Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Grammatikbasierte Generation von Debugging Hypothesen

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Rudolf Eckard Lehner
geboren am: 5.1.2001
geboren in: München

Gutachter/innen: Prof. Dr. Lars Grunske
Prof. Dr. Timo Kehrer

eingereicht am: ..... verteidigt am: .....

Inhaltsverzeichnis

Abstract In diesem Projekt werden wir Eingabedateien für ein Programm als Wörter einer kontextfreien Grammatik darstellen. Mittels eines Fuzzers oder einem ähnlichen Tool werden dann Testeingaben erzeugt, um Crashes zu lokalisieren. Anhand des Ableitungsbaums dieses Worts werden dann weitere Testeingaben erzeugt, mit dem Ziel weitere Crashes zu finden, um eine Fehlerhypothese aufstellen zu können.

1 Einleitung

Testen ist ein wesentlicher Bestandteil bei der Entwicklung neuer Software. Dabei wird versucht, möglichst viele Fehler in der Software zu finden.
Doch mit dem Fund eines Fehlers ist das Testen noch nicht abgeschlossen. Bei mehreren Millionen Zeilen Quellcode ist es in der Regel nicht möglich, anhand nur eines Fehlers die Fehlerursache auszumachen.

Stattdessen muss zunächst versucht werden, den Fehler zu reproduzieren. Einen Ansatz dafür hat man bereits mit dem gefundenen Fehler, der für die weitere Fehlersuche abgewandelt werden kann. Wenn der Fehler ausreichend reproduziert werden konnte, können Hypothesen zur Fehlerursache aufgestellt werden, welche die Fehlerquelle beschreiben und bei der Korrektur helfen.
Wir stellen in diesem Projekt ein allgemein anwendbares Verfahren, die grammatikbasierte Generation von Debugging Hypothesen, vor. Dabei wird jeder Input als ein Wort einer kontextfreien Grammatik aufgefasst. Führt ein Input zu einem Fehler, versuchen wir, den Fehler auf eine bestimmte Ableitung zurückzuführen, die in dem fehlerhaften Wort verwendet wird.
Im Fokus stehen dabei die Fehler, bei denen die zu untersuchende Software ohne Fehlermeldung während Laufzeit außerordentlich beendet wird, im folgenden auch Crashes genannt. Wir richten unser Augenmerk auf syntaktisch korrekte Eingaben und gehen davon aus, dass die zu untersuchende Software syntaktisch falsche Eingaben korrekt behandeln kann.

Wir werden ein Tool präsentieren, das für beliebige Software, deren Input von einer kontextfreien Grammatik dargestellt werden kann, anhand dieser Grammatik und eines gegebenen Inputs, der crasht, auf die Ursache dieser Crashes schließen kann. Dazu benötigt die von uns vorgestellte Software lediglich eine Möglichkeit, erstellte Inputs an der zu prüfenden Software zu testen, um in Erfahrung zu bringen, ob ein Crash vorliegt. Anhand der Grammatik werden wir den Crash reproduzieren und stellen Hypothesen zu seiner Ursache auf. Wir werden Software entwickeln, die diese Aufgaben übernimmt.

Ein kürzlich vorgestelltes Tool [Buc18] ermöglicht die automatisierte Fehlersuche bei LibreOffice mittels einer kontextfreien Grammatik. Mit diesem Tool wollen wir Crashes in LibreOffice finden und mit der von uns entwickelten Software die Ursache

der Crashes ermitteln.

2 Grundlagen

Zunächst werfen wir einen Blick auf die Grundlagen, die für dieses Projekt von Interesse sind. Dazu betrachten wir zuerst das Fuzzing an sich, sowie die Abwandlung des grammatikbasierten Fuzzens. Außerdem werden formelle Grammatiken, insbesondere kontextfreie Grammatiken, sowie die zur Lösung des Wortproblems für kontextfreie Grammatiken benötigten Algorithmen vorgestellt.

2.1 Fuzzing

Fuzzing ist ein Prinzip des automatisierten Softwaretestens [MFS90].
Beim Fuzzing werden mittels eines Tools genannt Fuzzer zufällige Zeichensequenzen generiert. Diese werden dann der zu testenden Software als Eingabe übermittelt. Dabei wird zwangsläufig ein Teil der Eingaben nicht dem Eingabeschema entsprechen, das die Software erwartet.

Für die Verarbeitung der Eingabe gibt es drei mögliche Ergebnisse:

1. *Crash* Das Programm ist abgestützt. Es wurde außerplanmäßig durch einen Fehler während der Laufzeit beendet.

2. *Aufgehängt* Das Programm scheint in einer Endlosschleife festzuhängen. Ein Programm wird als aufgehängt bezeichnet, wenn es bei kontinuierlichem oder bereits beendetem Input weiter läuft, jedoch nach einer festgelegten Zeit immer noch keinen Output produziert hat.

3. *Erfolgreich* Das Programm wurde planmäßig beendet. Hierbei kann noch unterschieden werden, ob der Output für den angegebenen Input korrekt war oder nicht.

Das grundlegende Fuzzing, wie es von Miller et al. [MFS90] beschrieben wurde, wurde im Laufe der Zeit zwar immer weiter entwickelt und es wurden verschiedene leistungsfähigere Tools entwickelt, jedoch stellen Miller et al. rund 30 Jahre später fest, dass ihr anfänglicher Ansatz immer noch eine signifikante Anzahl an Crashes finden kann. [MZH21]

Somit kann Fuzzing auch heute noch als gutes Werkzeug zur Fehlersuche für Software angesehen werden.

2.2 Grammarbased Fuzzing

Im weiteren Verlauf gehen wir davon aus, dass Inputs, die syntaktisch falsch sind, korrekt abgefangen und behandelt werden.
In [BC67] wird die Idee präsentiert, Eingabe für Software mittels einer Grammatik zu beschreiben. Beim grammarbased Fuzzing verwenden wir eine kontextfreie Grammatik (Beschreibung folgt in Kapitel 2.3), um den Input für die Software zu generieren. Die

kontextfreie Grammatik beschreibt genau die Inputs, die für die zu untersuchende Software syntaktisch korrekt sind. In der Regel handelt es sich hierbei um eine unendliche Menge, da sonst keine Grammatik notwendig wäre, sondern jeder mögliche Input getestet werden könnte.
Die Inputs können dann mit einer Kombination aus Fuzzing und der Grammatik erzeugt werden. Der Fuzzer wählt nun nicht mehr willkürlich irgend ein Zeichen aus, sondern eine der zur Verfügung stehenden Ableitungen, die durch Grammatik definiert werden. Den genauen Prozess der Ableitung aus Grammatik beschreiben wir nachfolgend in Kapitel 2.3. Nach Abschluss dieses Prozesses wurde ein syntaktisch korrekter Input für die zu untersuchende Software erstellt. Mit diesem Input kann wie beim normalen Fuzzing ein Test durchgeführt werden, und das Ergebnis kann wie in Kapitel 2.1 klassifiziert werden.

2.3 Formelle Grammatiken

Formelle Grammatiken sind ein Bestandteil der theoretischen Informatik. Sie dienen dazu Sprachen, die eine beliebige Menge von Wörtern sind, einfacher darzustellen. Jede Grammatik G kann durch ein 4-Tupel $G = (V, \Sigma, P, S)$ dargestellt werden [Kö17]. Dabei ist

- V eine endliche Menge an Variablen (auch Nichtterminalsymbole oder Nichtterminale genannt). Variablen werden für die Ableitungsschritte verwendet, dürfen aber nicht im finalen Wort vorkommen.

- Σ die endliche Menge aller Terminalsymbole. Jedes gültige Wort, das aus dieser Grammatik abgeleitet wird, besteht ausschließlich aus Terminalsymbolen.

- $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ die Menge aller Produktionen (auch Ableitungen genannt).
  Man schreibt Ableitungen auch in der Form $A \to B$ wobei $A \in (V \cup \Sigma)^+$ und $B \in (V \cup \Sigma)^*$. A wird linke Seite und B rechte Seite der Regel genannt.
  In jedem Schritt wird von der bis dahin abgeleiteten Satzform $s \in (V \cup \Sigma)^*$ ein Teil ausgewählt, der auf der linken Seite einer Produktion steht und durch den entsprechenden rechten Teil ersetzt.
  Es ist auch zulässig, dass die rechte Seite einer Regel leer bleibt. Dann handelt es sich um eine ε-Ableitung und man schreibt $A \to \varepsilon$.
  Regeln, deren linke Seite gleich ist, kann man auch zusammenfassen. Dann trennt man die rechten Seiten durch einen vertikalen Strich (z.B. für $A \to B, A \to C$ schreibt man auch $A \to B|C$)

- S ist das Startsymbol. Jede Ableitung eines Worts beginnt mit dem Startsymbol.

Durch konsekutive Ausführung von Ableitungen, die mit dem Startsymbol beginnen und bei Satzformen enden, die nur aus Terminalen bestehen, können alle Worte der Sprache, die die Grammatik G beschreibt, abgeleitet werden.

Kontextfreie Grammatiken Um den Aufbau der Eingabedateien darzustellen, sollen im folgenden kontextfreie Grammatiken verwendet werden.
Diese schränken die Menge der Produktionen weiter ein. Für jede Produktion $p \in P$ muss gelten $p \in V \times (V \cup \Sigma)^*$

2.4 Chomsky Normalform (CNF)

Eine Grammatik $G = (V, \Sigma, P, S)$ befindet sich in Chomsky-Normalform, wenn für die Menge der Produktionen gilt $P \subseteq V \times (V^2 \cup \Sigma)$ [Kö17]. Das heißt, jede Ableitung ist entweder der Form $A \to BC$ oder $A \to \alpha$, wobei A, B, C beliebige Nichtterminale und α ein beliebiges Terminal ist.

