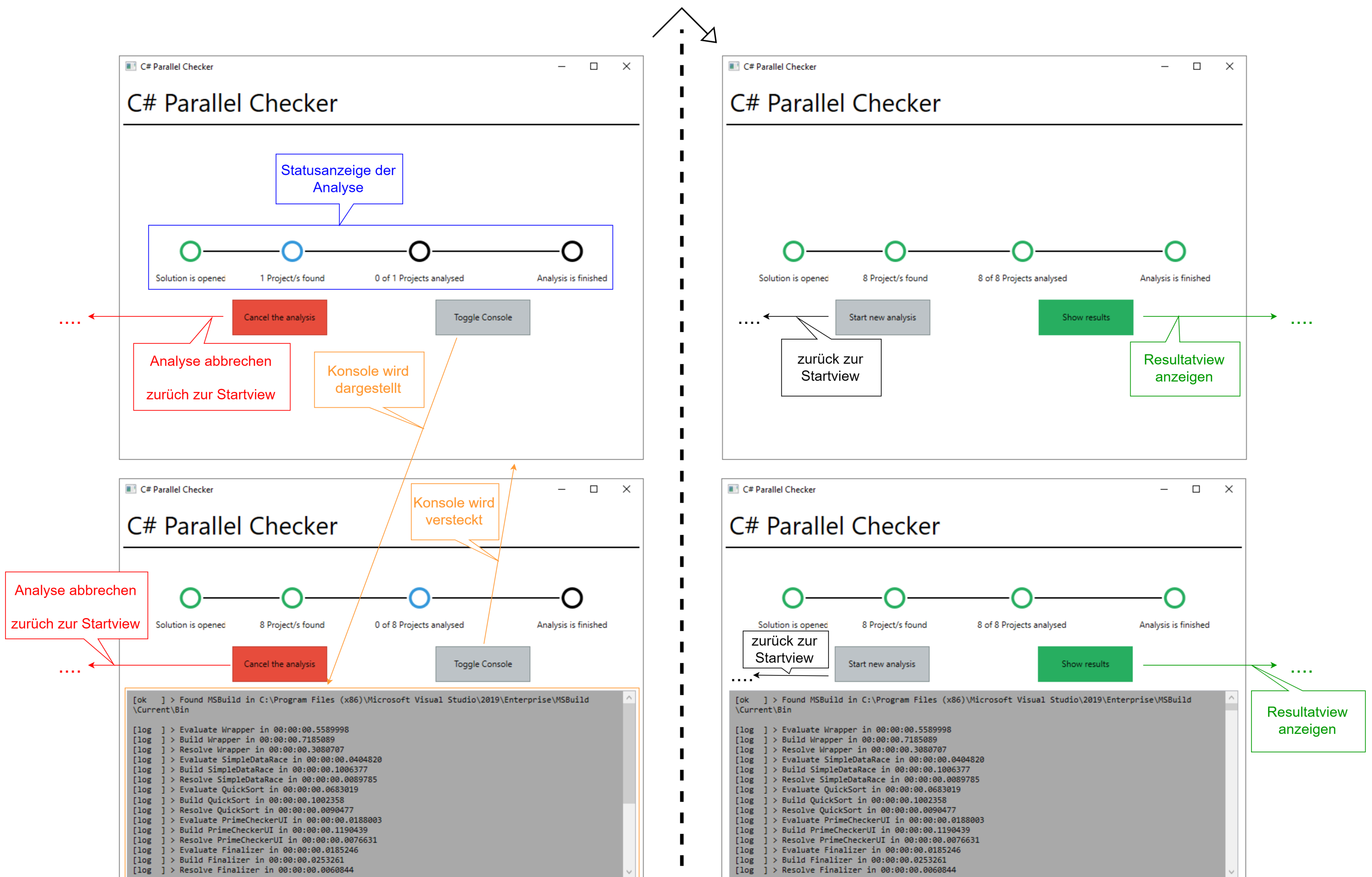

Abbildung 10: Durchführungs-View

6.2.3 Resultatview

Die Resultate der Analyse werden auf der Resultat-View dargestellt. Dabei stehen dem Benutzer mehrere Unteransichten zur Verfügung. Die erste davon ist eine Übersicht über die gesamte Analyse. Dabei wird dem Benutzer angezeigt, wie lange die gesamte Analyse gedauert hat und wie viele Fehler in den einzelnen Kategorien gefunden wurden.
Zudem steht dem Benutzer hier das Report-Feature zur Verfügung. Über zwei Buttons hat er die Möglichkeit,die Resultate der Analyse als Word- oder JSON-Report zu exportieren.

Neben der Übersicht über die Analyse steht dem Benutzer auch eine Detail-Ansicht für jedes analysierte Projekt zur Verfügung. In dieser Ansicht kann sich der Benutzer die gefundenen Nebenläufigkeitsfehler als Liste darstellen. Diese Liste steht sowohl textuell als auch visuell zur Verfügung. Die visuelle Darstellung stellt die gefundenen Nebenläufigkeitsfehler als Graph dar.

Der Benutzer kann jederzeit eine neue Analyse starten. Dafür steht ihm ein Button zur Verfügung.
In den folgenden zwei Abschnitten werden zwei sehr zentrale Features unseres Parallel Checkers genauer erklärt. Das ist zum einen die Möglichkeit, Graphen zu den gefundenen Nebenläufigkeitsfehlern zu generieren und andererseits die Erstellung von Report-Files aus den durchgeführten Analysen.

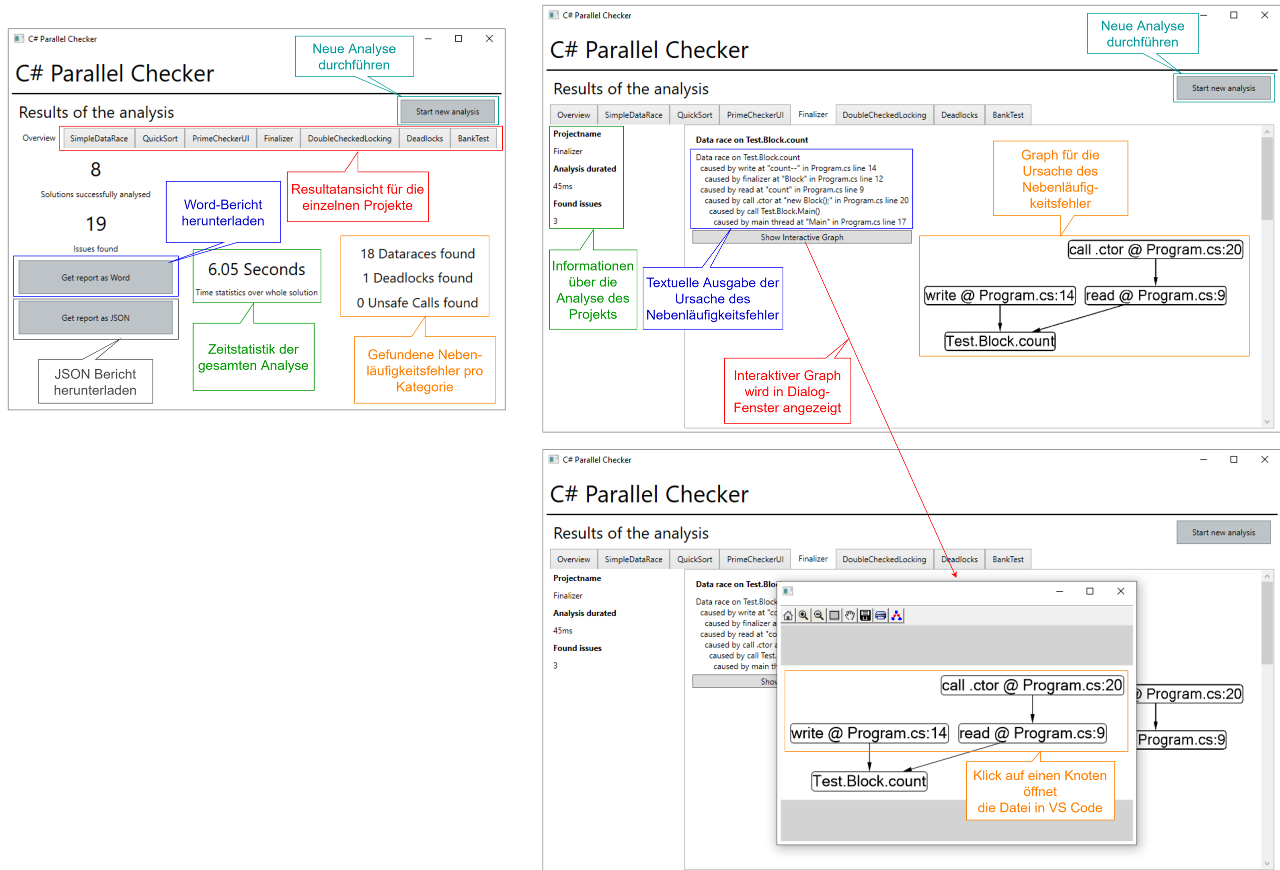

Abbildung 11: Resultatview

6.2.4 Graphen

Wie man bereits in der Abbildung 11 erkennen kann, zeigen die Graphen die Entstehung eines Nebenläufigkeitsfehler auf. Sie werden für jeden Issue dargestellt und können zusätzlich in einem eigenen Dialog-Fenster betrachtet werden. Im Fenster kann sich der Benutzer den Graphen als Bild abspeichern lassen. Weiter wird bei einem Klick auf einen Knoten die entsprechende Codestelle in Visual Studio Code geöffnet. Die folgende Abbildung zeigt einen Graphen, den wir aus einer Analyse exportiert haben.

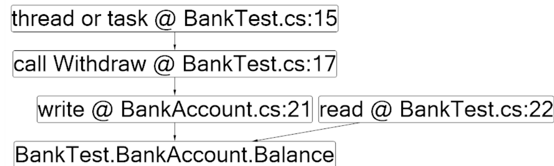

Abbildung 12: Data Race Graph

Für das Rendern der Graphen wird eine Windows.Forms Komponente benötigt, die von WPF nicht fehlerfrei unterstützt wird. [25] So kann z.B. das Scrollen durch eine Liste nicht richtig umgesetzt werden. Um trotzdem eine ansprechende View zu erhalten, haben wir den interaktiven Graphen als Workaround in ein eigenes Fenster verlagert.

6.2.5 Reports

Wie bereits erwähnt können Reports im Word oder JSON Format erstellt werden. Darin werden die allgemeinen Informationen zur durchgeführten Analyse aufgeführt. Zusätzlich werden die Informationen zu jedem analysierten Projekt exportiert. Die oben beschriebenen Graphen werden ebenfalls exportiert. Wie man in der Abbildung 13 erkennen kann, werden sie im Word-File direkt bei den jeweiligen Nebenläufigkeitsfehlern dargestellt. Für den JSON-Report werden sie in einen Unterordner exportiert.

