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

Grammar-Based Fuzzing for Libre Office

Bachelorarbeit

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

eingereicht von: Daniel Bucher
geboren am: 18.10.1996
geboren in: Bonn

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

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

Inhaltsverzeichnis

Abstract Das Ziel dieser Arbeit ist die Entwicklung einer Software, mit welcher Grammatik basierte Eingaben für LibreOffice, mittels eines Fuzzers, generiert werden können. Dabei können probabilistische Informationen verwendet werden, um die Güte der generierten Eingaben zu verbessern. In dieser Arbeit wird die entwickelte Software beschrieben und ihre Funktionsweise erläutert. Außerdem wird ein Überblick über das OpenDocument Format gegeben.

1. Einleitung

Das Testen ist ein großer Bestandteil der Softwareentwicklung. Dabei wird versucht, eine möglichst umfassende Abdeckung des Quellprogramms zu erreichen [RL10]. Das Ziel ist es, viele Fehler zu finden, um eine robuste und sichere Software mit einer hohen Qualität entwickeln zu können [WPC06].
Um ein Programm zu testen, werden Eingaben benötigt, welche die verschiedenen Funktionen des zu testenden Programms ausführen. Die Komplexität von Tests steigt jedoch mit dem Umfang der Eingaben, da diese meist nicht trivial sind. In dieser Arbeit wollen wir geeignete Testeingaben für die, von der *Document Foundation* vertriebenen, Büroanwendung *LibreOffice*[1] generieren. Die Schwierigkeit in dieser Aufgabe besteht darin, dass Eingaben für *LibreOffice* aus mehreren Dateien zusammengesetzt werden, welche alle einer vorgegebenen Syntax folgen.

Fuzzing, als Methode der Testfallgenerierung eignet sich gut, um ein Fehlverhalten des Programms aufzudecken [GKL08]. Dabei können unter anderen Fehler abgedeckt werden, welche entstehen, wenn Eingabemengen für Parameter überschritten werden [GKL08]. Solche Fehler können sich auch als sicherheitskritisch erweisen, wenn dadurch unbeabsichtigte Aktionen ausgeführt werden [BFF08].

[1] https://de.libreoffice.org/

2. Hintergrund & verwandte Arbeiten

In diesem Abschnitt sollen technische Hintergründe zu Fuzzing, probabilistischen Informationen und dem in der Arbeit verwendeten *OpenDocument Standard* gegeben werden. Außerdem werden verwandte Arbeiten behandelt, welche bereits auf diesem Themengebiet Forschungsergebnisse erarbeitet haben.

2.1. Grammatik

Bei Grammatiken handelt es sich um einen essentiellen Bestandteil der theoretischen Informatik. Diese werden unter anderem in der Berechenbarkeitstheorie und im Compilerbau eingesetzt [Wir13]. Grammatiken sind eine bliebte Methode um formale Sprachen zu beschreiben. Dabei wird eine Grammatik durch ein 4-Tupel dargestellt [Cho56].

$$G \;=\; (V, \Sigma, P, S)$$

V ist dabei die Menge der *Variablen*. Eine *Variable* kann nicht in einem endgültigen Wort stehen. Sie muss durch Produktionsregeln zu einem *Terminalsymbol* abgeleitet werden.
Diese Menge wird auch als *Nichtterminalsymbole* bezeichnet.

Σ ist das Alphabet der *Terminalsymbole*. Konkatenationen von Zeichen dieser Menge bilden die Menge aller zulässigen Wörter. In einem final abgeleiteten Wort dürfen ausschließlich
Terminalsymbole enthalten sein.

P steht für die endliche Menge von Produktionsregeln. Eine Produktionsregel ist dabei eine Vorschrift, nach welcher eine Symbolfolge zu einer anderen Symbolfolge angeleitet werden kann. Als Beispiel kann eine Folge von *Nichtterminalsymbolen* zu einem Wort abgeleitet werden.

S ist das *Startsymbol*. Wird ein Wort nach den Regeln der Grammatik abgeleitet, so wird bei der Ableitung mit dem Startsymbol begonnen. Das Startsymbol ist immer ein Element der *Nichtterminalsymbole*.