Jede kontextfreie Grammatik, die keine ε-Ableitungen oder Variablenumbennung enthält, kann in Chomsky Normalform überführt werden. Variablenumbenennungen sind Ableitungen der Form $V \times V$, also alle Regeln, bei denen auf der linken und der rechten Seite jeweils nur ein Nicht-Terminalsymbol steht. Zur Lösung der Probleme werden von Köbler [Kö17] zwei einfache Algorithmen vorgestellt, auf die hier nicht weiter eingegangen werden soll.

Haben wir nun eine kontextfreie Grammatik ohne Variablenumbenennungen und ε-Ableitungen, können wir daraus eine äquivalente Grammatik in Chomsky-Normalform erzeugen [Kö17]. Dazu sind die folgenden zwei Schritte notwendig:

1. Für jedes Terminal α wird ein neues Nichtterminal α' sowie die Regel $\alpha' \to \alpha$ eingefügt. Dann wird jedes Vorkommen von α auf der rechten Seite einer Regel durch α' ersetzt, außer, wenn α alleine auf der rechten Seite einer Regel steht.

2. So lange noch Regeln der Form $A \to B_1 \dots B_k$ mit $k > 2$ existieren wählen wir eine davon aus und entfernen diese. Stattdessen wird ein neues Nichtterminal $A^\#$ sowie die beiden Ableitungen $A \to B_1 A^\#$ und $A^\# \to B_2 \dots B_k$ eingefügt.

Beispiel Betrachten wir die kontextfreie Grammatik

$$G = (\{S, A, B\}, \{a, b, c\}, P, S)$$

mit den Produktionen

$$\begin{aligned} P = \{&S \to aBc|ac \\ &A \to aB|Bc \\ &B \to aA|Sc|ac|b\} \end{aligned}$$

Abbildung 1: Beispielgrammatik

Diese Grammatik besitzt keine Variablenumbenennungen und keine ε-Ableitungen. Somit kann direkt eine äquivalente Grammatik in Chomsky-Normalform gebildet werden.

Im ersten Schritt werden drei neue Nichtterminale A', B', C' sowie die drei Regeln $A' \to a, B' \to b$ und $C' \to c$ hinzugefügt. Außerdem müssen einige Terminale auf rechten Regelseiten ersetzt werden. Somit erhalten wir als Zwischenergebnis die Grammatik:

$$\begin{aligned} G' &= (\{S, A, A', B, B', C'\}, \{a, b, c\}, P, S) \\ P = \{&S \to A'BC'|A'C' \\ &A \to A'B|BC' \\ &B \to A'A|SC'|A'C'|b \end{aligned}$$

Abbildung 2: Zwischenschritt bei der Umwandlung in CNF

Im zweiten Schritt muss lediglich die Regel $S \to A'BC'$ gekürzt werden. Dafür erhalten wir dann die Regeln $S \to A'S^{\#}$ sowie $S^{\#} \to BC'$.

Abschließend erhalten wir eine zu G äquivalente Grammatik in Chomsky-Normalform

$$
\begin{aligned}
G' &= (\{S, S^{\#}, A, A', B, B', C'\}, \{a, b, c\}, P, S) \\
P &= \{S \to A'S^{\#} | A'C' \\
&\quad S^{\#} \to BC' \\
&\quad A \to A'B | BC' \\
&\quad B \to A'A | SC' | A'C' | b \\
&\quad A' \to a, B' \to b, C' \to c\}
\end{aligned}
$$

Abbildung 3: Beispielgrammatik in CNF

2.5 CYK-Algorithmus

Der CYK-Algorithmus dient der Lösung des Wortproblems von kontextfreien Sprachen. Beim Wortproblem gilt es herauszufinden, ob ein gegebenes Wort x zu einer Sprache L gehört.
Für kontextfreie Sprachen in Chomsky-Normalform ist das Wortproblem mit dem CYK-Algorithmus in $\mathcal{O}(n^3)$ entscheidbar [Kö17].
Zusätzlich kann parallel dazu ein möglicher Ableitungsbaum für das untersuchte Wort ermittelt werden.

Eingabe Ein Wort x, für das die Zugehörigkeit zur Sprache L einer Chomsky-Normalform Grammatik G geprüft werden soll
Ausgabe Zugehörigkeit von x zu L sowie einen möglichen Ableitungsbaum, falls $x \in L$

Der CYK-Algorithmus arbeitet mit einer $n \times n$ Matrix M wobei n die Anzahl der Terminale in dem Wort x ist. Zunächst muss die erste Zeile in dieser Matrix gefüllt werden. In jedes Feld $M_{1,k}$ werden alle Nichtterminale eingetragen, die direkt zum k-ten Terminal x_k in x abgeleitet werden können. Also stehen in jedem Feld $M_{1,k}$ dann alle Nichtterminale, die zum Teilwort $x' = x_k$, das an der k-ten Stelle von x beginnt und 1 Terminal lang ist, abgeleitet werden können.

Auf dieser Grundlage wird dann Zeile für Zeile gefüllt. In der zweiten Zeile werden in die Felder $M_{2,k}$ dann alle Nichtterminale eingetragen, die zu dem zwei Terminale langen Teilwort $x' = x_k x_{k+1}$ von x, das an der Position k beginnt, abgeleitet werden können.
Hier spielt die Chomsky-Normalform nun eine entscheidende Rolle. Jedes Nichtterminal kann entweder zu genau zwei Nichtterminalen oder genau einem Terminal abgeleitet werden. Alle Terminalableitungen sind bereits in der ersten Zeile erfasst. Also kann x' nochmals in zwei Teilwörter (in diesem Fall sind nur die Teilwörter x_k und x_{k+1} möglich) aufgeteilt werden.

Nun gilt es in der vorherigen Zeile nachzuschlagen, aus welchen Nichtterminalen die beiden Teilwörter abgeleitet werden können und welche Paare aus diesen Nichtterminalen ableitbar sind.
Für die Teilung gibt es bei der Länge zwei nur eine Möglichkeit, da wir ε-Ableitungen ausgeschlossen hatten. Im weiteren Verlauf müssen dann aber alle möglichen Teilungen beachtet werden.

Wenn die gesamte Matrix gefüllt ist, muss geprüft werden, ob das Startsymbol der Grammatik im Feld $M_{n,1}$ steht. Wenn dies der Fall ist, so ist $x \in L(G)$, und der Ableitungsbaum lässt sich anhand von M leicht bestimmen.

Kurz In jedem Feld $M_{i,k}$ stehen alle Nichtterminale, aus denen das Teilwort $x_k \dots x_{k+i-1}$ von x, das an der Stelle k beginnt und i lang ist, abgeleitet werden kann.

Beispiel Betrachten wir noch einmal die Beispielgrammatik aus Kapitel 2.4.
Jetzt soll überprüft werden, ob das Wort $x = aaabc \in L(G)$ ist. Wir erhalten für dieses Wort die Matrix M:

	a	a	a	b	c
1	$\{A'\}$	$\{A'\}$	$\{A'\}$	$\{B', B\}$	$\{C'\}$
2	$\{\}$	$\{\}$	$\{A\}$	$\{S^{\#}, A\}$	
3	$\{\}$	$\{B\}$	$\{S, B\}$		
4	$\{A\}$	$\{S^{\#}, A\}$			
5	$\{S\}$				

Abbildung 4: CYK-Tabelle des Beispielwortes

Die Felder rechts von der Gegendiagonalen bleiben dabei komplett frei, da von dieser Stelle aus startende Teilwörter über das Ende von x hinaus gehen würden.
Da das Startsymbol S im Feld $M_{5,1}$ steht gehört das Wort *aaabc* zur Sprache der Beispielgrammatik. Aus der Matrix ergibt sich außerdem ein Ableitungsbaum für das Wort *aaabc*

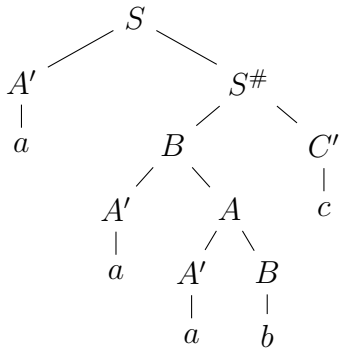

Abbildung 5: Ableitungsbaum des Beispielwortes

3 Verwandte Arbeit

Alhazen Alhazen ist eine, ebenfalls auf kontextfreien Grammatiken beruhende, Methode zur automatisierten Untersuchung des Verhaltens von Software unter bestimmten Umständen, z.B. bei Crashes. [KHSZ20]
Alhazen benutzt eine kontextfreie Grammatik. Dabei werden Teilen in den Worten, sowie in deren Ableitungen, numerische Werte zugewiesen, um ein Modell der untersuchten Software aufzustellen. Dieses Modell soll das Verhalten der Software beschrieben. Es wird eine Feedback-Schleife verwendet, um Testfälle zu generieren. So werden mit jedem Durchlauf weitere Testfälle generiert, um die bisherige Hypothese zu präziseren. Die Hypothese wird durch einen Entscheidungsbaum dargestellt, der anhand der zuvor zugewiesenen numerischen Werte das Verhalten der untersuchten Software beschreibt. Dieser beruht allerdings ausschließlich auf den zuvor zugewiesenen numerischen Werten und ist unabhängig von dem Ableitungsbaum des fehlerhaften Inputs.