Project: **BankTest**

Time needed for analysis: 18ms
Number of found issues: 2

Found issues:
Data race on BankTest.BankAccount.Balance

thread or task @ BankTest.cs:11
call Deposit @ BankTest.cs:11
write @ BankAccount.cs:11 | read @ BankTest.cs:22
BankTest.BankAccount.Balance

caused by write at "Balance += amount" in BankAccount.cs line 11
caused by call Deposit at "() => { account.Deposit(100); }" in BankTest.cs line 11
caused by thread or task at "() => { account.Deposit(100); }" in BankTest.cs line 11
caused by call BankTest.BankTest.Main()
caused by main thread at "Main" in BankTest.cs line 8
caused by read at "account.Balance" in BankTest.cs line 22
caused by call BankTest.BankTest.Main()
caused by main thread at "Main" in BankTest.cs line 8

Data race on BankTest.BankAccount.Balance

thread or task @ BankTest.cs:15
call Withdraw @ BankTest.cs:17
write @ BankAccount.cs:21 | read @ BankTest.cs:22
BankTest.BankAccount.Balance

caused by write at "Balance -= amount" in BankAccount.cs line 21
caused by call Withdraw at "result = account.Withdraw(50)" in BankTest.cs line 17
caused by thread or task at "() => { var result = account.Withdraw..." in BankTest.cs line 15
caused by call BankTest.BankTest.Main()
caused by main thread at "Main" in BankTest.cs line 8
caused by read at "account.Balance" in BankTest.cs line 22
caused by call BankTest.BankTest.Main()
caused by main thread at "Main" in BankTest.cs line 8

```
1  {
2    "totalTime": "00:00:09.1061723",
3    "analysisTime": "00:00:02.3340932",
4    "projects": [
5      {
6        "analysisTime": 1346,
7        "compilationResultName": "NoError",
8        "compilationResultValue": 0,
9        "projectName": "Wrapper",
10       "issues": []
11     },
12     {
13       "analysisTime": 201,
14       "compilationResultName": "NoError",
15       "compilationResultValue": 0,
16       "projectName": "SimpleDataRace",
17       "issues": [
18         {
19           "message": "Data race on x",
20           "category": 0,
21           "originalDescription": "Data race on x\r\n  caused by write
22           "imageName": "865d7f21-1c35-4bd8-bb69-7099c6ffaa9d.png",
23 >         "causes": [···
61           ]
62         }
63       ]
64     },
65     {
66       "analysisTime": 217,
67       "compilationResultName": "NoError",
68       "compilationResultValue": 0,
69       "projectName": "QuickSort",
70 >     "issues": [···
```

Abbildung 13: Report-Files (Word und JSON)

6.3 Analyseablauf

In diesem Abschnitt wird der Ablauf einer Analyse und das Zusammenspielen der einzelnen Software Komponenten erklärt. Der Ablauf mit der CLI Komponente erfolgt ähnlich. Die Parameter werden dort jedoch über ein Analyseprojekt mitgegeben.
Jeder der Schritte, sowie Teilschritte werden durch eine Loggerkomponente dem GUI mitgeteilt. Dies ist notwendig, da die Anzeige der Daten von der Logik getrennt sein soll. Die Loggingschritte wurden in der Grafik Zweck besserer Übersicht nicht aufgeführt.

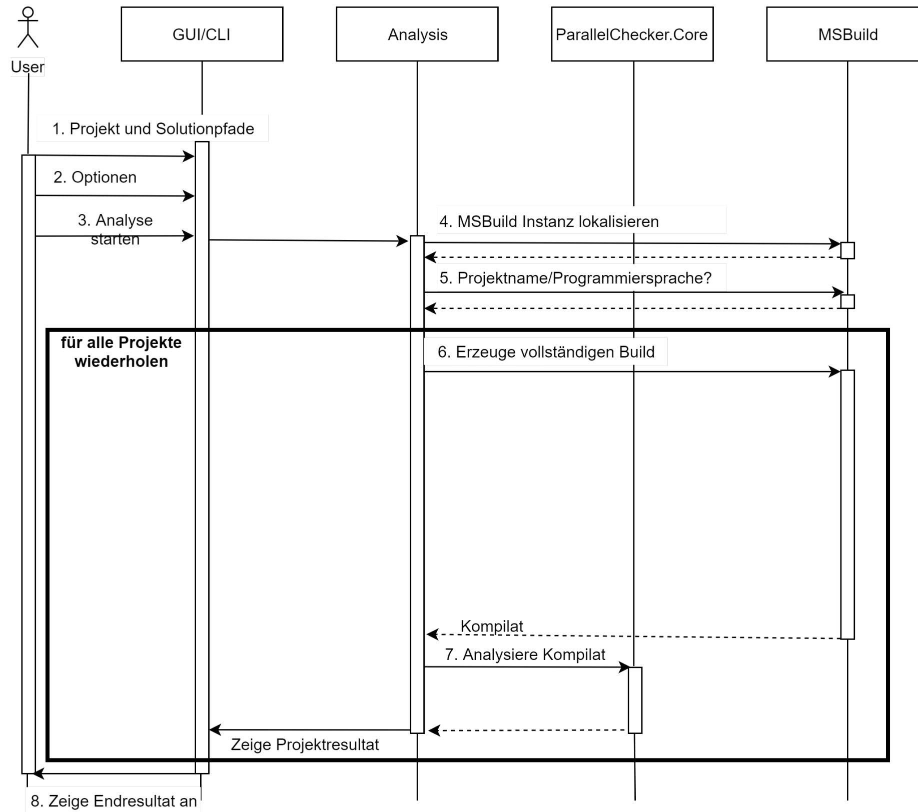

Abbildung 14: Analyse Ablauf

1. Als Erstes wählt der User mehrere Projekte und/oder Solutions aus, die er analysieren möchte.
2. Weiter kann der Benutzer in der Startview Analyseparameter pro Pfad, sowie die gewünschte MS-Build Instanz angeben (MSBuild muss lokal installiert sein). Wenn er keine Eingaben trifft, werden die Standard-Werte verwendet.
3. Auf Knopfdruck wird die Analyse gestartet.
4. Die Analysis Komponente registriert die gewählte MSBuild Instanz oder standardmässig die erste, die es auf dem System findet.
5. Anhand der Pfade werden die Projekte gesucht und ein Design-Time Build wird erstellt. Diese spezielle Art von Build erstellt ein Projekt Objekt, welches allgemeine Daten wie Projektname, Programmiersprache etc. beinhaltet. Diesen Build verwenden wir zum Beispiel, um Projekte zu überspringen, die nicht in C# geschrieben wurden. [26]
6. Der effektive Build wird gestartet und für jedes Projekt wird ein Kompilat erzeugt.
7. Nach der Überprüfung auf Kompilierungsfehler wird jedes Kompilat einzeln dem Checker-Kern übergeben und durch diesen mit gewählten Parameter analysiert.
8. Nach dem Analyseende werden die Resultate auf dem GUI/CLI angezeigt.

6.4 Concurrency Design

Damit unsere Applikation auch während der Analyse reaktionsfähig bleibt und nicht einfriert, haben wir eine Multithreading Architektur verwendet.

GUI Frameworks erlauben nur Single-Threading. Somit darf nur der spezielle UI-Thread auf UI-Komponenten wie Buttons, Text etc. zugreifen. Wird dies nicht eingehalten, können Race Conditions enstehen. Möchte man längere Rechenoperationen in einen Worker Thread auslagern und das UI nach Beendigung benachrichtigen, muss die Benachrichtung in eine Queue eingereiht werden. Dies wird Dispatching genannt. [3]

In unserer Applikation haben wir die Erstellung des Design Time Builds, das Kompilieren der Projekte sowie die Analyse aus dem UI-Thread in einen TPL-Worker-Thread ausgelagert. (1) Dafür haben wir die neuere `async` /`await` Syntax verwendet. Durch den Aufruf einer asychronen Methode aus dem UI-Thread `await` wird das Task Resultat automatisch gedispatched. (2) [3]
MSBuild erstellt Progress Objekte um Fortschritte beim Buildvorgang mitzuteilen. Diese Objekte erhält man, wenn man einen Eventhandler registiert. Da sich Events nicht mit dem `async`/`await` Pattern kombinieren lassen und nicht, wie fälschlicherweise oft angenommen, asynchron sind, ist es wichtig, die Objekte manuell zu dispatchen. (3)

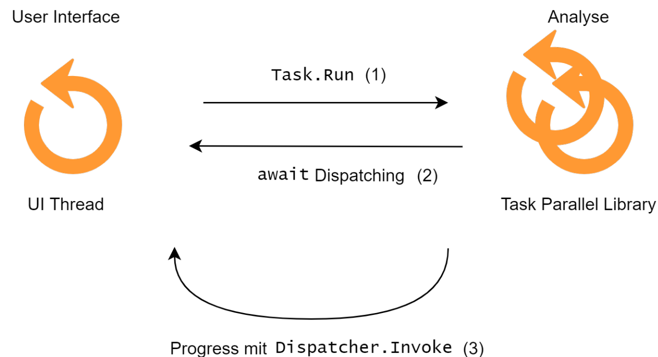

Abbildung 15: Verwendetes Concurrency Design

7 Auswertung

Prof. Dr. Bläser hat bereits in seiner Publikation Tests mit populären Open Source C# Projekten durchgeführt. [1] Wir haben eine Auswahl davon übernommen und unsere eigenen Tests damit durchgeführt. Folgende Untersuchungen haben wir dabei vorgenommen:

- **Tauglichkeit:** Wir haben analysiert, ob unser Tool grosse reale Projekte verarbeiten kann. Dies ist im Hinblick auf die produktive Nutzung von hoher Bedeutung.
- **Parametriesierung:** Wir haben untersucht, wie sich veränderte Parameter auf die Menge an gefundenen Fehler auswirkt. Dabei haben wir untersucht, ob alle Parameter den gleichen Einfluss auf die Resultate haben.

7.1 Auswahl an Projekten [1]

Folgende Projekte haben wir von denen im Paper [1] ausgewählt. Die Versionen der einzelnen Projekte unterscheiden sich von denen, die im Paper untersucht wurden. Dies führt einerseits zu einer anderen Anzahl an Codezeilen (LOC) und sowie zu anderen Nebenläufigkeitsfehlern.

Projekt	Version	Analysierte Solution	Lines Of Code
SignalR	2.4.1	Microsoft.AspNet.SignalR.sln	53’671
Orleans	3.0.0	Orleans.sln	149’480
NLog	4.6.8	NLog.sln	137’555
Polly	7.1.1	Polly.sln	57’984
ILSpy	5.0.2	ILSpy.sln	185’783
MSBuild	16.3.2.50909	MSBuild.sln	511’372
Total			1’095’845

Tabelle 2: Testprojekte

Das Projekt Rx.Net wäre auch noch interessant gewesen, da beim Benchmark des Papers [1] einige Issues entdeckt werden konnten. Leider war es nicht möglich dieses Projekt mit den Visual Studio Tools zu builden. Man hätte noch viel zusätzliche Software installieren müssen, welche man nicht einfach über den Visual Studio Installer installieren konnte.

7.2 Setup

Für die Tests haben wir pro Projekt eine virtuelle Maschine mit Microsoft Windows 10 Pro und Visual Studio Build Tools 2019 verwendet.

Die virtuelle Maschine liessen wir mit 8 GB RAM auf unterschiedlichen Hosts laufen. Als Virtualisierungslösung haben wir VMware Workstation 15 Player verwendet.

Je nach Projekt waren andere .NET Software Development Kits (SDKs) oder Target Pakete notwendig, die wir dann via Visual Studio Installer oder manuell installiert hatten. Um gewisse Projekte zu erstellen, mussten zudem node package manager oder Git installiert werden. Das Builden der Projekte mit den Visual Studio Tools stellte sich teilweise schon als Herausforderung heraus.

7.3 Ergebnisse

ILSpy liess sich fehlerfrei builden. Leider sind wir beim Testen mit den anderen Projekten auf sehr viele Hindernisse gestossen. Zusammenfassend lässt sich sagen, dass unsere Lösung für grössere Projekte nicht zuverlässig funktioniert.

[1] LOC mittels Visual Studio Metriken berechnet

Projekt	Buildresultate
SignalR	2 von 32 Projekten mit Kompilierungsfehler
Orleans	2 von 85 Projekten ohne Entrypoint
NLog	15 von 29 Projekten mit Kompilierungsfehler
Polly	2 von 6 Projekten mit Kompilierungsfehler
ILSpy	ohne Fehler
MSBuild	9 von 42 Projekten mit unterschiedlichen Fehler

Tabelle 3: Testresultate

7.3.1 MSBuild Probleme

Die Projekte Polly, NLog und SignalR liessen sich zwar auch mit unserem GUI und CLI builden, jedoch nur mit Kompilierungsfehler. Bei all diesen Programmen sind dabei jeweils ähnliche Kompilierungsfehler im Stile des unten aufgeführten Beispiels aufgetreten.

```
error CS0012: The type 'Object' is defined in an assembly that is not referenced.
You must add a reference to assembly 'netstandard, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=cc7b13ffcd2ddd51
```

Nach einigen Tagen ausprobieren, hat sich herausgestellt, dass wir diesen Fehler nicht erhalten, wenn das analysierte Projekt mindestens .Net Standard 2.0 konform (bedeutet Projekte müssen mind. .Net Framework 4.6.1 oder .Net Core 2.0 verwenden) ist. Jedoch ist nicht jedes ältere Projekt fehlerhaft. Ein Ändern der im Frontend referenzierten NuGet-Pakete oder auch Teile der API hat das Problem nicht gelöst.