Eine Grammatik kann dabei anhand der Chomsky-Hierarchie klassifiziert werden.[Cho56] So gilt für eine Grammatik $G \;=\; (V, \Sigma, P, S)$:

1. G heißt Typ 3 oder regulär, wenn für alle Regeln P gilt:
   $P \subseteq V \times (\Sigma V \cup V \cup \epsilon)$
2. G heißt Typ 2 oder kontextfrei, falls für alle Regeln P gilt:
   $P \subseteq V \times (\Sigma \cup V)^*$

3. G heißt Typ 1 oder kontextsensitiv, falls für alle Regeln P gilt:
   $P \subseteq ((\Sigma \cup V)^* - \Sigma^*) \times (\Sigma \cup V)^*$ und die rechte Seite jeder Ableitungsregel in P zu gleich vielen oder weniger Zeichen ableitet.

4. Jede Grammatik G ist immer auch vom Typ 0 oder rekursiv aufzählbar.

In dieser Arbeit werden hauptsächlich kontextfreie Grammatiken verwendet.

2.2. Kontrollflussorientierung

Kontrollflussorientierte Testverfahren gehören zu den strukturierten Testverfahren [BE96]. Strukturierte Testverfahren bestimmen ihre Testfälle anhand des bekannten Quelltextes. Im Gegensatz dazu haben unstrukturierte Testverfahren kein Wissen über den Quelltext. Im kontrollflussorientierten Testen wird das Programm in Form eines Kontrollflussgraphen notiert. Unter einem Kontrollflussgraphen versteht man einen gerichteten Graphen, in welchem der Verlauf eines Programms dargestellt wird. Im Folgenden ist der Kontrollflussgraph (Abb. 2) für eine Minimumfunktion (Abb. 1) dargestellt. Dabei entsprechen die Nummern in den Zuständen des Kontrollflussgraphen den ihnen zugeordneten Zeilennummern im Programmcode. Der Startzustand ist mit *Start* und der Endzustand ist mit *End* gekennzeichnet.

```
0 function integer min(int a; int b) {
1   if(a < b) {
2     return a;
3   }
4   return b;
5 }
```

Abbildung 1: Diese Methode gibt das Minimum zweier Zahlen zurück.

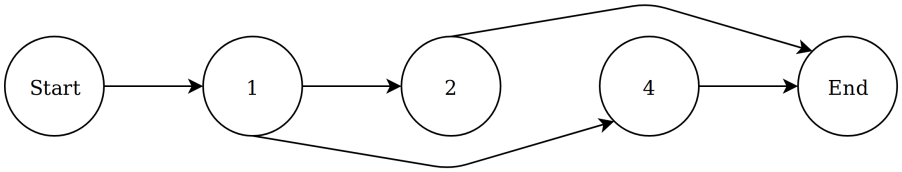

Abbildung 2: Der Kontrollflussgraph zur Minimumfunktion aus Abbildung 1

Balzert et al. unterteilen das kontrollflussorientierte Testen noch weiter [BE96]: Dabei werden jeweils verschiedene Strategien angewandt, um eine möglichst umfassende Abdeckung des Quellcodes zu erreichen.

Anweisungsüberdeckungstest (C_0): Das Ziel dieses Verfahrens ist es jede Anweisung des zu testenden Programms mindestens einmal auszuführen. Wird eine vollständige Abdeckung des Quellcodes erreicht, so ist sichergestellt, dass kein toter Code[2] im Programm enthalten ist. Als Metrik wird dabei das Verhältnis zwischen den *überdeckten Anweisungen* und der *Gesamtzahl der Anweisungen* angegeben. Ein entscheidender Nachteil dieses Vorgehens ist jedoch der Umgang mit Schleifen, da das Programmverhalten für mehrere Schleifendurchgänge nicht berücksichtigt wird.

Zweigüberdeckungstest (C_1) Bei der Zweigüberdeckung wird versucht die Menge der Kanten im Kontrollflussgraphen abzudecken. Dabei werden meist mehr Testfälle benötigt, als bei der Anweisungsüberdeckung, da jede Entscheidung im Kontrollflussgraphen mindestens einmal zu `true` und einmal zu `false` evaluiert werden muss. Als Metrik wird hier das Verhältnis zwischen *Anzahl der abgedeckten Zweige* und *Gesamtzahl der Pfade* verwendet. Genau wie beim Anweisungsüberdeckungstest wird auch von diesem Verfahren nur eine geringe Abdeckung von Schleifen erreicht.

Pfadüberdeckungstest (C_2) Die Pfadüberdeckung stellt eine Erweiterung der Zweigüberdeckung dar. Bei diesem Vorgehen werden die möglichen Pfade vom Start- zum Zielknoten betrachtet. Für den Umgang mit Schleifen, werden verschiedene Subklassen des Pfadüberdeckungstests verwendet. Dabei werden in den verschiedenen Varianten die Pfade einer Schleife unterschiedlich häufig durchlaufen.