Beispiel Betrachten wir einen Teil von einem Taschenrechner, der die Quadratwurzel und den Tangens einer Zahl berechnen kann. Dazu könnten in der Grammatik folgende Regeln existieren:

```
1 Funktion -> sqrt(Zahl)
2 Funktion -> tan(Zahl)
```

Exemplarisch wird dieses Programm crashen, wenn die Quadratwurzel einer negativen Zahl berechnet werden soll.
Alhazen kann nun feststellen, dass es sich bei dem Nichtterminal *Zahl* um eine Zahl handelt, die abgeleitet wird. Diesem Nichtterminal wird dann der numerische Wert der Zahl, zu der das Nichtterminal *Zahl* abgeleitet wird, zugewiesen.
Anschließend werden Testfälle generiert und daraus folgender Entscheidungsbaum entwickelt:

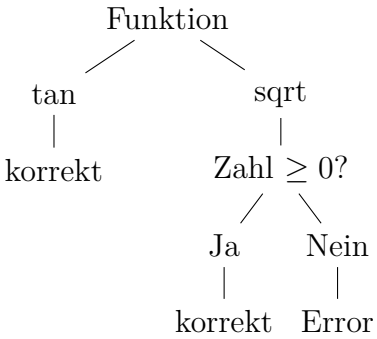

Abbildung 6: Ermittelter Entscheidungsbaum für unser Beispiel

Wir präsentieren einen alternativen Entwurf, in dem wir das Programmverhalten, insbesondere Crashes, anhand des Auftretens bestimmter Ableitungsregeln in der

Ableitung des Wortes, das als fehlerhafter Input ermittelt wurde, festmachen werden. Abschließend bietet sich ein Vergleich mit Alhazen an.

Eingabeüberdeckung Primäres Ziel der entwickelten Software ist nicht das Aufspüren von Crashes, sondern das Identifizieren der Ursache selbiger. Daher wird bei der Entwicklung der Software kein besonderes Augenmerk auf die Wahl des Fuzzers gelegt werden.
Das zielorientierte Aufspüren von Fehlern anhand einer Grammatik erfordert jedoch Coverage Mechanismen, sodass auch für ungünstig gebaute Grammatiken möglicherweise schwer erreichbare Ableitungsschritte sowie die daraus resultierenden Teilworte ausreichend getestet werden.
Außerdem ist Wachstumskontrolle für Wörter notwendig, sodass beim Fuzzing keine Endlosschleife auftritt oder bestimmte Teile des Inputs nur sehr selten auftauchen [HZ19].
Havrikov und Zeller beschreiben ein Verfahren, das alle möglichen Pfade der Länge k für Inputs betrachtet. Sie nennen eine Menge von Inputs dann überdeckend, wenn jeder der möglichen Pfade der Länge k in mindestens einem der Ableitungsbäume der Inputs auftritt.
Wir haben diese Methode in unserem Dummyfuzzing nicht implementiert, raten jedoch dazu, den Fuzzer, der später unseren Dummy ersetzen wird, nicht ohne diese oder eine ähnliche Methode zu verwenden.

Grammarbased Fuzzing von LibreOffice Daniel Bucher [Buc18] beschreibt in einer Bachelorarbeit eine Möglichkeit, mittels einer kontextfreien Grammatik Dateien im OpenDocument Format zu fuzzen. Anschließend wird versucht, die gefuzzte Datei mit LibreOffice zu öffnen. Ziel dabei ist es eine Datei zu finden, die einen Crash auslöst.
In der Arbeit berichtet er von einem Testumfang von 60 Dateien. Bei keiner dieser hat er einen Crash erzeugen können.
Dieser Ansatz wird im vorliegenden Projekt fortgesetzt und in Kapitel 4.1 weiter ausgeführt.

4 Vorgehensweise

Ziel des Projekts war es, mittels eines Tools zum Fuzzen von LibreOffice Crashes zu finden, die die Referenz zu realen Anwendungen darstellen soll, und Software zu entwickeln, die mit einer kontextfreien Grammatik, einem Crash-Orakel und einem Fuzzer Debugging-Hypothesen aufstellt, um den Fehler, der den Crash auslöst, beheben zu können.
In diesem Abschnitt betrachten wir die Suche nach LibreOffice Crashes und werfen einen Blick auf die Funktion der Software und den Entwicklungsprozess. Die entwickelte Software kann unter
`https://scm.cms.hu-berlin.de/lehnerru/bachelorarbeit`
eingesehen und heruntergeladen werden.

4.1 Fuzzing von LibreOffice

Zuerst sollten mithilfe eines von Daniel Bucher beschriebenen Tools zum Fuzzing von LibreOffice Crashes gefunden werden [Buc18], um eine Verbindung zu realer Software herzustellen.

Dabei stellten wir fest, dass die präsentierte Software nicht darauf ausgelegt war, von ihrem .jar File aus ausgeführt zu werden. Das lag an im Code fest eingetragenen Pfaden. Das musste angepasst werden. Anschließend konnte mit dem Fuzzing begonnen werden.

Das Tool bietet zwei Modi an, einen Zufallsmodus und einen, der probabilistische Informationen erfordert. Diese Informationen sollen bewirken, dass bestimmte Ableitungen häufiger oder weniger häufig von der Software ausgewählt werden. Zum Erstellen der probabilistischen Informationen waren Dateien mitgeliefert, die zuvor abgearbeitet werden mussten. Anschließend sollte es möglich sein, mittels der erstellten Probabilistik zu fuzzen. Es war jedoch gänzlich undurchsichtig, wo diese Informationen gespeichert werden und wie diese dann aufgerufen wurden. Es gab außerdem keine Meldung, dass im probabilistischen Modus mit der Probabilistik gearbeitet wurde, sodass wir uns vorerst auf den Zufallsmodus beschränkt haben.

Im Zufallsmodus wurden mit dem Tool fortlaufend LibreOffice Dateien erzeugt und getestet. Während der Laufzeit des Programms wurde fortlaufend die Anzahl der Knoten ausgegeben, aus denen die XML-Datei besteht, die den Datenanteil von LibreOffice Dateien ausmacht. Während des Ausführens des Tools stellten wir fest, dass unmittelbar nach dem Start Dateien mit Knotenanzahlen im Bereich zwischen 1000 und 10000 Knoten erstellt wurden. Jedoch pendelte sich im Verlauf bei 10 bis 20 Knoten pro Datei ein.
Das steht im Widerspruch zu der Auswertung von Bucher, die eine durchschnittliche Knotenanzahl von ca. 300 suggeriert sowie 50 % der Ergebnisse ungefähr im Bereich zwischen 200 und 600 Knoten angibt. [Buc18]

Ungeachtet dessen haben wir das Tool vorerst weiter benutzt. Nach mehreren erfolglosen Wochen und rund drei Millionen erfolglosen Testfällen haben wir diesen Ansatz allerdings aufgegeben und die Softwareentwicklung ohne die Verbindung zu LibreOffice fortgesetzt.

4.2 Grammatikdarstellung

Zunächst wurde eine Möglichkeit geschaffen, kontextfreie Grammatiken darzustellen. Dazu wurde mit der Generalisierung *Variablen* begonnen und diese in die Subklassen *Terminale* und *NonTerminale* unterteilt. Abweichend von der Beschreibung in Kapitel 2.3 benutzen wir hier den Begriff Variable für Terminale und Nichtterminale als Generalisierung, sodass in der weiteren Entwicklungsphase Terminale und Nichtterminale gemeinsam in Mengen oder Listen dargestellt werden können. Das ist zum Beispiel bei der Darstellung von rechten Regelseiten oder der Ableitung in Satzformen von Vorteil.
Variablen dürfen beliebig benannt werden, mit der Einschränkung, dass im Namen kein Leerzeichen, Apostroph oder Zirkumflex vorkommen darf. Zwei Variablen vom selben Typ (*Terminal* oder *NonTerminal*) werden dabei als gleich angesehen, wenn ihr Name gleich ist.

Instanzen von Variablen werden anschließend verwendet, um Regeln darzustellen. Jede Regel benötigt genau ein *NonTerminal* als linke Seite und besitzt eine geordnete Liste von beliebig vielen *Variablen*, die die rechte Seite darstellen.

Damit ist die Grundlage für die Darstellung von kontextfreien Grammatiken gelegt. Diese werden als Instanzen der Klasse *CFG* dargestellt. Um eine Instanz zu erstellen wird lediglich eine Liste mit den Namen der Nichtterminale, eine mit den Namen der Terminale, der Name des Startsymbols sowie eine Liste mit den Ableitungsregeln benötigt. Dabei darf zusätzlich kein *Terminal* den gleichen Namen wie ein *NonTerminal* haben.