Bei MSBuild Solution sind in 9 von 42 Projekten Fehler aufgetreten. Diese Fehler haben wir in anderen Projekten nicht beobachtet. Einige Fehlermeldungen betreffen Referenzen, die nicht gefunden wurden.

Orleans lässt mit Ausnahme von 2 Projekten fehlerfrei builden. Bei den fehlerhaften Projekten wird kein Entrypoint gefunden. Diese Fehler sind unserer Meinung nach aber vernachlässigbar, da sie keine Files beinhalten, sondern nur Referenzen auf andere Projekte.

Grunsätzlich haben wir beobachtet, dass sich die Build Vorgänge im Visual Studio, in der Konsole (msbuild.exe) und über die API stark unterscheiden. Alle getesteten Projekte lassen sich ohne Probleme mit dem Visual Studio Plugin analysieren. Mit der API [24] gibt es aber teilweise sehr viele Fehler. Weiter lässt sich feststellen, dass die Probleme meistens beim Auflösen von Referenzen enstehen. Aufgrund mangelnder Dokumentation ist es sehr schwierig eine Lösung dafür zu finden. Im nächsten Abschnitt diskutieren wir zwei Lösungsmöglichkeiten. Wir haben während des Projektverlaufs einige Vermutungen bezüglich der Build-Unterschiede zwischen der Plugin Variante und unserer Applikation aufgestellt, leider konnten wir keine davon beweisen.

- **Bessere Mechanismen:** Visual Studio könnte andere, ausgereiftere Mechanismen verwenden, um die Referenzen aufzulösen.
- **Andere Referenzen:** Visual Studio könnte andere Referenzen laden.
- **Fehler werden geschluckt:** Es könnte sein, dass die beobachteten Fehler intern von Visual Studio einfach geschluckt werden und die Analyse ohne Hinweis auf dem unvollständigen Build ausgeführt wird.
- **Verschiedene Prozesse:** Es könnte sein, dass das Ausführen der Analyse und des Buildens im selben Prozess Probleme verursacht. In Visual Studio wird dies anders gehandhabt.

7.4 Mögliche Lösungsansätze

Um die oben genannten Probleme zu umgehen, haben wir zwei Alternativen angetestet:

7.4.1 Parallel Checker als nuGet-Paket

Der Parallel Checker wurde als Roslyn Analyzer konzipiert. Roslyn Analyzers können zu einem Visual Studio Plugin, sowie auch zu einem NuGet-Paket gebuildet werden. Der Nachteil von Analyzers, die per NuGet-Paket ausgeliefert werden, ist, dass das Paket bei jedem Projekt einzeln hinzugefügt werden muss. Auf der anderen Seite wird der hinzugefügte Analyzer bei jedem Buildvorgang mitausgeführt. Die Überlegung war, ob man das Paket automatisiert hinzufügen kann und so weniger Fehler als mit dem bisherigen API-Ansatz erhält. [27]

Dieser Ansatz ist leider nicht umsetzbar, da ältere Projekte (< .Net Standard 2.0 bzw. .Net Framework 4.6.1, .Net Core 2.0) nicht mit dem NuGet-Paket kompatibel sind. Das NuGet-Paket lässt sich somit bei älteren Projekten nicht hinzufügen.

7.4.2 SonarQube Ansatz

SonarQube bietet statische Analysetools für viele Programmiersprachen an, unter anderem auch für C#. In Korporation mit Microsoft wurde der SonarScanner for MSBuild entwickelt. Anders als unser Ansatz über die MSBuild API Calls und den oberen Ansatz hängt sich der SonarScanner direkt in die MSBuild Pipeline ein. [28] [29]

Die SonarQube Tools können durch eigene Analyzers erweitert werden. [30] Um diese Variante zu testen, haben wir versucht ein SonarQube Plugin zu erstellen. Dafür steht ein Hilfstool zur Verfügung. [31] Leider ist dies uns nicht gelungen. Die erste Hürde war, dass SonarQube nur ältere Roslyn Versionen bis und mit Version 2.8.2 unterstützt. Prof. Dr. Bläser hat uns freundlicherweise eine Parallel Checker Version mit älteren Roslyn-Abhängigkeiten erstellt, diese konnte mit dem Tool jedoch auch nicht zu einem Plugin umgewandelt werden (Fehlermeldung siehe Anhang).

Aufgrund von mangelnder Zeit haben wir uns nicht mehr weiter damit beschäftigt. Eventuell könnte ein ähnlicher Ansatz für den Parallel Checker ebenfalls implementiert werden. Weiter könnte man auch noch die Resharper CLI Tools genauer untersuchen. [32]

7.5 Tests mit verschiedenen Parameter

In unserer Applikation bieten wir einige Optionen an, um Einfluss auf die Analyse zu nehmen. Deshalb haben wir als zweiten Schritt getestet, inwiefern eine Änderung der Parameter eine Auswirkung auf die Analyseresultate hat.

Grunsätzlich stehen beim Checker, neben der Filterung nach den verschiedenen Fehlertypen (Data Races, Deadlocks etc.), folgende Optionen zur Verfügung: [1]

- **MaximumTimePerRun:** Der Checker führt mehrere Runden aus, in denen er Threads zufällig scheduled. Wie lange eine Runde geht, kann mit diesem Parameter (in Anzahl logischer Schritte, um deterministische Resultate zu erhalten) festgelegt werden.
- **MaximumTotalTime:** Die Totale Analyselaufzeit kann man mit diesem Parameter setzen (ebenfalls in Anzahl logischer Schritte).
- **MaximumDllEntries:** Pro Runde wird jeweils eine bestimmte Anzahl an public Methoden analysiert. Durch diesen Parameter lässt sich diese Zahl steuern.

Die Standard-Werte sind auf `MaximumTimePerRun`: 1'200'000 Schritte, `MaximumTotalTime`: 10'000'000 Schritte und `MaximumDllEntries`: 1'000 Methoden gesetzt. Diese wurden heuristisch festgelegt, um dem Nutzer des Visual Studio Plugins eine optimale Nutzererfahrung zu ermöglichen.

7.5.1 Geändertes Setup

Wie in Abschnitt 7.3 erläutert, sind die Buildresultate vieler Projekte mit unserer Applikation fehlerhaft. Für die Messungen der Parameter haben wir nun die Projekte teilweise auf eine .NET Standard 2.0 konforme Target-Version angehoben, um das oben genannte Problem zu eliminieren. Die restlichen fehlerhaften Projekte haben wir entfernt, um keine verfälschten Resultate zu erhalten. Für die Versuche haben wir unser CLI ein wenig abgeändert und die Logs auf Files geschrieben, anstelle auf der Konsole auszugeben.

7.5.2 Idealer Faktor

Als Erstes haben wir untersucht, welcher Parameterbereich die meisten Resultate liefert. Dafür haben wir bei allen Parameter die Standardwerte mit dem gleichen Faktor multipliziert. Den Test haben wir mit Faktor 1 gestartet und bis Faktor 50 durchgeführt. Wie man in der Abbildung 16 sehen kann, erzielt man stetig bessere Resultate bis zu einem Faktor von 11 (NLog) bis 17 (Orleans). Bei den meisten Projekten erreicht man danach keine höheren Werte mehr. ILSpy und Polly sind im Graphen nicht aufgeführt, da wir dort keine Fehler gefunden haben. Die starken Schwankungen sind damit zu erklären, dass je nach Einstellung der Parameter die simulierten Threads anders verzahnt werden.

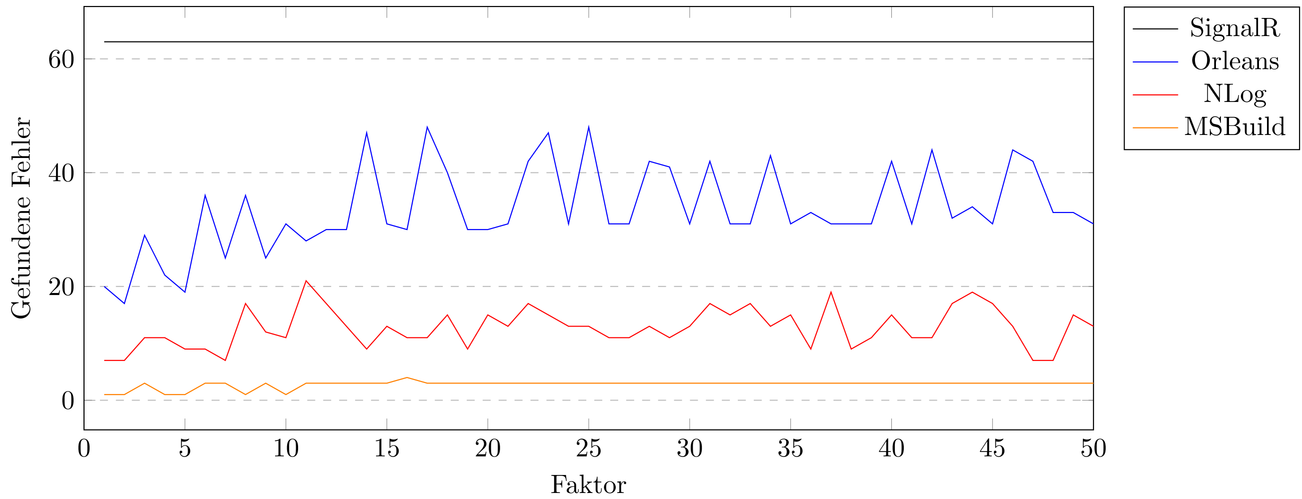

Abbildung 16: Gefundene Fehler pro Runde und Projekt

7.5.3 Akkumulierte Werte

Neben dem idealen Faktor haben wir untersucht, ob immer die gleichen Fehler gefunden werden. Dafür haben wir die neu gefundenen Fehler gezählt und über alle Faktoren aufsummiert. Wie man in Abbildung 17 sehen kann, flachen die meisten Graphen im idealen Parameterbereich ab. Orleans ist mit seinem kontinuierlich ansteigenden Graphen eine nennenswerte Ausnahme. Somit kann man sagen, dass es sich durchaus lohnen kann, eine Analyse mit unterschiedlichen Parameter zu wiederholen.