Bedingungsüberdeckungstest (C_3) Beim Bedingungsüberdeckungstest werden die Verzweigungen des Kontrollflussgraphen überdeckt. Dabei unterscheidet man zwischen folgenden Varianten:

C_{3a} Beim *Einfachbedingungsüberdeckungstest* muss jede Bedingung mindestens einmal zu `true` und einmal zu `false` evaluiert werden.

C_{3b} Beim *Mehrfachbedingungsüberdeckungstest* muss jede Kombinationen aller atomarer Bedingungen jeweils zu `true` und zu `false` evaluiert werden. Dadurch ergeben sich bei n atomaren Aussagen in einer `if` Verzweigung n^2 mögliche Kombinationen, da jede atomare Aussage genau zwei Zustände annehmen kann.

C_{3c} Der minimale *Mehrfachbedingungsüberdeckungstest* liegt vom Umfang zwischen C_{3a} und C_{3b}. Dabei wird die logische Struktur der Bedingung mit einbezogen.

[2]Quellcode, welcher bei der Programmausführung nie erreicht werden kann.

2.3. *Fuzzing*

Beim Fuzzing handelt es sich um eine Technik aus dem Bereich der Softwaretests [GKL08]. Fuzzing wird verwendet um große Mengen an *Testdaten* für Programme zu generieren und zu testen. Dabei werden in der ursprünglichen Variante des Fuzzings ausschließlich zufällige Daten generiert. Diese werden im Weiteren gegen die Eingabeschnittstelle des jeweiligen Programms getestet. Für die Minimumfunktion aus Abbildung 1 würden somit Paare von zufälligen Integer Werte generiert werden. Vorteile der Verwendung von Fuzzing sind, dass durch das Verwenden von Zufallszahlen die Eingaben auch ungewöhnliche Formen annehmen können, welche bei gewöhnlichen Eingaben gar nicht oder nur selten auftreten [FM00]. Ein weiterer Vorteil ist die Zeitersparnis gegenüber der händischen Erstellung von *Testdaten*. Darüber hinaus kann mittels der durch Fuzzing zufällig erzeugten Daten, im Allgemeinen eine hohe Pfadabdeckung erreicht werden [DN81].

Ein Nachteil des einfachen Fuzzings ist jedoch die Komplexität von benötigten Eingabedaten. Eine Methode zur Berechnung des Minimums aus zwei Zahlen benötigt nur zwei Zufallszahlen, während bereits bei einem Taschenrechner die Komplexität der Eingaben ansteigt. Für diese nicht trivialen Eingaben kann Fuzzing erweitert werden. Dabei kann zwischen *Black und White-Box-Fuzzing* unterschieden werden.[ND12] Diese beiden Gruppen werden im Weiteren genauer vorgestellt.

Blackbox-Fuzzing:

Hierbei wird, ähnlich wie beim *unstrukturierten Testen* ohne Wissen über den Programmcode und wenig bis keinem Wissen über die zugrunde liegende Programmstruktur gearbeitet [ND12]. Zur Erstellung der zufälligen Eingaben existieren mehrere Möglichkeiten.

Mutation-based Blackbox-Fuzzing benötigt als Grundlage für die Erstellung der zufälligen Testdaten eine Grundmenge von korrekten (validierten) Eingaben. Diese Grundmenge wird auch als *Seedmenge*, *Seed-Daten* oder einfach *Seeds* bezeichnet. Die *Seeds* sollten dabei den möglichen Eingabebereich idealerweise umfassend abdecken. Auf Kopien dieser *Seeds* werden dann verschiedene Mutationsoperatoren angewandt. Zu den möglichen Mutationsoperatoren zählen unter anderem *Bitstring Mutation* und mehrere Arten der Vertauschung. Daraus können dann beliebig viele Eingabedaten generiert werden [RCA+14]. Wir wollen zur Verdeutlichung dieses Vorgehens einen einfachen Taschenrechner betrachten, welcher die Operationen *Multiplikation*, *Division*, *Addition* und *Subtraktion* anwenden kann. Als *Seed-Daten* wählen wir:

- 1 + 2
- 6 / 2

Durch Anwendung von Mutationsoperatoren auf diese *Seeds* erhalten wir die folgenden mutierte Testdaten:

- $*$ + 2 (Vertauschung einer Ziffer durch einen Operator),
- 6 / 0 (Vertauschung einer Ziffer durch eine andere Ziffer).