Grammatikeingabe Für die Eingabe der Grammatiken wurde eine Schnittstelle geschaffen, die die Eingabe sowohl von der Konsoleneingabbe (*System.in*), als auch aus einer Datei ermöglicht. Gleichermaßen wird es ermöglicht eine Grammatik auf die Konsole auszugeben (*System.out*) oder in eine Datei zu schreiben.
Damit Regeln, Terminale, Nichtterminale und das Startsymbol unterschieden werden können muss die Eingabe die folgende Form einhalten:

1. In jeder Zeile steht entweder ein Terminal, ein Nichtterminal oder eine Ableitungsregel

2. Am Anfang jeder Zeile steht ein Deskriptor, der anzeigt, was in der Zeile folgt:

S: gibt an, dass das Startsymbol folgt

T: weist auf ein folgendes Terminal hin

N: zeigt an, dass ein Nichtterminal folgt

R: zeigt eine Ableitungsregel an

3. Das Startsymbol wird wie ein Nichtterminal behandelt. Wenn ein zweites Mal ein Startsymbol auftritt, wird das erste überschrieben. Das vorherige Startsymbol besteht als *NonTerminal* weiter.

4. Startsymbol, Terminale und Nichtterminale dürfen kein Leerzeichen, Anführungszeichen oder Zirkumflex beinhalten

5. Ableitungsregeln bestehen aus dem Namen eines Nichtterminals, gefolgt von -> und abschließend einer Sequenz von beliebig vielen Terminalen und Nichtterminalen jeweils durch Leerzeichen separiert. Es darf immer nur eine Regel pro Zeile stehen.

6. Das Einlesen endet, sobald *ENDE* alleinstehend in einer Zeile gelesen wurde, oder wenn kein Input mehr gelesen werden konnte.

Beispiel Werfen wir erneut einen Blick zurück auf die Beispielgrammatik aus Kapitel 2.4. Die Grammatik kann durch folgende Eingabe eingelesen werden:

```
1  S: S
2  N: A
3  N: B
4  T: a
5  T: b
6  T: c
7  R: S -> a B c
8  R: S -> a c
9  R: A -> a B
10 R: A -> B c
11 R: B -> a A
12 R: B -> S c
13 R: B -> a c
14 R: B -> b
```

Abbildung 7: Eingabedatei für die Beispielgrammatik

CNFGrammar Die Umwandlung in Chomsky-Normalform ist für umfangreiche Grammatiken sehr umständlich und händisch nicht durführbar. Zum Beispiel besitzt die Grammatik für OpenDocument Format-Dateien ca. 5000 Regeln [Buc18], und

für deren Hypothesenbildung soll dieses Projekt die Grundlage stellen. Daher wurde gleich die Möglichkeit implementiert, Objekte von *CFG* in die Klasse *CNFGrammar*, die Grammatiken in Chomsky-Normalform darstellt, umzuwandeln. Dabei stellt *CNFGrammar* eine Spezifizierung von *CFG* dar. Somit bleiben alle Funktionen, die auf *CFG* anwendbar sind, auch für *CNFGrammar* anwendbar. Das Erstellen von *CNFGrammar* Instanzen ist direkt und ausschließlich mit Objekten von *CFG* möglich. So stellen wir sicher, dass die grundlegenden Bedingungen für kontextfreie Grammatiken erfüllt sind, bevor in CNF umgewandelt wird.
Für die Bezeichnung der Ersatz-Nichtterminale, die für jedes Terminal erstellt werden, wird der Terminal-Name verwendet und um ein Apostroph erweitert. Da es keine zwei gleichnamigen Terminale sowie ebenfalls kein gleichnamiges Nichtterminal geben darf und in den Variablennamen Apostrophe nicht erlaubt sind, führt das nicht zu Mehrdeutigkeit.
Analog wird für den Ersatz bei rechten Seiten mit mehr als 2 Nichtterminalen das Nichtterminal auf der linken Seite um ein Zirkumflex (ˆ) erweitert.

Beispiel Betrachten wir erneut die in Kapitel 4.2 eingelesene Grammatik. Diese lassen wir nun in eine *CNFGrammar* umwandeln und uns dann wieder ausgeben. Dabei erhalten wir folgende Datei:

```
1  S: S
2  N: S^
3  N: c'
4  N: a'
5  N: b'
6  T: b
7  T: c
8  T: a
9  R: b' -> b
10 R: c' -> c
11 R: S -> a' c'
12 R: B -> b
13 R: A -> a' B
14 R: B -> a' A
15 R: B -> a' c'
16 R: a' -> a
17 R: S -> a' S^
18 R: A -> B c'
19 R: B -> S c'
20 R: S^ -> B c'
```

Abbildung 8: Ausgabe der Beispielgrammatik in CNF

4.3 Lösung des Wortproblems

Vorbereitung Zunächst wird eine Möglichkeit Ableitungsbäume darzustellen benötigt. Da dank der Chomsky-Normalform ein Nichtterminal zu höchstens zwei Variablen abgeleitet werden kann, eignen sich für die Darstellung Binärbäume. Die Klasse *BinParseTree* ist zu diesem Zweck erschaffen worden. Ein *BinParseTree* bedarf immer genau einer Wurzel, rechter und linker Teilbaum können ebenfalls *BinParseTrees* sein, sind aber optional und können auch *null* sein. Bei Ableitungen zu Terminalen wird in der Software in der Regel für das Terminal der linke Teilbaum benutzt, der rechte Teilbaum wird leer gelassen und ist somit *null*.

Implementation des CYK-Algorithmus Um den Ableitungsbaum zu einem gegebenen Wort zu ermitteln, der später bei der Fehlerreproduktion benötigt wird, wurde der CYK-Algorithmus implementiert. Gleichzeitig dient er der syntaktischen Fehlerüberprüfung und kann einen syntaktischen Fehler am Eingabewort ermitteln. Das Verhalten beim Auftreten von syntaktischen Fehlern wird in diesem Projekt nicht untersucht.
Bei der Implementation dieses Projekts wird die im CYK Algorithmus erstellte Tabelle nicht nur mit den ermittelten Nichtterminalen gefüllt, sondern mit Paaren aus *NonTerminal* und *BinParseTree*, sodass gleich ein Zugriff auf den entsprechenden Ableitungsbaum möglich ist.

4.4 Fuzzer und Orakel

Für die Tests werden zumindest ein Fuzzer und ein Crash Orakel benötigt. Da zum aktuellen Zeitpunkt keine Fehlerbeispiele aus realen Implementationen vorlagen, mussten hier Prototypen erstellt werden. Wir werfen hier einen Blick auf die Funktion der Prototypen und erläutern, welche Funktionen bei der Ersetzung implementiert werden müssen.

Fuzzer Bei dem im Prototyp verwendeten Fuzzer handelt es sich um eine sehr einfache Version.
Der Fuzzer wird mit einer kontextfreien Grammatik instanziiert und generiert sich dann intern eine entsprechende Chomsky-Normalform Grammatik sowie einen Zufallsgenerator.

Die grundlegende Funktion des Fuzzers beginnt mit dem Startsymbol und läuft dann auf der Satzform von links nach rechts. Dabei werden *Terminals* übersprungen und nur *NonTerminals* ausgewählt. Für das ausgewählte *NonTerminal* wird dann:

1. Die Menge an Regeln in der Grammatik herausgesucht, die aus dem gewählten Nichtterminal ableiten

2. Eine von den im ersten Schritt gesammelten Regeln wird zufällig ausgewählt

3. Das gewählte Nichtterminal wird durch die rechte Seite der gewählten Regel ersetzt.

Anschließend wird in der neu erschaffenen Satzform von der gleichen Stelle aus weiter gesucht, an der soeben das Nichtterminal ersetzt wurde.

Bemerkung Der Dummyfuzzer benutzt im zweiten Schritt zur Auswahl eine gleichverteilte Pseudozufallszahl. Eine Erstellung bzw. Benutzung von probabilistischen Informationen wird nicht vorgenommen.

Des Weiteren bietet der Fuzzer die Möglichkeit, eine bestimmte Menge an Nichtterminalen beim Ableiten mit einer bestimmten Wahrscheinlichkeit zu ignorieren. Dann wird in dem oben beschriebenen Ablauf ein 0. Schritt vorab hinzugefügt:

0. Prüfe, ob das gewählte Nichtterminal zu ignorieren ist. Wenn ja, dann ignoriere es mit der gegebenen Wahrscheinlichkeit.

Dies ermöglicht es uns im weiteren Verlauf in Kapitel 4.5 bestimmte Ableitungen zu erhalten. Außerdem ist der Dummyfuzzer in der Lage mit einer bestimmten Satzform zu beginnen und diese weiterzuentwickeln.
Nebenbei bietet der Fuzzer auch die Möglichkeit einen *BinParseTree* in eine Satzform umzuwandeln, was ebenfalls in Kapitel 4.5 benötigt wird.

Ein nachträglich eingebauter Fuzzer muss dementsprechend die folgenden drei Hauptfunktionen besitzen:

- Standardmäßiges Ableiten eines Worts der Grammatik aus dem Startsymbol sowie aus bereits vorhandenen Satzformen der Grammatik
- Fortsetzen einer in einem unvollständigen Ableitungsbaum beschriebenen Ableitung
- Ableitung von Satzformen der Grammatik mit Ignorieren bestimmter Nichtterminale zu angegebener Wahrscheinlichkeit

Orakel Um das Auftreten von Crashs beim Testen mit den Prototypen zu bestimmen, wurde ein Dummy Orakel erschaffen. Dieses Orakel fungiert während der Tests als die zu untersuchende Software und kann bei Befragung mittels eines booleschen Wertes simulieren, ob das Programm gecrasht ist.
Das Orakel bietet zwei Möglichkeiten Crashs zu erzeugen.
Einerseits können festgelegte Muster, beispielsweise bestimmte Worte oder Zeichenkombinationen in den Worten, als Fehlerursachen angenommen werden, bei deren Auftreten das Orakeln einen Fehler auslöst. Diesen Modus bezeichnen wir im folgenden als den Patternmodus.
Andererseits ist es möglich, Fehler in einzelnen *BinParseTrees* zur Ursache zu erklären, sodass, sobald dieser Baum in der Ableitung als Teilbaum auftritt, ein Crash

angenommen wird. Dieser Baum kann aus einer einzelnen Regel hervorgehen, lässt jedoch auch die Kombination mehrerer Regeln oder komplexerer Bäume zu. Diesen Modus werden wir im Folgenden den Treemodus nennen.
Im weiteren Verlauf dieser Arbeit gehen wir von der Verwendung des Orakels als Instrument zur Fehlersuche aus.
Das Orakel muss aber nicht zwangsläufig zum ermitteln von Fehlern genutzt werden. Vielmehr bietet sich auch die Option, ein bestimmtes Programmverhalten an Eingaben festzumachen.