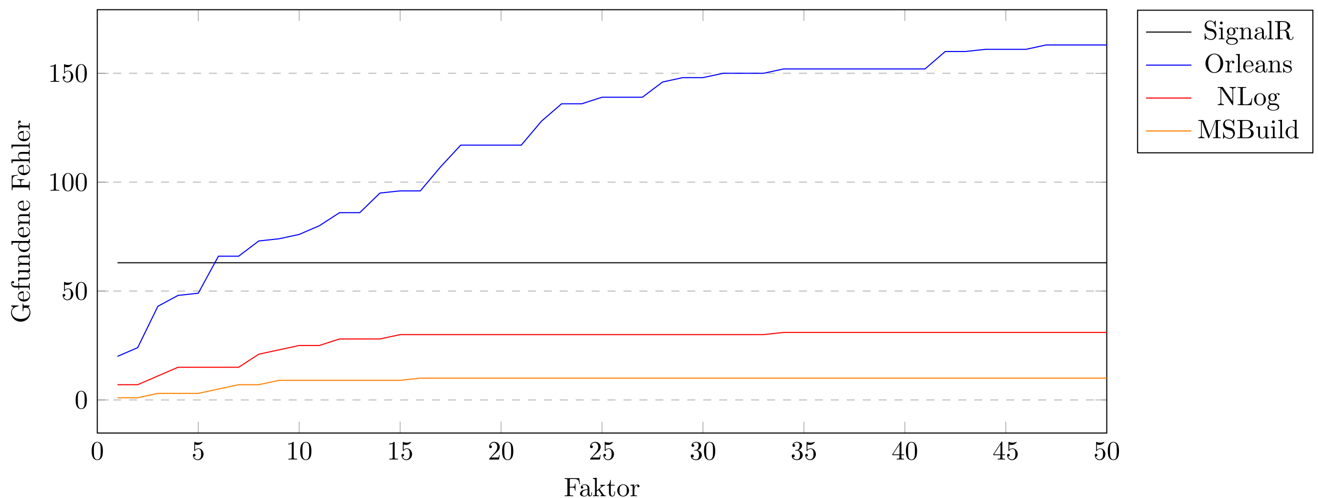

Abbildung 17: Akkumulierte Fehler pro Projekt

7.5.4 Asynchrone Werte

Die oben beschriebenen Ergebnisse basieren auf einer synchronen Erhöhung der Parameterwerte mit dem gleichen Faktor. Um eine Aussage über den Einfluss der einzelnen Parameter machen zu können, haben wir einen weiteren Versuch mit asynchronen Veränderungen durchgeführt. Wir haben analysiert, wie sich die Resultate verhalten, wenn wir jeweils nur einen der drei Parameter mit dem jeweiligen Faktor erhöhen, während die anderen beiden Parameter über die 50 Runden konstant bleiben. Dabei fällt auf, dass vor allem der Parameter `MaximumTotaTime` starken Einfluss auf die gefundenen Fehler hat. Ausser bei Orleans (siehe Abbildung 18) hatten die Parameter `MaximumTimePerRun` und `MaximumDLLEntries` nur minime Auswirkungen.

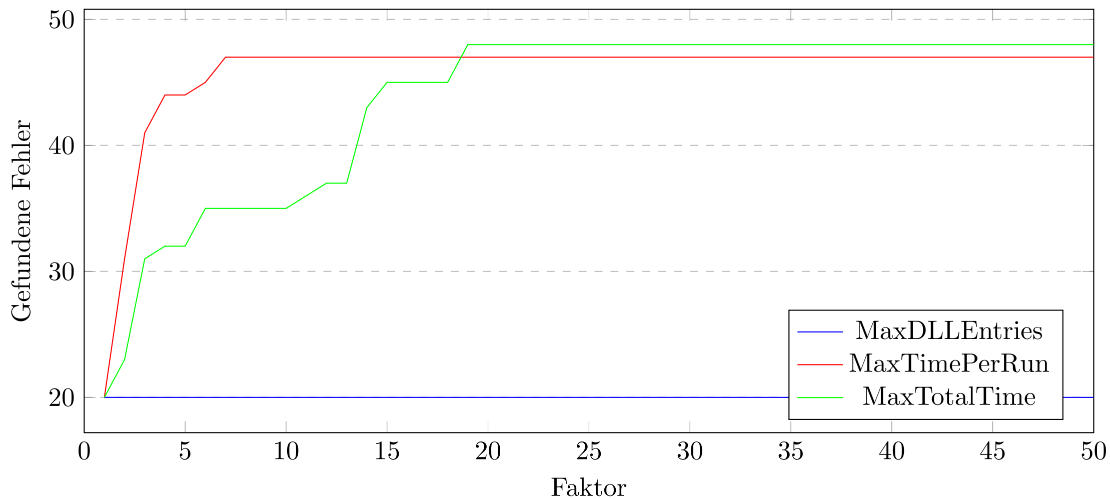

Abbildung 18: Orleans - Einfluss der einzelnen Parameter

In obiger Abbildung erkennt man ein plötzliches Abflachen der Graphen für die Parameter `MaximumTimePerRun` und `MaximumTotalTime`. Wenn man diese Abbildung mit der Abbildung 17 vergleicht, erkennt man, dass dort auch noch nach der Runde 19 neue Fehler gefunden wurden. Wir vermuten daher, dass man für eine zuverlässige Fehlersuche die Parameter `MaximumTotalTime` und `MaximumTimePerRun` synchron vergrössern muss.

7.5.5 Asynchrone Werte, 2. Versuch

Um die Vermutung aus dem vorherigen Abschnitt zu bestätigten, haben wir eigens für Orleans eine weitere Analyse durchgeführt. Dabei wurden die Parameter `MaximumTotalTime` und `MaximumTimePerRun` synchron erhöht. Die gleiche synchrone Erhöhung wurde auch für die Parameter `MaximumTotalTime` und `MaximumDLLEntries` vorgenommen. In Abbildung 19 kann man erkennen, dass sich unsere Vermutung bestätigt. Die Parameter `MaximumTotalTime` und `MaximumTimePerRun` müssen beide vergrössert werden, um zuverlässig nach Fehlern zu suchen. Der Parameter `MaximumDLLEntries` spielt, wie vermutet, keine so starke Rolle.

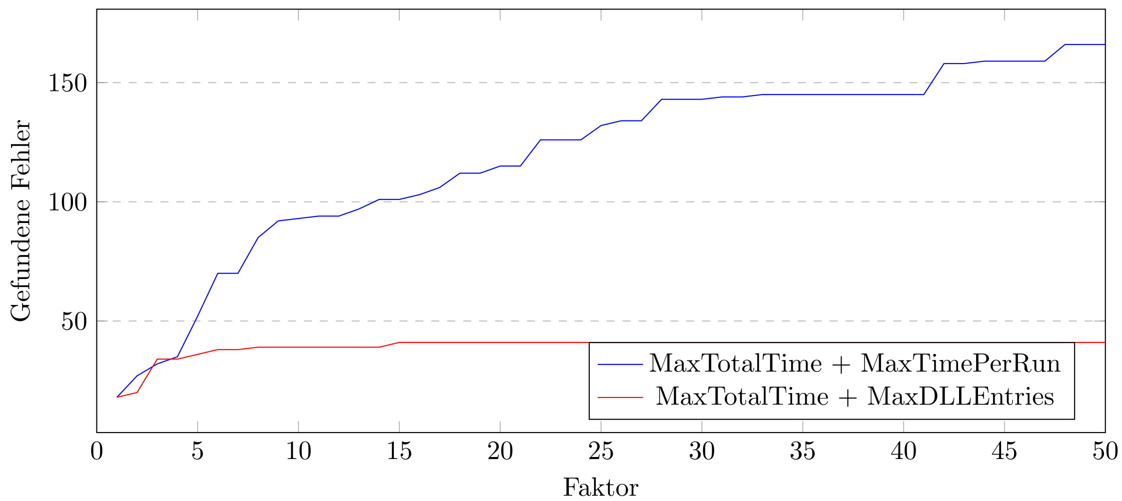

Abbildung 19: Orleans - Einfluss der einzelnen Parameter

7.5.6 Erkenntnisse

Wegen den Schwankungen zwischen zwei benachbarten Faktoren und den Unterschieden zwischen den Projekten lassen sich keine idealen Parameterwerte ableiten. Wir können jedoch eine Empfehlung für ein Wertebereich angeben. Eine einmalige Analyse mit diesen Parametereinstellungen findet vergleichsweise am meisten Fehler:

`MaximumTimePerRun`	13'200'000	bis	20'400'000	Schritte
`MaximumTotalTime`	110'000'000	bis	170'000'000	Schritte
`MaximumDllEntries`	11'000	bis	17'000	Schritte

Anhand der Orleans Resultate stellen wir fest, dass es bei grösseren Projekten lohnenswert sein kann, die Analyse mit unterschiedlichen Parametern zu wiederholen. Weiter lässt sich sagen, dass der Parameter `MaximumDllEntries` gemäss dem letzten Untersuch eine untergeordnete Rolle spielt.

Für die detaillierten Messresultate verweisen wir auf den Anhang.

8 Schlussfolgerungen

Grundsätzlich sind wir mit dem Endresultat der Arbeit zufrieden, da wir die Anforderungen grösstenteils erfüllen konnten.

Wie bereits in Abschnitt 7.3.1 ausgiebig beschrieben, führen die MSBuild Probleme dazu, dass einige Projekte nicht oder nur teilweise analysiert werden können. Diese Probleme müssen leider zwingend behoben werden, um unsere Applikation produktiv einsetzen zu können. Die diskutierten Alternativen könnte man in diesem Zusammenhang noch ausgiebiger testen. Gegebenenfalls liessen sich die Probleme durch einen anderen Ansatz lösen.

Die Plattformunabhängigkeit haben wir uns selbst als Ziel gesetzt. Dies war im Nachhinein ein wenig zu ambitioniert und hat uns noch einiges an Zusatzaufwand gebracht.

Weiter haben wir im Verlaufe des Projekts erkannt, dass die angegebenen Zeilennummern in den gefundenen Fehlern nicht immer stimmen. Dies führt dazu, dass die interaktive Auswahl im Graphen nicht die korrekte Zeile im Visual Studio Code öffnet. Dieses Problem wurde mit Prof. Dr. Bläser besprochen. Inzwischen wurde es als Bug erkannt und gefixt. Die Änderungen werden mit dem nächsten Release des HSR Parallel Checkers ausgeliefert.

8.1 Ausblick

Unsere Applikation wäre noch um einige Features erweiterbar. Diese Ausbauszenarien werden in den folgenden Abschnitten diskutiert.

8.1.1 Betriebsmittel-Graph

Mit dem GUI des Parallel Checkers hat man die Möglichkeit, sich die gefundenen Fehler als Graph darstellen zu lassen. Mit der Darstellung von Data Races und Unsafe Calls sind wir zufrieden.
Für Deadlocks würde es Sinn machen, sie wie in Abbildung 1 als Betriebsmittelgraph darzustellen. Dafür müssten jedoch noch mehr Daten aus dem Checker-Kern geliefert werden, um die Warte- und Haltebeziehungen darstellen zu können.

8.1.2 Mehrere Projekte gleichzeitig parameterisierbar

Aus Zeitgründen konnten wir dieses Feature nicht umsetzen. Somit können in unserer Applikation die Analyse Parameter nur auf einzelne Projekte gesetzt werden. Es wäre aus Usability Sicht noch wünschenswert, dass man mehrere Projekte miteinander auswählen und die Parameter für die Auswahl gleichzeitig ändern könnte. Das WPF Control ListBox kennt hierfür die verschiedenen Selection Modes. [33] Beachten müsste man jedoch, wie die Parameter dargestellt werden, wenn sie sich unter den ausgewählten Projekten unterscheiden.

8.1.3 Selbstständige Optimierung

Um möglichst alle Nebenläufigkeitsfehler zu finden, könnte unsere Applikation so erweitert werden, dass die Analyseparameter selbstständig verändert werden, bis keine neuen Fehler mehr entdeckt werden. Diese Idee basiert auf den Ergebnissen aus Abschnitt 7.5.3.
Dazu müsste der Parallel Checker in der Lage sein, zu entscheiden wann er nochmals eine Analyse mit veränderten Parametern durchführen soll und wann er die Analyse abschliessen kann. Durch dieses Feature wäre das Tool nochmals deutlich zuverlässiger beim Auffinden von Nebenläufigkeitsfehlern. Die gewonnene Zuverlässigkeit hat aber je nach Fall lange Laufzeiten zur Folge.

8.1.4 Cloud-Service

Die Idee einer Weblösung aus Abschnitt 6.1 könnte man nochmals aufgreifen. Wie bereits erwähnt wäre man damit plattformunabhängig und der User hätte keinen Aufwand bei der Installation. Bei einer Umsetzung gilt es sich die erwähnten Risiken bewusst zu sein. Gegebenenfalls liesse sich die Idee mit Hilfe einer Sandbox Umgebung umsetzen. Die Analysis und Service Komponente liessen sich einfach für eine Webanwendung wiederverwenden.