Diese können nun gegen die Eingabeschnittstelle des Taschenrechners getestet werden. Wie in diesen Testdaten zu erkennen ist, führt diese Art des Fuzzings vermehrt dazu, dass generierte Eingaben aufgrund einer fehlerhaften Syntax von der Eingabeschnittstelle des Zielprogramms abgelehnt werden.

Ein weiterer Nachteil dieses Vorgehens ist darüber hinaus, dass Teile des Eingabebereichs nicht durch den Fuzzer erreicht werden, wenn diese nicht durch die *Seed-Daten* abgedeckt werden. So kann es bei diesem Ansatz passieren, dass selbst durch häufige Anwendung von Mutationsoperatoren der Testdaten, große Teile des Eingabebereichs nicht berücksichtigt werden.

Beim *Modelbased Blackbox-Fuzzing* werden hingegen keine *Seeds* benötigt. Stattdessen werden die Eingabedaten vollständig aus einem vorher definierten Modell generiert [BPR17]. Wie auch in dieser Arbeit kann als Modell eine beliebige Grammatik genutzt werden. Der Fuzzingprozess orientiert sich bei der Generierung der Eingabedaten, je nach Art des Modells und der Implementation des Fuzzers, an dem zugrunde liegenden Modell [HHZ12]. Im Falle einer Grammatik werden, beginnend mit dem Startsymbol, die *Nichtterminalsymbole* Regel für Regel zu feststehenden *Terminalsymbolen* abgeleitet. Dies bietet den Vorteil gegenüber dem *Mutationbased Blackbox-Fuzzing*, dass ohne geschickte Auswahl einer Grunddatenmenge alle Teile des Eingabebereiches erreicht und abgedeckt werden können [HHZ12]. Auch dieses Vorgehen wollen wir im Folgenden mit einem beispielhaften Taschenrechner verdeutlichen. Dazu wählen wir als beschreibendes Modell eine Grammatik mit den unten stehenden Ableitungsregeln:

$$\begin{aligned} \mathit{Start} &\rightarrow \mathit{Digit}\ \mathit{Operator}\ \mathit{Digit} \\ \mathit{Operator} &\rightarrow \text{"}-\text{"}|\text{"}+\text{"}|\text{"}/\text{"}|\text{"}*\text{"} \\ \mathit{Digit} &\rightarrow \text{"0"}|\text{"1"}|\text{"2"}|\text{"3"}|\text{"4"}|\text{"5"}|\text{"6"}|\text{"7"}|\text{"8"}|\text{"9"} \end{aligned}$$

Wir nehmen an, dass bei einer Entscheidung alle Wahrscheinlichkeiten für die Wahl einer Möglichkeit gleichverteilt sind. Nun lassen sich aus diesem Modell beispielsweise folgende Testeingabe ableiten.[3]

$$\underline{\mathit{Start}} \rightarrow \underline{\mathit{Digit}}\ \mathit{Operator}\ \mathit{Digit} \rightarrow 4\ \underline{\mathit{Operator}}\ \mathit{Digit} \rightarrow 4 + \underline{\mathit{Digit}} \rightarrow 4 + 3$$

Diese können nun gegen die Eingabeschnittstelle des Taschenrechners getestet werden.

Jedoch bietet dieses Vorgehen auch allgemeine Nachteile. Eins der großen Probleme sind Sonderfälle. Wählen wir für ein Beispiel einen beliebigen Programmpfad, welcher nur im Fall `(x == 10)` ausgeführt wird, so wird dieser Fall nur mit einer geringen Wahrscheinlichkeit von 1 zu 2^{32} ausgeführt[4] [GLM+08].

[3] Der nächste Ableitungsschritt ist dabei immer unterstrichen.

[4] Dabei wird ein 32 Bit Integer angenommen.