Auch unser Dummy Orakel muss später durch eine echte Version eines Orakels ersetzt werden. Dabei wird von dem ersetzenden Orakel lediglich erwartet, dass es einen Input, der durch den Fuzzer generiert wurde, an der Software testet und anschließend angeben kann, ob das Programm gecrasht ist.

4.5 Hypothesenbildung

Damit sind nun alle Grundlagen für die Bildung der Fehlerhypothesen geschaffen. Einzig ein Wort, das einen Crash auslöst, muss durch den Anwender gefunden werden. Dieses Wort werden wir nachfolgend auch initialen Fehler oder initiales Fehlerwort nennen.
Mit der Aufstellung der Fehlerhypothesen beschäftigt sich die *Tester* Klasse. Ihre Arbeitsweise werden wir im Folgenden erläutern.

Reproduktion Als Erstes wird der Ableitungsbaum des initialen Fehlerwortes gebildet. Dazu wird die Implementation des CYK-Algorithmus, wie in Kapitel 4.3 beschrieben, verwendet.
Für die Reproduktion betrachten wir Mengen von Teilbäumen des Ableitungsbaumes des initialen Fehlerwortes. Um diese Mengen zu bilden haben wir zwei Ansätze entwickelt:

ohne Grammatikeinfluss Zunächst wird keine Rücksicht auf die Grammatik genommen. Jeder mögliche Teilbaum wird untersucht. Das führt je nach Wortgröße auch zu einer sehr großen Menge durchzuführenden Tests.

Grammatikbasiert Da bei den ersten Tests mit dem ersten Ansatz schnell klar wurde, dass die Menge der zu untersuchenden Teilbäume nicht handhabbar ist, wurde nach einer weiteren Lösung gesucht. Dabei spielt folgende Art von Teilbäumen eine entscheidende Rolle:

Sollte es in den Ableitungsregeln außer der Regel $A \to BC$ keine weiteren Regeln der Form $A \to BX$ oder $A \to YC$ geben, so wird an dieser Stelle kein Teilbaum entfernt, sondern der in Abbildung 9 abgebildete Teilbaum kann als eine Einheit betrachtet werden. Jede Ableitung, die nur mit einem Teil dieses Teilbaums arbeitet, würde

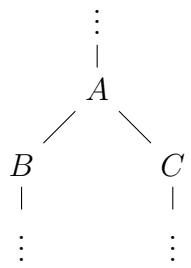

Abbildung 9: für Kombination relevante Teilbäume

den Teilbaum sowieso wieder produzieren, da lediglich eine Regel dafür vorhanden ist.

Anschließend wird mit dem Fuzzer jeder der generierten Teilbäume in weitere Wörter eingebaut und somit der Fehleranteil für den untersuchten Teilbaum ermittelt. In wie viele weitere Wörter jeder einzelne zu untersuchende Fehlerbaum eingebaut werden soll, kann durch den Nutzer angegeben werden. Standardmäßig sind 100 Teilbäume eingestellt. Diese Anzahl werden wir auch im weiteren Verlauf des Projekts nutzen.
Am Ende dieses Schrittes ist für jeden Teilbaum eine Fehlerrate ermittelt worden.

Reduktion Durch die ausgedehnte Sammlung der Teilbäume werden wir zwangsläufig nicht nur die Fehlerquellen selbst, sondern auch all die Bäume erhalten, in denen der Fehlerbaum enthalten ist.
Nehmen wir an, der folgende Baum führt jedes Mal zu einem Crash

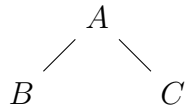

Abbildung 10: fehlerhafter Baum

so werden alle Bäume, in denen dieser Baum enthalten, ist diesen Crash erben und ebenfalls immer crashen

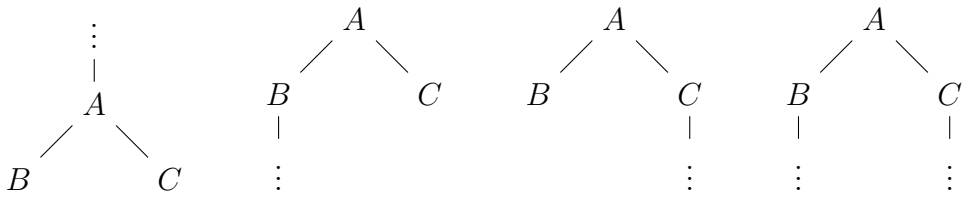

Abbildung 11: Bäume die den Crash erben

Um diese Vererbung von Crashes zu handhaben, führen wir den Reduktionsschritt ein.
Dabei werden alle Bäume paarweise miteinander verglichen. Wenn dabei ein Baum A einen Baum B enthält, aber keinen größeren Fehleranteil als Baum B besitzt, so wird A entfernt. Hat A jedoch einen größeren Fehleranteil, so behalten wir beide Bäume und nehmen an, dass A mehrere Fehlerquellen begünstigt. Das kann unter anderem daran liegen, dass mehrere fehlerhafte Ableitungen häufig mit Baum A einhergehen. Alle Bäume aus Abbildung 11 werden also mit dem Baum aus Abbildung 10 verglichen. Sofern die Quote, mit der bei einem Baum aus Abbildung 11 Crashes festgestellt wurden, nicht größer ist als die, die zum Baum aus Abbildung 10 gehört, so wird der entsprechende Baum aus Abbildung 11 aus der Ergebnismenge entfernt.

Ergebnisse Abschließend werden noch die Ergebnisse der Untersuchung präsentiert. Dazu werden dem Benutzer die Teilbäume vorgeschlagen, die die höchsten Fehlerquoten aufweisen. Besitzt ein solcher Teilbaum die Höhe eins, besteht also nur aus einer Wurzel und höchstens zwei Kindern der Wurzel, so werden alle Regeln, die zu diesem Teilbaum abgeleitet werden können, als potentiell fehlerhaft angegeben.
Im Falle eines größeren Teilbaums wird keine Regel ausgegeben und behauptet, dass es sich um einen komplexeren Fehler handelt oder der Teilbaum häufig in verschiedenen fehlerhaften Ableitungen vorkommt.

4.6 Verwendung unserer Software

Unsere Software kann direkt mit dem Tester.jar-File von der Kommandozeile ausgeführt werden. Das Tester.jar File ist unter folgendem Link erhältlich.
`https://scm.cms.hu-berlin.de/lehnerru/bachelorarbeit/-/blob/main/GrBasedDebugHypotheses/target/Tester.jar`

Der Aufruf erfolgt mittels

```
1 $ java -jar Tester.jar [Grammar] [Modus] [#Steps] [GrBased]
```

Der Tester erwartet dabei folgende Eingaben:

- [*Grammatik*] - eine Grammatikdatei, die, wie in Kapitel 4.2 beschrieben, formatiert ist.
- [*Fehlermodus*] - den Fehlermodus, der gewählt wird. Hier wird entweder *tree* oder *pattern* erwartet. Dementsprechend werden die, für das Testen zufällig erstellten Fehler, generiert. Wie diese ausgewählt werden erläutern wir nachfolgend in Kapitel 5.
- [*#Wiederholungen*] - optionaler Parameter. Die Anzahl der Versuche, die für jeden zu untersuchenden Fehlerbaum durchgeführt werden soll. Standardmäßig liegt diese bei 100.

- [$GrBased$] - optionaler Parameter. Wenn der grammatikbasierte Modus zum Aufteilen des initialen Fehlers verwendet werden soll, muss dieses Argument ein + und das 3. Argument vorhanden sein. Andernfalls wird der normale Modus zum Aufteilen verwendet.

Um unsere Software für Tests an realen Programmen verwenden zu können bedarf es weiteren Anpassungen. Dazu müssen der Fuzzer und das Orakel durch Funktionen zum Erstellen und Testen von Inputs (genauere Anforderungen siehe Kapitel 4.4) ersetzt werden. Des weiteren muss in der Tester Klasse die Generation der Beispielfehler in Zeilen 109 bis 172 entfernt werden.
Anschließend kann das .jar File neu gebaut werden und ist zur Verwendung bereit.

5 Versuche mit der Software

Da mit der Software zum Fuzzen von LibreOffice, wie in Kapitel 3 beschrieben, unerwarteterweise keine Crashes gefunden werden konnten, müssen wir uns bei den Versuchen, die wir mit der Software durchführen wollen, auf selbst erstellte Grammatiken sowie den Dummyfuzzer und das Dummyorakel beschränken.
Daher handelt es sich bei diesen Versuchen eher um eine Verifizierung der Funktion der erstellten Software als um reale Tests.