Glossar

.NET Core ist die Weiterentwicklung von .NET Framework mit dem Ziel der Plattformunabhängigkeit. 18

.NET Framework ist ein Software Framework zur Entwicklung von Applikationen. Die damit entwickelten Applikationen sind grundsätzlich nur auf Windows lauffähig. 18–20, 35

API Ein Application Programming Interface ist eine Schnittstelle die zwischen Client und Server genutzt wird. Über eine API können Daten angefragte Daten vom Server zum Clientund zu speichernde Daten vom Client zum Server übermittelt werden. 19

Build ist ein Softwareerzeugnis. 26

CLI Ein Command Line Interface ermöglicht die textuelle Steuerung einer Software über das Konsolen-Fenster. 3, 10, 11, 14–18, 20, 30, 36

GUI Ein Graphical User Interface ermöglicht die Steuerung einer Software über grafische Elemente wie Fenster, Buttons etc. 10, 11, 13, 14, 17, 18, 20, 34, 36, 37

IDE Eine integrierte Entwicklungsumgebung für die Herstellung von Software. 10–14

MSBuild ist das offizielle Build Werkzeug von Microsoft um .NET Applikationen zu erzeugen. 26

NuGet ist der Paket Manager für .NET Projekte. Er bietet zahlreiche Bibliotheken und Erweiterungen an, die man für die Entwicklung von .NET-Applikationen nutzen kann. 29, 30

Visual Studio ist eine Entwicklungsumgebung von Microsoft, um Software für C# Programme zu erstellen. 6, 10–12, 14, 16, 25, 28–30

WPF Windows Presentation Framework ist ein Software Framework zur Entwicklung von Applikationen mit grafischen Benutzeroberflächen. 19

XML ist ein Dateiformat zur Speicherung von strukturierten Daten. 12

Abbildungsverzeichnis

1 Deadlock als Graph 8
2 Screenshot vom GUI des Polyspace Code Provers. 13
3 GUI von FindBugs zeigt die gefundenen Fehler. 14
4 Architektur mit plattform-unabhängigem GUI 18
5 Electron in Kombination mit C# 19
6 Architektur mit Windows-GUI und plattformunabhängiger CLI 20
7 Architektur mit GUI und CLI für Windows 20
8 Ablauf der Views 21
9 Startview 22
10 Durchführungs-View 23
11 Resultatview 24
12 Data Race Graph 25
13 Report-Files (Word und JSON) 25
14 Analyse Ablauf 26
15 Verwendetes Concurrency Design 27
16 Gefundene Fehler pro Runde und Projekt 31
17 Akkumulierte Fehler pro Projekt 31
18 Orleans - Einfluss der einzelnen Parameter 32
19 Orleans - Einfluss der einzelnen Parameter 33
20 SignalR - Gefundene Fehler total 39
21 SignalR - Gefundene Fehler 39
22 SignalR - Neu gefundene Fehler pro Runde 40
23 SignalR - Einfluss der einzelnen Parameter 40
24 Orleans - Abweichungen vom Mittelwert 41
25 Orleans - Gefundene Fehler 41
26 Orleans - Neu gefundene Fehler pro Runde 42
27 NLog - Abweichungen vom Mittelwert 43
28 NLog - Gefundene Fehler 43
29 NLog - Neu gefundene Fehler pro Runde 44
30 NLog - Einfluss der einzelnen Parameter 44
31 MSBuild - Abweichungen vom Mittelwert 45
32 MSBuild - Gefundene Fehler 45
33 MSBuild - Neu gefundene Fehler pro Runde 46
34 MSBuild - Einfluss der einzelnen Parameter 46

Tabellenverzeichnis

1 Übersicht der Analyse 14
2 Testprojekte 28
3 Testresultate 29

Codebeispiele

1 Race Condition mit Data Races 7
2 Race Condition ohne Data Races 7
3 Deadlock 8

Literaturverzeichnis

[1] Luc Bläser. „Practical Detection of Concurrency Issues at Coding Time“. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: ACM, 2018, S. 221–231.

[2] *HSR Parallel Checker*. URL: https://parallel-checker.com. (Zugegriffen am 09.11.2019).

[3] Luc Bläser. *Vorlesungs- und Übungsunterlagen Parallele Programmierung*. Hochschule für Technik, Rapperswil. Durchführung 2019.

[4] John Regehr. *Race Condition vs. Data Race*. URL: https://blog.regehr.org/archives/490. (Zugegriffen am 11.11.2019).

[5] Richard C. Holt. „Some Deadlock Properties of Computer Systems“. In: *ACM Comput. Surv.* 4.3 (Sep. 1972), S. 179–196.

[6] *Polyspace*. URL: https://ch.mathworks.com/de/products/polyspace.html. (Zugegriffen am 23.09.2019).

[7] *ThreadSanitizer*. URL: https://clang.llvm.org/docs/ThreadSanitizer.html. (Zugegriffen am 23.09.2019).

[8] *Intel Inspector*. URL: https://software.intel.com/en-us/inspector. (Zugegriffen am 23.09.2019).

[9] *FindBugs*. URL: http://findbugs.sourceforge.net/. (Zugegriffen am 23.09.2019).

[10] *SpotBugs*. URL: https://spotbugs.readthedocs.io/en/latest/. (Zugegriffen am 23.09.2019).

[11] *VS Tools*. URL: https://docs.microsoft.com/en-us/visualstudio/code-quality/?view=vs-2019. (Zugegriffen am 23.09.2019).

[12] *PathFinder*. URL: https://github.com/javapathfinder/jpf-core/wiki. (Zugegriffen am 23.09.2019).

[13] *Polyspace Code Prover*. URL: https://ch.mathworks.com/de/products/polyspace-code-prover.html#features. (Zugegriffen am 23.09.2019).

[14] *Intel Inspector Features*. URL: https://software.intel.com/en-us/inspector/features#threading. (Zugegriffen am 23.09.2019).

[15] *Screenshot vom GUI des Polyspace Code Provers*. URL: https://ch.mathworks.com/de/products/polyspace-code-prover.html. (Zugegriffen am 23.09.2019).

[16] *FindBugs - Browsing Results*. URL: http://findbugs.sourceforge.net/manual/example-details.png. (Zugegriffen am 23.09.2019).

[17] *OWASP Top Ten Project*. URL: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. (Zugegriffen am 10.12.2019).

[18] Andreas Müller. *Vorlesungs- und Übungsunterlagen Automaten und Sprachen*. Hochschule für Technik, Rapperswil. Durchführung 2017.

[19] *Electron*. URL: https://electronjs.org/. (Zugegriffen am 16.09.2019).

[20] *Google Chromium*. URL: https://www.chromium.org/. (Zugegriffen am 16.09.2019).

[21] *EdgeJS*. URL: https://github.com/agracio/electron-edge-js. (Zugegriffen am 16.09.2019).

[22] *Buildalyzer*. URL: https://github.com/daveaglick/Buildalyzer. (Zugegriffen am 16.09.2019).

[23] Dave Glick. *Running A Design-Time Build With MSBuild APIs*. URL: https://daveaglick.com/posts/running-a-design-time-build-with-msbuild-apis. (Zugegriffen am 16.09.2019).

[24] *MSBuild Workspace*. URL: https://gist.github.com/DustinCampbell/32cd69d04ea1c08a16ae5c4cd21dd3a3. (Zugegriffen am 16.09.2019).

[25] *Technology Regions Overview*. URL: https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/technology-regions-overview. (Zugegriffen am 29.11.2019).

[26] *Design-time builds*. URL: https://github.com/dotnet/project-system/blob/master/docs/design-time-builds.md. (Zugegriffen am 20.11.2019).

[27] *Einstieg in Roslyn-Analysen*. URL: https://docs.microsoft.com/de-ch/visualstudio/extensibility/getting-started-with-roslyn-analyzers?view=vs-2019. (Zugegriffen am 4.11.2019).

[28] *SonarScanner for MSBuild*. URL: https://docs.sonarqube.org/latest/analysis/scan/sonarscanner-for-msbuild/. (Zugegriffen am 4.11.2019).

[29] *SonarScanner auf Github*. URL: https://github.com/SonarSource/sonar-scanner-msbuild. (Zugegriffen am 4.11.2019).

[30] *SonarQube Scanner for MSBuild v2.0 released: support for third-party Roslyn analyzers*. URL: https://devblogs.microsoft.com/devops/sonarqube-scanner-for-msbuild-v2-0-released-support-for-third-party-roslyn-analyzers/. (Zugegriffen am 6.11.2019).

[31] *SonarQube Roslyn SDK project*. URL: https://github.com/SonarSource/sonarqube-roslyn-sdk. (Zugegriffen am 6.11.2019).

[32] *ReSharper Command Line Tools*. URL: https://www.jetbrains.com/help/resharper/ReSharper_Command_Line_Tools.html. (Zugegriffen am 4.11.2019).

[33] *ListBox Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.windows.controls.listbox. (Zugegriffen am 18.12.2019).