5.1 Verwendete Grammatiken

Zunächst stellen wir die zwei Grammatiken, die für das Testen erschaffen wurden, vor. Beide Grammatiken stellen Input dar, wie ihn primitive Software haben könnte. Wir verwenden das in Kapitel 4.4 vorgestellte Orakel als die untersuchte Software. Die gültigen Inputs werden durch die folgenden zwei Grammatiken dargestellt:

Taschenrechner Die erste Grammatik stellt einen Input dar, wie ihn ein Taschenrechner für ganze Zahlen haben könnte:

$$G = (\{Start, Term, Operator, Zahl, ZahlNN\},\\ \{+, -, *, :, (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, P, Start)$$

mit den Produktionen

$$\begin{aligned} P = \{&Start \to Term\ Operator\ Term\\ &Term \to (\ Term\ Operator\ Term\ ) \mid (\ -\ Term\ ) \mid ZahlNN\ Zahl\\ &Zahl \to Zahl\ Zahl \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\\ &Operator \to\ + \mid - \mid * \mid :\\ &Term \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\\ &ZahlNN \to 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\} \end{aligned}$$

Abbildung 12: Taschenrechnergrammatik

Sprache Die zweite verwendet Grammatik orientiert sich an der deutschen Sprache, ist jedoch nicht vollständig. Um sie übersichtlicher zu halten, wurden Beugungen, Großschreibung oder ähnliche Anpassungen weggelassen:

$$
\begin{aligned}
G = (\{&Start, Aussage, AAusage, Auforderung, Frage, Subjekt, Objekt, Verb,\\
&Fragewort, Temporaladverb, derSubst, dieSubst, dasSubst\},\\
\{&., !, ?, und, oder, Wo, Wer, Was, Wie, Wieso, vorgestern, gestern,\\
&heute, morgen, übermorgen, essen, fliegen, fahren, lesen, spielen, reden,\\
&sitzen, messen, reiten, üben, ich, du, er, sie, es, wir, ihr, der, die, das,\\
&Spieler, Spielerin, Fahrer, Fahrerin, Saft, Gurke, Bahn, Kind, See, Fluss,\\
&Wasser, Müsli, Geschäft, Karussel\}, P, Start),
\end{aligned}
$$

mit den Produktionen:

$$
\begin{aligned}
P = \{Start &\rightarrow Aussage\ . \mid Auforderung\ ! \mid Frage\ ?\\
Aussage &\rightarrow Subjekt\ Verb \mid Subjekt\ Verb\ Temporaladverb \mid\\
&\quad AAussage\ und\ Aussage \mid AAusage\ oder\ Aussage\\
AAussage &\rightarrow Subjekt\ Verb \mid Subjekt\ Verb\ Temporaladverb\\
Frage &\rightarrow Verb\ Subjekt \mid Fragewort\ Verb\ Subjekt\\
Auforderung &\rightarrow Verb\ Subjekt\\
Fragewort &\rightarrow Wo \mid Wer \mid Was \mid Wie \mid Wieso\\
Temporaladverb &\rightarrow vorgestern \mid gestern \mid heute \mid morgen \mid übermorgen\\
Verb &\rightarrow essen \mid fliegen \mid fahren \mid fliegen \mid lesen \mid spielen \mid reden \mid\\
&\quad sitzen \mid messen \mid reiten \mid üben\\
Subjekt &\rightarrow ich \mid du \mid er \mid sie \mid es \mid wir \mid ihr \mid\\
&\quad der\ derSubst \mid die\ dieSubst \mid das\ dasSubst\\
Objekt &\rightarrow ich \mid du \mid er \mid sie \mid es \mid wir \mid ihr \mid\\
&\quad der\ derSubst \mid die\ dieSubst \mid das\ dasSubst\\
derSubst &\rightarrow Spieler \mid Fahrer \mid Saft \mid See \mid Fluss\\
sieSubst &\rightarrow Spielerin \mid Fahrerin \mid Gurke \mid Bahn\\
dasSubst &\rightarrow Kind \mid Müsli \mid Karussel \mid Geschäft \mid Wasser\}
\end{aligned}
$$

Abbildung 13: stark vereinfachte Grammatik der deutschen Sprache

5.2 Versuchsaufbau

Für die Tests wird das Orakel in den beiden Modi, dem Treemodus und dem Patternmodus, verwendet.

Treemodus Das Orakel kann bei der Angabe von Fehlerbäumen Crashes auslösen, wenn diese im Wort auftreten. Dabei werden von der gegebenen Grammatik zuerst drei Regeln zufällig ausgewählt und deren Ableitungsbäume dem Orakel als fehlerhaft übergeben.

Anschließend werden mit dem Fuzzer so lange Wörter erstellt, bis mit dem Orakel ein Fehler festgestellt wurde. Da Fuzzing nicht das primäre Ziel dieses Projekts ist, wurde hier eine Grenze bei 10000 Versuchen gezogen. Wenn dann immer noch kein Crashwort gefunden wurde, werden erneut drei Regeln zufällig ausgewählt und von vorne mit der Suche eines Crashes begonnen. Sobald ein Crash gefunden wurde, wird das auslösende Wort an den in Kapitel 4.5 beschriebenen Algorithmus übergeben.

Dabei wird das gleiche Wort sowohl vom grammatikbasierten, als auch vom ohne Grammatikeinfluss arbeitenden Teilbaumerstellungsansatz bearbeitet. Für jeden der mit den beiden Modi erstellten Teilbäume werden dann 100 Wörter erstellt, in denen diese Teilbäume enthalten sind. Diese werden dann mit dem Orakel getestet, um die Fehlerraten zu ermitteln. Das Ergebnis, die Laufzeit sowie die Anzahl der untersuchten Teilbäume werden festgehalten.
Einen Versuch bezeichnen wir dann als erfolgreich, wenn einer der drei vorab festgelegten Teilbäume mit der höchsten Fehlerquote ermittelt wird. Sonst werten wir den Test als negativ.

Bei den ersten Versuchen wurde festgestellt, dass die zur Verfügung stehende Rechenleistung nicht ausreichend war, um viele Mengen mit vielen Teilbäumen laufen zu lassen. Da das Aufsplitten in Teilbäume verhältnismäßig schnell geht und die Rechenleistung die Grenze von 10000 Teilbäumen vertretbar erscheinen ließ, wurde diese Grenze gezogen, und alle Fehlerwörter, die in mehr als 10000 Teilbäume aufgespalten wurden, werden ignoriert.

Patternmodus Bei der Angabe von Fehlermustern werden Inputs unabhängig von ihrem Ableitungsbaum durch das Orakel als Fehler ausgegeben. Das könnte zu anderem Verhalten führen und soll ebenfalls untersucht werden. Dabei werden für die Initialisierung des Orakels zufällig ein oder zwei Terminale gewählt.

Wenn nur ein Terminal gewählt wurde, so wird jedes Wort, das dieses Terminal umgeben von beliebigen Zeichenkombinationen enthält, als fehlerhaft markiert.
Wenn zwei Terminale gewählt wurden, wird zuerst eine Reihenfolge festgelegt. Jedes Wort, das mit einer beliebigen Zeichenfolge beginnt, dann das erste Terminal gefolgt von einer weiteren beliebigen Zeichenkette sowie dem zweiten Terminal enthält und

zum Abschluss eine dritte beliebige Zeichenkette beinhaltet, soll crashen.

Nachdem das Orakel initialisiert wurde, wird, analog zum Treemodus, nach einem initialen Crashwort gesucht. Anschließend wird der Fehler reproduziert und Hypothesen aufgestellt.
Das Überprüfen der Hypothesen wird hier allerdings händisch durchgeführt. Eine Automatisierung ist zwar generell möglich, dafür könnte unter anderem ein Subgraph-Matching Algorithmus verwendet werden. Dies hätte aber den zeitlichen Rahmen unseres Projekts gesprengt.
Beim händischen überprüfen der Fehlerhypothesen des Patternmodus wird darauf Wert gelegt, dass der Ansatz des Fehlers gefunden wurde und somit der Fehler nach Behebung der einen Quelle zumindest reduziert werden kann.

Nehmen wir für die in Abbildung 12 präsentierte „Taschenrechner“-Grammatik an, dass jedes Wort, das das Terminal 5 enthält, Crashes auslöst. Wird dann von unserer Software zumindest eine der drei Regeln $ZahlNN \to 5$, $Zahl \to 5$ oder $Term \to 5$ als Fehlerquelle mit der höchsten Wahrscheinlichkeit erkannt, so werten wir diesen Test als positiv.

6 Ergebnisse

Wie in Kapitel 5.2 beschrieben haben wir nun die Tests mit unserer Software durchgeführt. Die Ergebnisse werden wir nachfolgend präsentieren und diskutieren.

6.1 Ergebnisse

In den folgenden Tabellen werden wir einige Abkürzungen verwenden:

TR: mit der Grammatik „Taschenrechner“
Spr: mit der Grammatik „Sprache“
Gr.based: grammatikbasierte Baumaufteilung (siehe Kapitel 4.5)

Zuerst betrachten wir die Erfolgsquote unserer Versuche.

		Treemodus		Patternmodus	
		normal	Gr.based	normal	Gr.based
	Anz. Tests	2079		50	
TR	... positiv	2079	2079	48	48
	... 100 % Quote	2079	2079	48	48
	Anz. Tests	4950		50	
Spr	... positiv	4950	4950	48	47
	... 100 % Quote	4950	4950	48	47

Tabelle 1: Erfolgsquote bei den Versuchen

Wir stellen fest, dass der Treemodus in allen getesteten Fällen so arbeitet, wie er soll. Zu jedem untersuchten Crash konnte mindestens eine Fehlerhypothese korrekt aufgestellt werden. Im Patternmodus stellten wir einige falsche Fehlerhypothesen fest. Wir haben die Ergebnisse es Patternmodus händisch ausgewertet. Einige Spezialfälle, wie wir diese Fälle gewertet haben, und mögliche Ursachen, die zu diesen Fällen führen, präsentieren wir nachfolgend in Kapitel 6.2. Außerdem haben wir festgestellt, dass, unabhängig vom Orakelmodus (Tree oder Pattern), in allen Fällen, in denen die Fehlerhypothese korrekt aufgestellt wurde, sie auch eine 100 % Fehlerquote besaß.

Wir haben ebenfalls eine Laufzeitanalyse zum Vergleich der Modi, mit denen der ursprüngliche Fehlerbaum in seine Teilbäume aufgeteilt wird (erläutert in Kapitel 4.5), durchgeführt.

	Gr.based ist...	Treemodus	Patternmodus
TR	...schneller bei	2059 (99 %)	50 (100 %)
	...im Schnitt schneller	68 %	73 %
Spr	...schneller bei	4711 (95 %)	49 (98 %)
	...im Schnitt schneller	43 %	51 %

Tabelle 2: Laufzeitvergleich

Wir erkennen hier, dass die Verwendung der Grammatik beim Aufteilen des Ableitungsbaumes des ursprünglichen Fehlerwortes in den meisten Fällen zu einer Beschleunigung führt, jedoch nicht immer schneller ist. In unseren Ergebnissen sieht es so aus, als sei der Zeitgewinn, der durch den grammatikbasierten Modus zum Aufteilen der Teilbäume entsteht, abhängig von der verwendeten Grammatik. Also könnte es sein, dass unser Tool für bestimmte Software gut geeignet ist und für andere weniger gut. Dies gilt es weiter zu untersuchen.

Als drittes haben wir noch die Anzahl an Bäumen, die durch die grammatikbasierte Aufteilung, des ursprünglichen Fehlerbaumes, eingespart wurden, aufgezeichnet.

	Gr.based...	Treemodus	Patternmodus
TR	...reduziert bei	2079 (100 %)	50 (100 %)
	...im Schnitt kleiner	68 %	69 %
Spr	...reduziert bei	4950 (100 %)	50 (100 %)
	...im Schnitt kleiner	44 %	49 %

Tabelle 3: Vergleich der Anzahl der zu untersuchenden Teilbäume

Dabei stellen wir fest, dass in jedem Fall weniger Bäume untersucht werden mussten. Wie wir vorab bereits feststellten, führt das aber nicht dazu, dass weniger korrekte Fehlerhypothesen aufgestellt wurden. Somit ist der grammatikbasierte Modus in unseren Augen klar besser, als die Sammlung aller möglichen Teilbäume. Wie schon bei der Untersuchung der Laufzeiten (vgl. Tabelle 2) deutet auch hier alles darauf hin, dass das Einsparungspotenzial von der Grammatik abhängt.

6.2 Auswertung

Aus Mangel an einem realen Beispiel können wir mit den Versuchen nur zeigen, dass unsere Software so arbeitet, wie wir sie konzipiert haben. Eine Vergleich der Performance mit anderen Ansätzen, die das gleiche Ziel verfolgen, wie unter anderem Alhazen [KHSZ20], ist mit den hier ermittelten Daten nicht möglich.

Allgemeine Feststellungen Unsere erste Feststellung schon während der Überwachung der Tests war, dass die Anzahl der zu untersuchenden Teilbäume nicht

gleichmäßig verteilt zu sein scheint. Vielmehr ist diese Anzahl in unregelmäßigen Abständen gestaffelt. Diese Staffelung hängt anscheinend lediglich von der Grammatik selbst und dem Modus ab, in dem die Teilbäume erstellt werden. Wie zu erwarten war, haben die beiden Error-Modi des Orakels, *Tree* und *Pattern*, keinen Einfluss auf die Anzahl der untersuchten Teilbäume. Anzahl der zu untersuchenden Bäume änderte sich nur, wenn wir eine andere Grammatik oder den anderen Teilbaumerstellungsmodus gewählt haben.

Generell werten wir die durchgeführten Tests als einen Erfolg, denn mit Ausnahme weniger Fehlschläge im Patternmods konnte bei jedem Test eine in Richtung der Lösung führende Fehlerhypothese aufgestellt werden (siehe Tabelle 1). Dazu müssen wir aber auch ergänzen, dass die Tests aufgrund fehlender Rechenleistung nicht für sehr große Bäume gemacht wurden, und nur auf einer idealen Testumgebung.

Der grammatikbasierte Modus kam bei nahezu allen Testfällen mit weniger Workload (vgl. Tabelle 3) aus und war deutlich schneller (siehe Tabelle 2). Aus den Daten, die wir ermittelt haben, geht hervor, dass die Grammatik das limitierende Feature sein könnte. So konnte bei der „Taschenrechner"-Grammatik eine deutlich größere Reduktion und deutlich mehr Speed-Up erzielt werden als bei der „Sprache"-Grammatik. Da es sich bei den verwendeten Grammatiken aber um sehr einfache, lediglich für die Tests erstellte Beispielgrammatiken handelt, muss diese Vermutung weitergehend überprüft werden.

Feststellungen zum Patternmodus Während der Auswertung der Tests, die im Patternmodus durchgeführt wurden, sind uns einige weitere Besonderheiten aufgefallen. Generell gilt es zu sagen, dass die Hypothese bei den Patterns, bei denen nur das Auftreten eines Wortes zu einem Crash geführt hat, sehr präzise waren und dieses in allen Fällen auch richtig erkannt wurde. Sobald es kombinierte Fehlerquellen gab, deren Aufbau wir in Kapitel 5.2 unter dem Punkt Patternmodus beschreiben, waren nicht alle Hypothesen schlüssig, sodass der Fehler nicht direkt erkannt werden konnte. Einige Spezialfälle, die wir beobachteten werden wir hier noch weiter ausführen:

Die Implementierung der Patterns ist so erfolgt, dass es nicht zwingend erforderlich ist, dass das gewählte Terminal separat stehen muss. Es darf auch Teil eines anderen Terminals sein.

Beobachtung 1 Wir überprüften eine Hypothese mit dem initialen Fehlerwort W (wir verwenden _ um Leerzeichen darzustellen)

$$W = du\_messen\_\text{ü}bermorgen\_oder\_ihr\_lesen\_.$$

Das Orakel sollte bei allen Worten der Form $*_1$ *der* $*_2$ einen Crash mitteilen. Dabei dürfen $*_1$ und $*_2$ beliebige Zeichenkombinationen sein. So kam es dazu, dass bei dem

ursprünglichen Fehlerwort für die Kombinationen $*_1$ und $*_2$ festgelegt wurde:

$*_1 = du_messen_$übermorgen$_o$
$*_2 = \_ihr\_lesen\_$

Dadurch wurde W als erstes Fehlerwort gewählt. Das führte dazu, dass das Wort *oder* hauptsächlich untersucht wurde. Das Wort *der* hat zwar auch Fehler ausgelöst, da sich aber keine Ableitung des Wortes *der* in den zu untersuchen Teilbäumen befand, aus denen letztendlich auch die Fehlerhypothesen gebildet werden. Konnte auch keine der Hypothesen auf das Wort *der* hinweisen. Als Fehlerhypothese erhielten wir dann unter anderem die Ableitung von $oder' \rightarrow oder$, eine Ableitung aus einem Ersatz-Nichtterminal (siehe Kapitel 2.4).
Die gleiche Feststellung machten wir für die Wörterpaare Was und $Wasser$ sowie $Lehrer$ und $Lehrerin$.
Wir haben uns dazu entschieden, solche Feststellungen auch als positiven Schritt zu werten und die Hypothese als zielführend einzuordnen.

Gleichermaßen konnte dieser Umstand allerdings auch zu Fehldeutungen führen.

Beobachtung 2 Diesmal wurde für das initiale Fehlerwort W

$W = fahren\_der\_Fluss\_!$

ausgewählt. Das Orakel sollte jetzt bei dem Muster $*_1 er *_2 Fluss *_3$ anschlagen. Dabei durften $*_1$, $*_2$ und $*_3$ wieder beliebige Zeichenkombinationen sein. Diesen wurden bei W die folgenden Sequenzen zugewiesen:

$*_1 = fahren\_d$
$*_2 = \_$
$*_3 = \_!$

Aufgrund der in Abbildung 14 abgebildeten Regeln und dem Umstand, dass es keine weiteren Regeln gibt, bei denen das Wort $Fluss$ auf der rechten Seite vorkommt steht vor dem Wort $Fluss$ stets der Artikel *der*, von dem wiederum das Pattern *er* ein Teil ist. So wird jedes Wort, das $Fluss$ enthält crashen. Alle Wörter, die nicht $Fluss$ enthalten jedoch nicht.

```
1 Subjekt -> der' derSubst
2 der' -> der
3 derSubst -> Fluss
```

Abbildung 14: Regelauszug aus der „Sprache“-Grammatik

Als Fehlerhypothese erhielten wir die Regel $derSubst \rightarrow Fluss$. Der Grund,

warum wir keine weiteren Bäume erhielten liegt im Reduktionsschritt (siehe Kapitel 4.5).
Da die beiden Teilbäume aus Abbildung 15 die gleiche Fehlerquote haben und der rechte Teil des linken ist wird der linke Baum bei der Reduktion entfernt.