Anhang

Testresultate SignalR

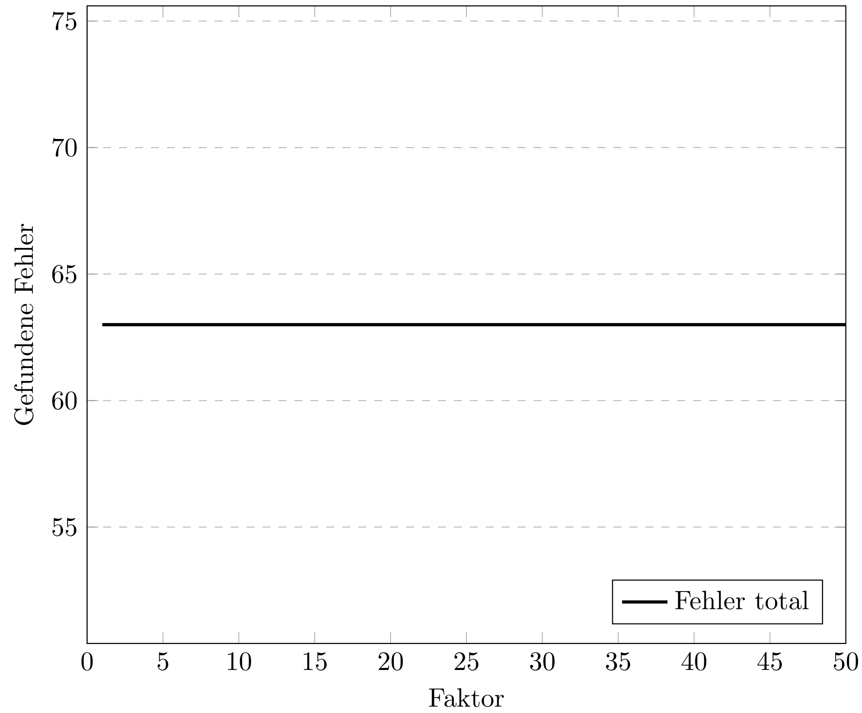

Abbildung 20: SignalR - Gefundene Fehler total

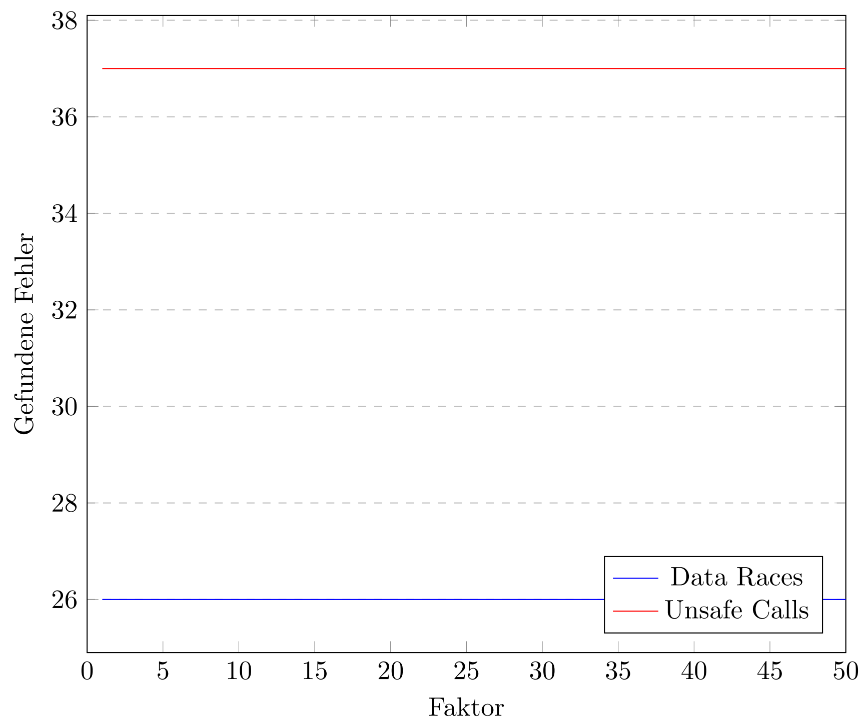

Abbildung 21: SignalR - Gefundene Fehler

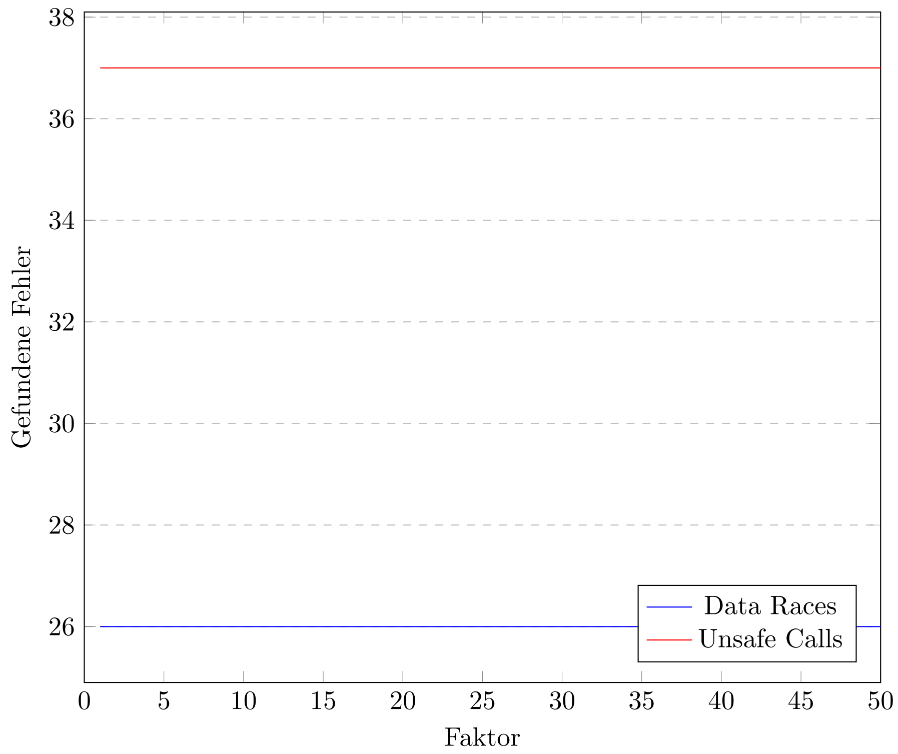

Abbildung 22: SignalR - Neu gefundene Fehler pro Runde

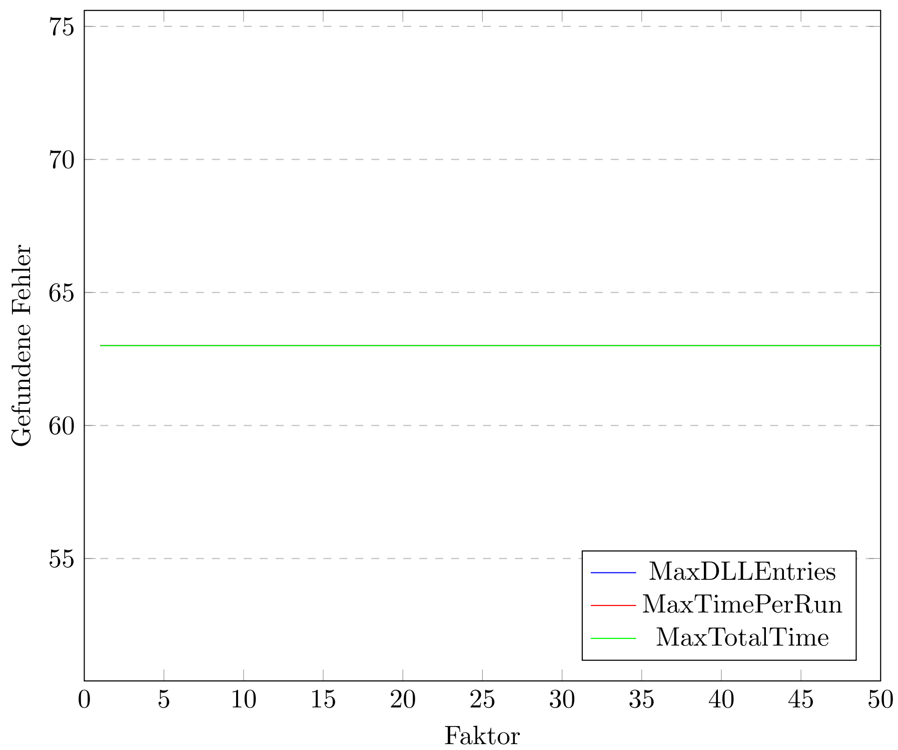

Abbildung 23: SignalR - Einfluss der einzelnen Parameter

Testresultate Orleans

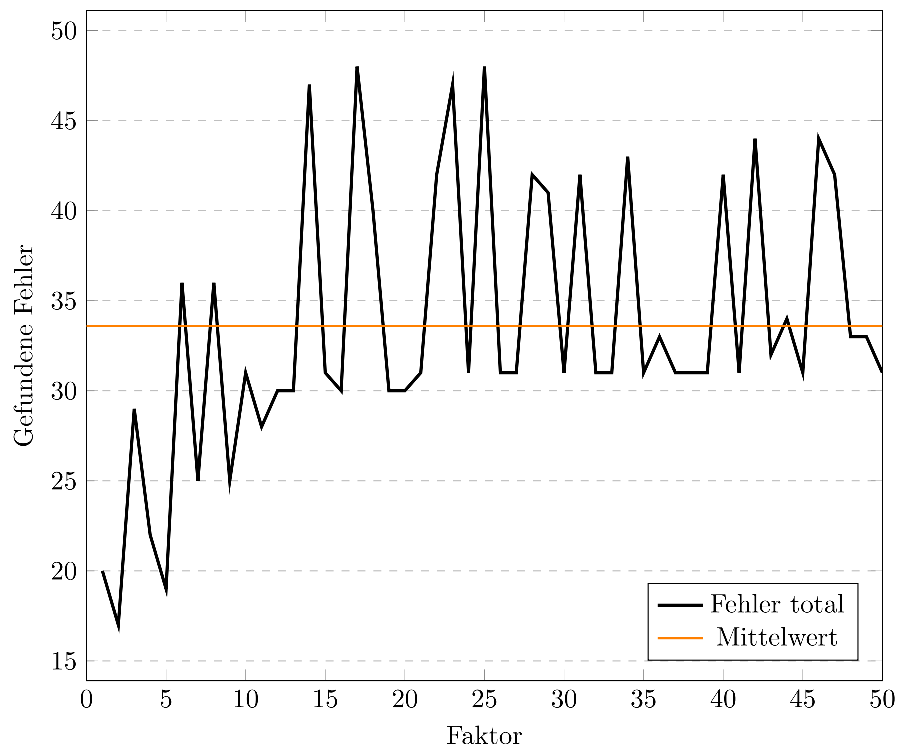

Abbildung 24: Orleans - Abweichungen vom Mittelwert

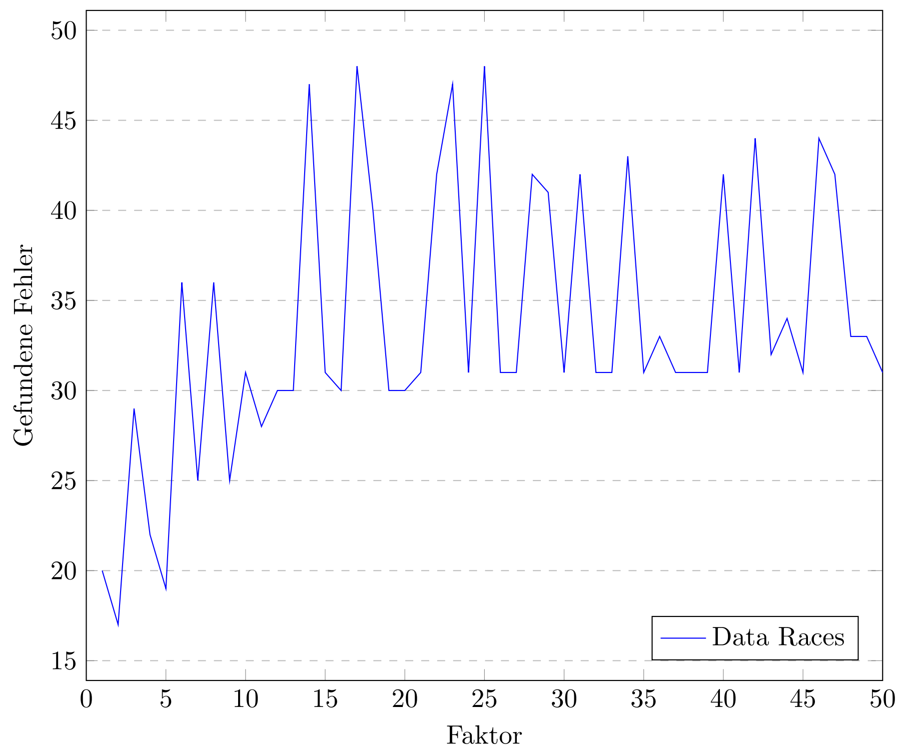

Abbildung 25: Orleans - Gefundene Fehler

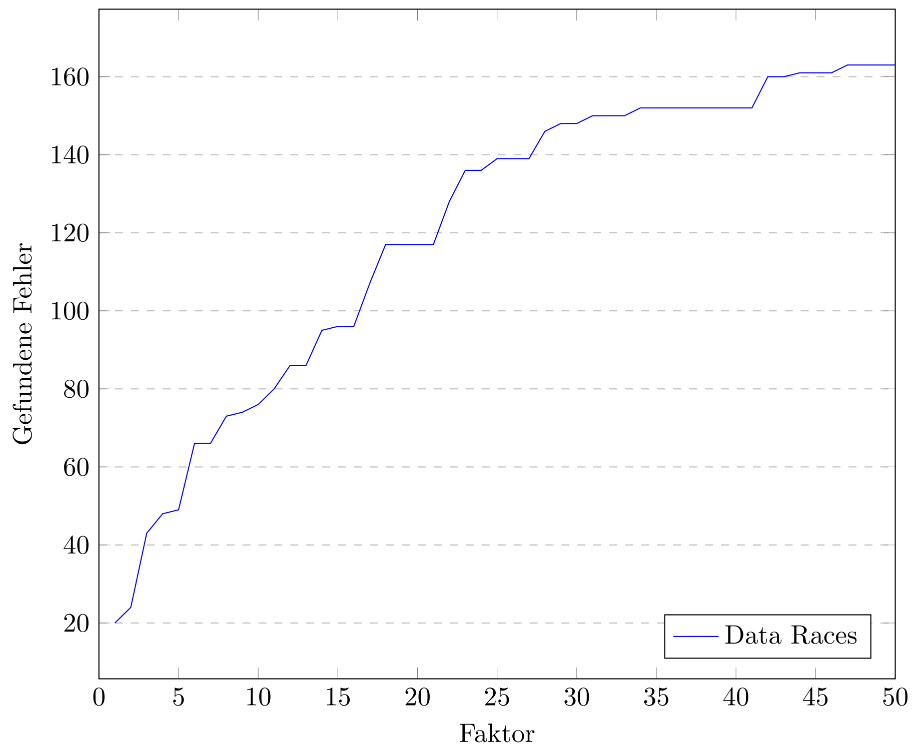

Abbildung 26: Orleans - Neu gefundene Fehler pro Runde

Testresultate NLog

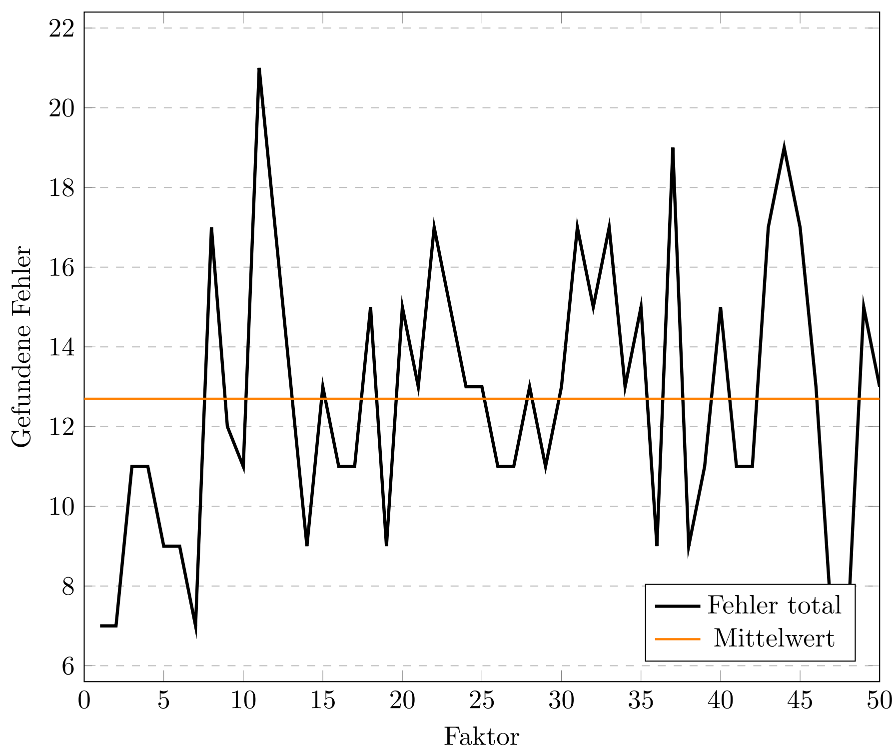

Abbildung 27: NLog - Abweichungen vom Mittelwert

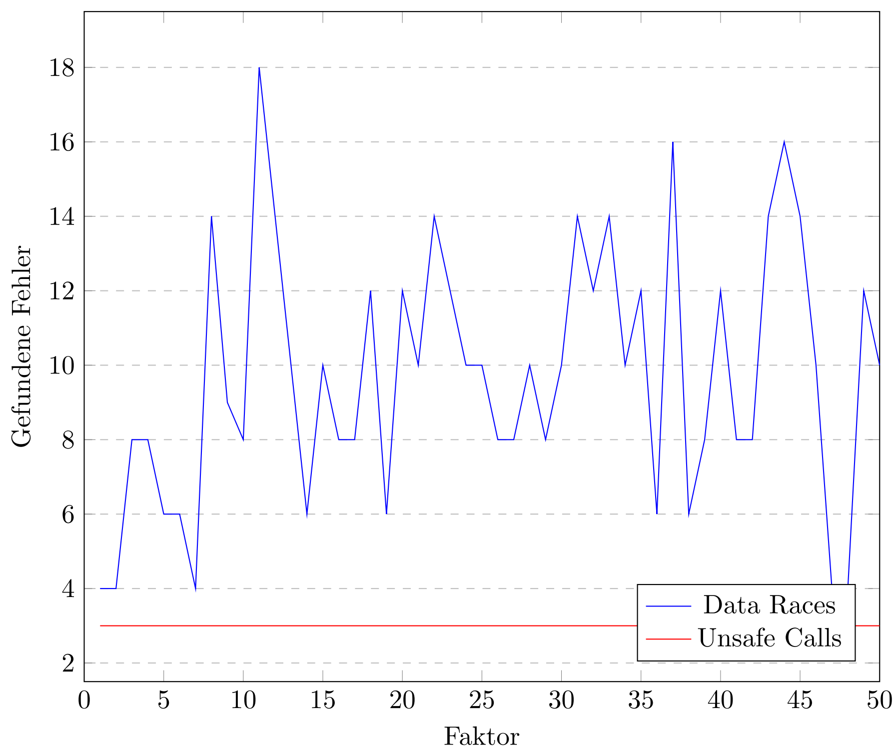

Abbildung 28: NLog - Gefundene Fehler

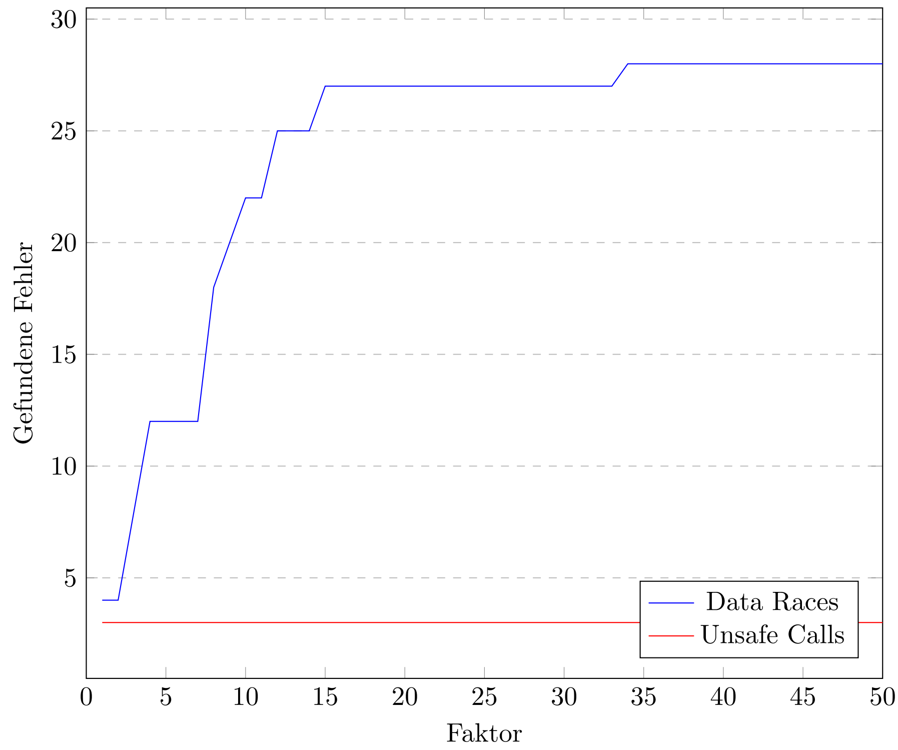

Abbildung 29: NLog - Neu gefundene Fehler pro Runde

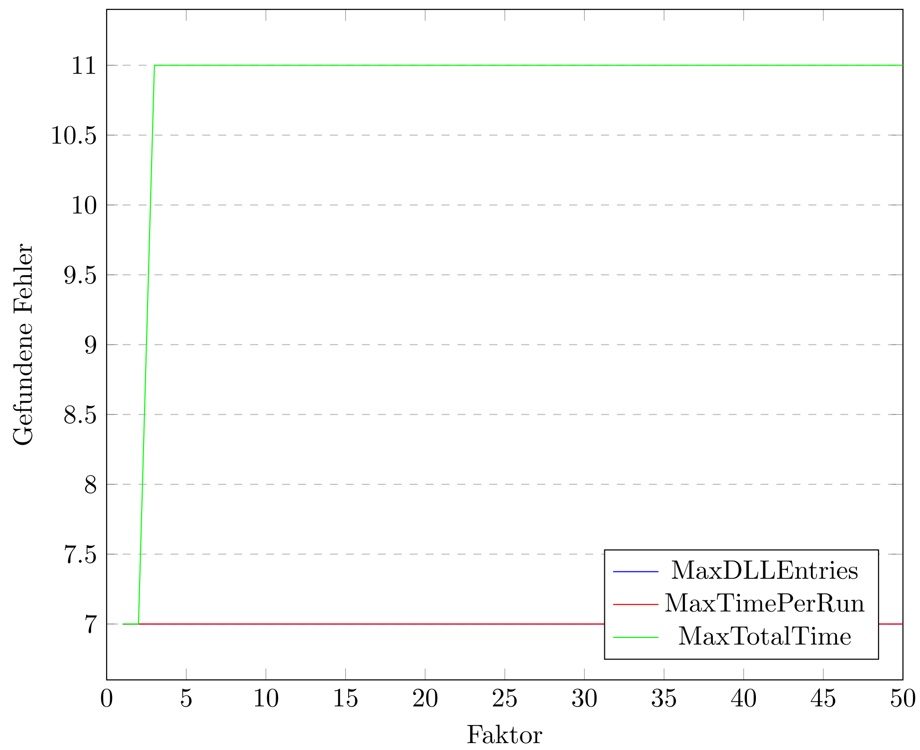

Abbildung 30: NLog - Einfluss der einzelnen Parameter

Testresultate MSBuild

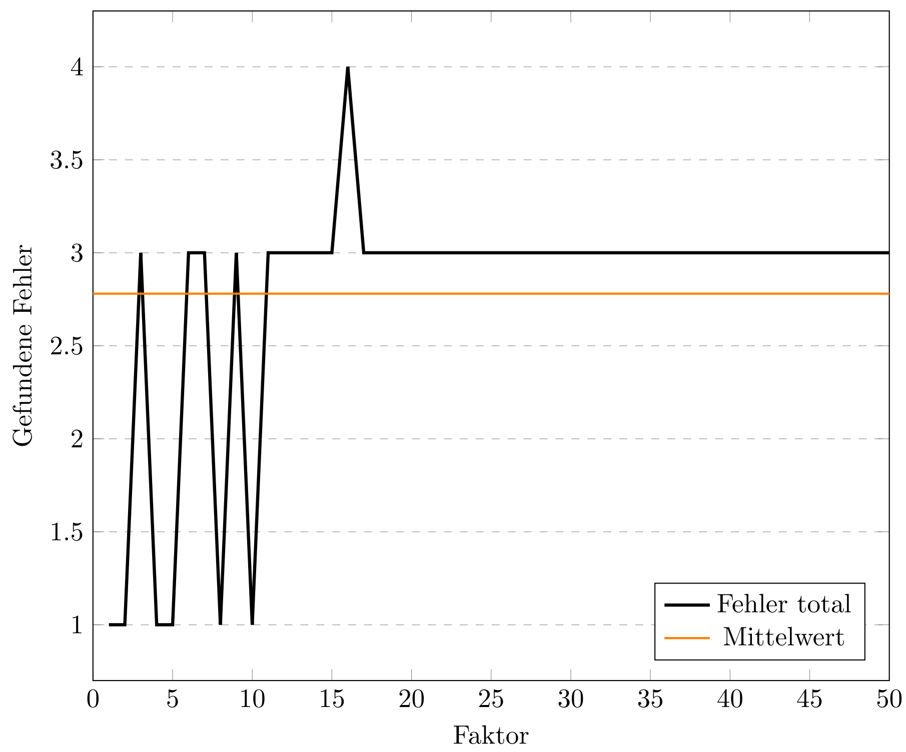

Abbildung 31: MSBuild - Abweichungen vom Mittelwert

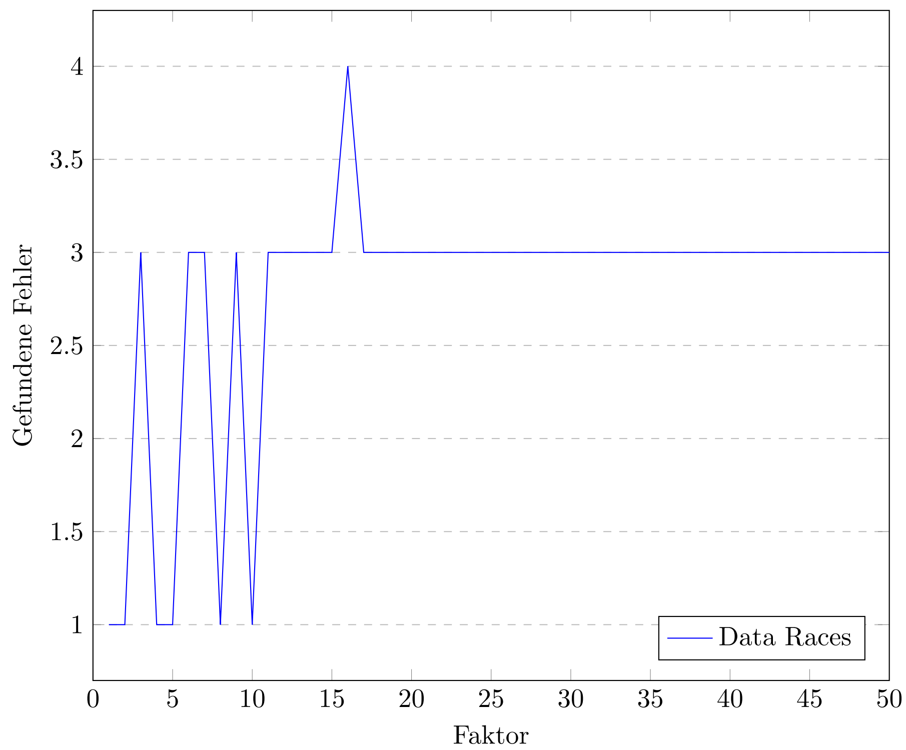

Abbildung 32: MSBuild - Gefundene Fehler

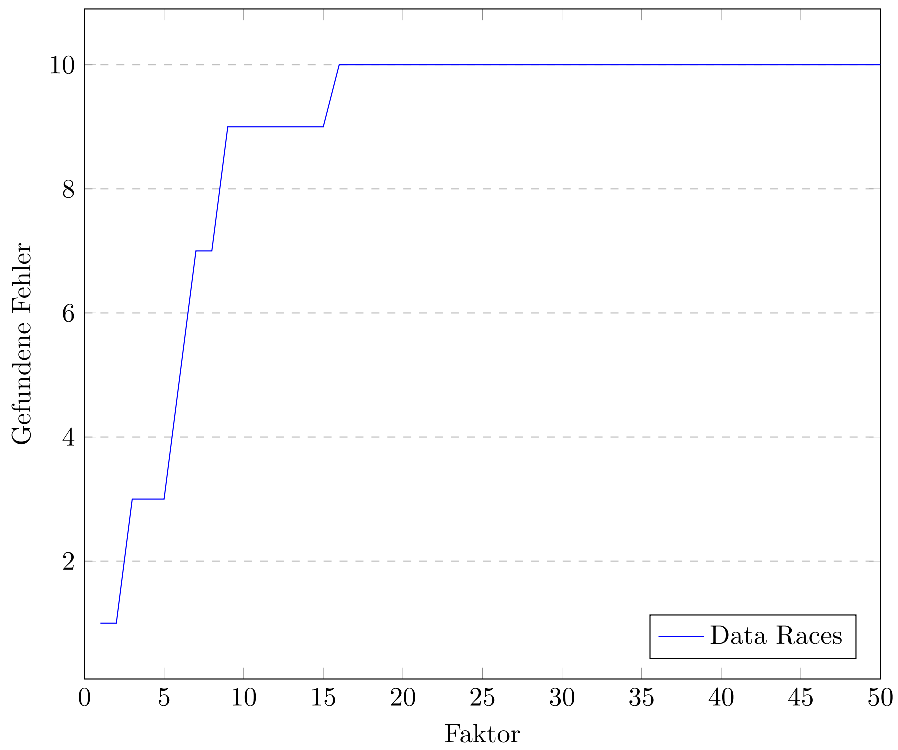

Abbildung 33: MSBuild - Neu gefundene Fehler pro Runde

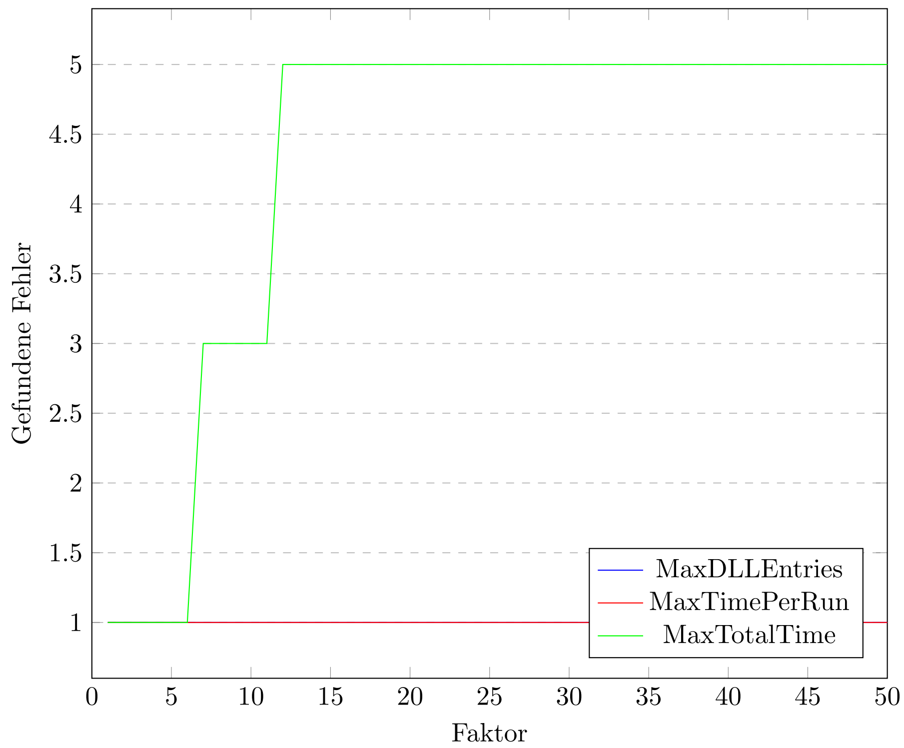

Abbildung 34: MSBuild - Einfluss der einzelnen Parameter

Fehlermeldung SonarQube Roslyn SDK

Mit der Version 2.0 des Tools erstellt. Folgende Fehlermeldung haben wir erhalten:

```
C:\Users\User\Downloads\SonarQube.Roslyn.SDK-2.0>RoslynSonarQubePluginGenerator.exe /a:HSR.ParallelChecker.NuGet
Roslyn Analyzer Plugin Generator for SonarQube 2.0.0.0
Maximum supported Roslyn version: 2.8.0.0
Minimum supported SonarQube version: 6.7 (highest version tested against: 7.3-alpha1)
[DEBUG] Parsed NuGet reference. Id: HSR.ParallelChecker.NuGet, version:
[DEBUG] Fetching NuGet config files...
[DEBUG] Enabled package sources:
[DEBUG]     https://www.nuget.org/api/v2/, machine-wide: False
[DEBUG]     C:\LocalNuGetFeed, machine-wide: False
[DEBUG]     https://dotnet.myget.org/F/roslyn/api/v3/index.json, machine-wide: False