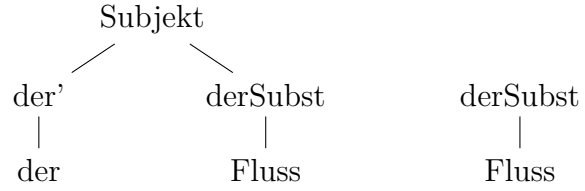

Abbildung 15: Zwei Bäume mit der gleichen Fehlerquote

Da dies in diesem Fall ein Schritt weg vom Ziel ist, haben wir uns dazu entschieden, diesen Fall als nicht ausreichend zu betrachten und negativ zu werten.

Beobachtung 3 Wir befinden uns in der „Taschenrechner"-Grammatik und betrachten das Fehlerpattern $*_1$) $*_2$ + $*_3$, wobei an den Stellen von $*_1$, $*_2$ sowie $*_3$ erneut beliebige Zeichenkombinationen stehen dürfen. Hierbei wurde fälschlicherweise die Hypothese aufgestellt, dass der Fehler in $*_1$ ($*_2$ + $*_3$ liegt. Das führen wir darauf zurück, dass, immer, wenn (vorkommt, irgendwann später) vorkommen muss. In der Originalgrammatik gibt es dafür die Regeln

```
1 Start -> Term Operator Term
2 Term -> ( Term Operator Term )
3 Term -> ( - Term )
```

Abbildung 16: Auszug aus der Taschenrechnergrammatik

Durch die Verwendung der Chomsky-Normalform werden diese Regeln aber noch aufgesplittet (siehe Kapitel 2.4).

```
1 Start -> Term Start^
2 Start^ -> Operator Term
3 Term -> (' Term^
4 Term -> (' Term^^
5 Term^ -> -' Term^^^
6 Term^^ -> Term Term^^^^
7 Term^^^ -> Term )'
8 Term^^^^ -> Operator Term^^^^^
9 Term^^^^^ -> Term )'
```

Abbildung 17: Auszug aus der CNF-Form der Taschenrechnergrammatik

Wir erwarteten eine Hypothese, wie in Abbildung 18 links abgebildet, erhielten allerdings die beiden rechts abgebildeten Bäume mit 100 % Fehlerquote als Hypothese. Zwar wird *Termˆˆ* immer auch zu) abgeleitet, aber das Zusammenspiel der beiden Hypothesen lässt aus unserer Perspektive nicht deutlich genug klar werden, das eigentlich) der gesuchte Ursachenteil ist.
Den Grund hierfür vermuten wir im Reduktionsschritt, den wir während der Hypothesenbildung durchführen (siehe Kapitel 4.5). Hier muss in der Zukunft untersucht werden, ob dieser Schritt verändert werden sollte, um bessere Ergebnisse zu liefern, oder entfernt werden muss.

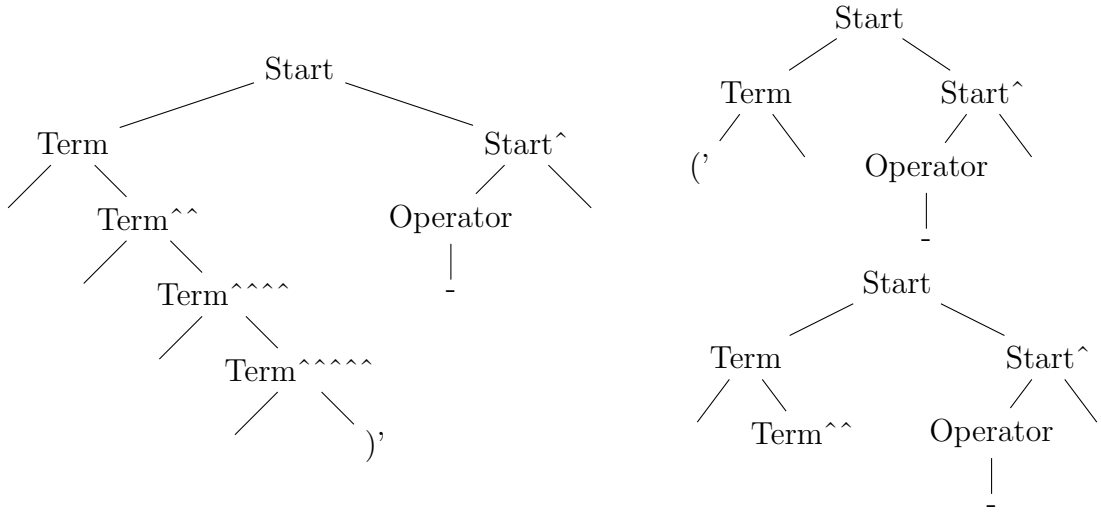

Abbildung 18: erwartete (li.) und erhaltene (re.) Fehlerhypothese

Wir stellen bei der Auswertung des Patternmodus also fest, dass kombinierte Fehler zu unpräzisen Hypothesen führen können. Für Patterns, die nur ein Terminal beinhalteten, das zu einem Crash führt, konnte von unserer Software präzise die Fehlerquelle ausfindig gemacht werden. Die Fehlerquote der erwarteten und auch gefundenen Hypothese hatte außerdem deutlichen Abstand zu den verbleibenden Fehlerquoten.

7 Zusammenfassung & Ausblick

Wir konnten unsere Ziele teilweise erreichen. Es ist uns gelungen Software zu entwickeln, die auf Grundlage einer kontextfreien Grammatik und eines gegebenen Crashes in einer Software Hypothesen zur Ursache dieses Crashes aufstellen kann.
Jedoch haben wir diese nur mit Dummygrammtiken evaluiert, da wir keinen Crash in LibreOffice finden konnten. Nach der von uns entdeckten Anomalie, beschrieben in Kapitel 4.1, halten wir eine weitreichende Überprüfung der Software für sinnvoll, bevor mit ihr weiter nach Fehlern gesucht wird. Als weitere Option bietet sich auch an, von LibreOffice wegzugehen und andere reale Testmöglichkeiten für unsere Software in Betracht zu ziehen. Dabei bietet sich dann auch ein Vergleich mit bestehenden Tools zur Ermittlung von Fehlerquellen an (bspw. Alhazen [KHSZ20]).

Wie wir bereits in Kapitel 5.2 beschrieben haben und in Tabelle 1 zu erkennen ist, ist die Anzahl der durchgeführten Test im Patternmodus unseres Orakels stark begrenzt. Hier könnten weitere Test und eine Automatisierung der Testauswertung sinnvoll sein. Außerdem ist noch zu untersuchen, ob unsere Software auch bei komplexeren Fehlermustern anwendbar ist, da wie in Kapitel 6.2 beschrieben bereits bei Mustern aus zwei Wörtern die Hypothesen ungenauer waren.

Da wir feststellen mussten, dass wir nicht ausreichend Rechenleistung für das schnelle, serielle Abarbeiten von großen Teilbaummengen zur Verfügung hatten, sodass wir eine Grenze bei Teilbaummengen der Größe 10000 ziehen mussten, sind die Testfälle, insbesondere für die zu Erwartenden größeren Inputs in realer Software, nicht repräsentativ. Außerdem ist zu erwarten, dass die Ausführung unserer Software dann sehr lange dauert. Da wir bei der Reproduktion keinerlei Parallelisierung eingebaut haben, bietet sich hier noch das Potenzial für eine Beschleunigung.

Eine weitere Möglichkeit stellt der Einbau einer gewissen Unschärfe dar. Unsere Software konzentriert sich bei der Teilbaumerstellung lediglich auf das ursprünglich ermittelte Fehlerwort. Das führt dazu, dass unsere Software beispielsweise bei einer Einstellung der „Sprache“-Grammatik, bei der jedes Wort, das *ich* enthält crasht, je nach initialem Fehler nur die Regel $Subjekt \rightarrow ich$ oder nur die Regel $Objekt \rightarrow ich$ als Hypothese liefern könnte. Es bietet sich hier zum Beispiel an, bei den abgeleiteten Terminalen auch rückwärts zu schauen, aus welchen Nichtterminalen diese noch abgeleitet werden können.

In diesen vier Lücken, die wir in dieser Arbeit nicht untersucht haben, ist in unseren Augen Potenzial für weitere Forschung.

Literatur

[BC67] W. H. Burkhardt and N. J. Camden. Generating test programs from syntax. *Computing*, 2:53–73, 1967.

[Buc18] Daniel Bucher. Grammar-based fuzzing for libreoffice. Bachelorarbeit, Humboldt Universität zu Berlin, 2018.

[HZ19] Nikolas Havrikov and Andreas Zeller. Systematically covering input structure. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 189–199. IEEE Press, 2019.

[Kö17] Prof. Dr. Johannes Köbler. Einführung in die theoretische informatik (wintersemester 2017/18), 2017.

[KHSZ20] Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, and Andreas Zeller. When does my program do this? learning circumstances of software behavior. ESEC/FSE 2020, page 1228–1239, New York, NY, USA, 2020. Association for Computing Machinery.

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990.

[MZH21] Barton P. Miller, Mengxiao Zhang, and Elisa R. Heymann. The relevance of classic fuzz testing: Have we solved this one?, 2021.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 28. August 2022