[DEBUG]     C:\Users\User\Downloads\, machine-wide: False
[DEBUG]     C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\, machine-wide: False
Attempting to locate package with id 'HSR.ParallelChecker.NuGet'
[WARNING] [NuGet] An error occurred while loading packages from 'https://dotnet.myget.org/F/roslyn/api/v3/index.json':
Der Remoteserver hat einen Fehler zurückgegeben: (404) Nicht gefunden.
[DEBUG] Number of packages located: 1
[DEBUG] Package versions:
  1.3.0 - latest version

[DEBUG] Using version marked as latest.
Version was not specified. Using version 1.3.0.
[NuGet] Attempting to resolve dependency 'Microsoft.CodeAnalysis.CSharp.Workspaces (= 3.0.0)'.
[WARNING] [NuGet] An error occurred while loading packages from 'https://dotnet.myget.org/F/roslyn/api/v3/index.json':
Der Remoteserver hat einen Fehler zurückgegeben: (404) Nicht gefunden.
[NuGet] Attempting to resolve dependency 'Microsoft.CodeAnalysis.Common (= 3.0.0)'.
[WARNING] [NuGet] An error occurred while loading packages from 'https://dotnet.myget.org/F/roslyn/api/v3/index.json':
Der Remoteserver hat einen Fehler zurückgegeben: (404) Nicht gefunden.
[NuGet] Attempting to resolve dependency 'Microsoft.CodeAnalysis.Analyzers (= 2.6.2-beta2)'.
[WARNING] [NuGet] An error occurred while loading packages from 'https://dotnet.myget.org/F/roslyn/api/v3/index.json':
Der Remoteserver hat einen Fehler zurückgegeben: (404) Nicht gefunden.
//viele Meldungen in diesem Stil ...

[DEBUG]   Duplicate dependency: System.Composition.Hosting version 1.0.31
[DEBUG]   Duplicate dependency: System.Composition.Runtime version 1.0.31
//viele Meldungen in diesem Stil ...

Looking for analyzers in the package...
[DEBUG] Looking for C# analyzers
[DEBUG] Adding AssemblyResolver to current AppDomain assembly resolution.
Loaded assembly: HSR.ParallelChecker.Core, Version=1.3.0.0, Culture=neutral, PublicKeyToken=134bc2708d917588
[DEBUG] Attempting to resolve assembly 'Microsoft.CodeAnalysis.CSharp, Version=3.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35', requested by '{unknown}'
[DEBUG] Assembly located: C:\Users\User\AppData\Local\Temp\.sonarqube.sdk\.nuget
\Microsoft.CodeAnalysis.CSharp.3.0.0\lib\netstandard2.0\Microsoft.CodeAnalysis.CSharp.dll
[DEBUG] Attempting to resolve assembly 'Microsoft.CodeAnalysis, Version=3.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35', requested by '{unknown}'
[DEBUG] Assembly located: C:\Users\User\AppData\Local\Temp\.sonarqube.sdk
\.nuget\Microsoft.CodeAnalysis.Common.3.0.0\lib\netstandard2.0\Microsoft.CodeAnalysis.dll
[DEBUG] Attempting to resolve assembly 'Microsoft.CodeAnalysis, Version=3.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35',
requested by 'Microsoft.CodeAnalysis.CSharp, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35'
[DEBUG] Assembly located: C:\Users\User\AppData\Local\Temp\.sonarqube.sdk
\.nuget\Microsoft.CodeAnalysis.Common.3.0.0\lib\netstandard2.0\Microsoft.CodeAnalysis.dll
No analyzers found in assembly HSR.ParallelChecker.Core, Version=1.3.0.0, Culture=neutral, PublicKeyToken=134bc2708d917588
Loaded assembly: HSR.ParallelChecker.NuGet, Version=1.3.0.0, Culture=neutral, PublicKeyToken=134bc2708d917588
[DEBUG] Attempting to resolve assembly 'System.Collections.Immutable, Version=1.2.3.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a', requested by '{unknown}'
[DEBUG] Assembly located: C:\Users\User\AppData\Local\Temp\.sonarqube.sdk
\.nuget\System.Collections.Immutable.1.5.0\lib\netstandard1.0\System.Collections.Immutable.dll
[DEBUG] Attempting to resolve assembly 'System.Collections.Immutable, Version=1.2.3.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a',
requested by 'Microsoft.CodeAnalysis, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35'
[DEBUG] Assembly located: C:\Users\User\AppData\Local\Temp\.sonarqube.sdk\.nuget
\System.Collections.Immutable.1.5.0\lib\netstandard1.0\System.Collections.Immutable.dll
No analyzers found in assembly HSR.ParallelChecker.NuGet, Version=1.3.0.0, Culture=neutral, PublicKeyToken=134bc2708d917588
Loaded assembly: HSR.ParallelChecker.NuGet, Version=1.3.0.0, Culture=neutral, PublicKeyToken=134bc2708d917588
No analyzers found in assembly HSR.ParallelChecker.NuGet, Version=1.3.0.0, Culture=neutral, PublicKeyToken=134bc2708d917588
[DEBUG] Removed AssemblyResolver from current AppDomain assembly resolution.
[WARNING] No analyzers were found in package: HSR.ParallelChecker.NuGet
[WARNING] Re-run this generator with /recurse if plugins should be generated for the dependencies of this package.
```