Whitebox-Fuzzing:
Hierbei kann im Vergleich zum *Blackbox-Fuzzing* auf den Quellcode des Programms und die Programmstruktur zurückgegriffen werden. Dadurch ergeben sich mehr Optionen um eine möglichst umfassende Abdeckung des Quellcodes zu erreichen. Mittels einer Analyse des Quellcodes lassen sich dabei alle endlichen und ausführbaren Programmpfade bestimmen. Nach einer Zuordnung der Variablen des Programms zu symbolischen Variablen, kann der Quellcode mit den symbolischen Variablen interpretiert werden. Aus den symbolsichen Variablen ergeben sich Pfadbedingungen für jeden der einbezogenen Pfade. Diese Pfadbedingungen, welche auch als *Constraints* bezeichnet werden, können im Anschluss mit einem *Satisfiability Modulo Theories Solver (SMT-Solver)* aufgelöst werden [Kin76]. Nach dem Auflösen der *Constraints*, kann für jeden Pfad festgestellt werden, ob dieser erreichbar ist. Darüber hinaus ergibt sich für jeden erreichbaren Programmpfad ein Wertebereich der Variablen, welcher die Ausführung dieses Programmpfads ermöglicht. Beim *Whitebox-Fuzzing* können nun, ausgehend von einer Eingabe, die Constraints eines Programmpfads bestimmt werden. Durch Manipulation der Constraints können somit weitere Programmpfade erreicht werden. Dies hat die Maximierung der Codeabdeckung zum Ziel. Im folgenden Beispiel, dargestellt in Abbildung 3, gehen wir davon aus, dass die erste bekannte Eingabe der Belegung `a = true` entspricht. Wird nun diese Pfadbedingung manipuliert (negiert), ergibt sich die Belegung `a = false`. Damit konnte ein weiterer Pfad abgedeckt werden. Die Pfadbedingungen der beiden Pfade sind somit vollständig bestimmt, da jede mögliche Belegung für `a` in mindestens einer Pfadbedingung enthalten ist.
Ein Problem bei einer solchen symbolischen Ausführung des Programms ist die Menge der zu testenden Pfade. In großen Programmen kann die Pfadmenge schnell in exponentiellem Maße anwachsen. Dies kann durch eine große Anzahl an Verzweigungen und Schleifen entstehen. In solchen Fällen kann aufgrund einer hohen Generierungszeit die Verwendung eines Blackbox-Fuzzers effizienter sein [BP15]. Zur Verbesserung der Leistung eines Whitebox-Fuzzers, kann dafür die Menge der zu testenden Pfade künstlich beschränkt werden. Darüber hinaus kann der Prozess durch Verwendung eines Eingabemodells, wie einer Grammatik, verbessert werden.[GKL08]

```
0  function foo(boolean: a) {
1    if(a) {
2      print("Pfad 1.")
3     }else{
4      print("Pfad 2.")
5     }
6  }
```

Abbildung 3: Beispielhafter Code einer Whitebox-Pfadanalyse

2.4. Fuzzing mit probabilistischen Informationen

Beim Fuzzing mit probabilistischen Informationen handelt es sich um eine Form des *Model-based Blackbox Fuzzings*. Dabei kann als beschreibendes Modell eine Grammatik genutzt werden. Zusätzlich wird eine möglichst große Menge an validen Eingabedaten benötigt. Diese Eingabedaten müssen im Anschluss auf ihre Struktur untersucht werden. Mittels der strukturellen Informationen jeder Datei aus den Eingabedaten können so Auftrittswahrscheinlichkeiten festgestellt werden. Mit diesen Wahrscheinlichkeiten kann das Modell erweitert werden [JLM92]. Dies wird an folgendem Beispiel dargestellt. Dazu sei eine beispielhafte Grammatik $G = (V, \Sigma, P, S)$ mit $V = \{A,\ B\}, \Sigma = \{w,\ x,\ y,\ z\}$ und

$$\begin{aligned} P = \quad & S \ \rightarrow \ A \mid B \mid w \\ & A \ \rightarrow \ xB \mid y \\ & B \ \rightarrow \ yA \mid z \end{aligned}$$

gegeben.
Darüber hinaus nehmen wir zwei valide Testeingaben der Form $xyxyxyxz$ und $yxyxyy$ an. Nach dem Ableiten der Eingaben, lassen sich durch die daraus resultierenden Ableitungsbäume die relativen Häufigkeiten für jede Ableitungsregel bestimmen. Ist die Ableitung einer Eingabe nicht eindeutig, so wird dies je nach Implementation der verwendeten Software behandelt. Diese werden im Anschluss an den Grammatikregeln annotiert. Es ergeben sich folgende Annotationen.

$$\begin{aligned} S \ &\rightarrow \ A \mid B \mid w \quad && \left(\tfrac{1}{2}; \tfrac{1}{2}; 0\right) \\ A \ &\rightarrow \ xB \mid y && \left(\tfrac{6}{7}; \tfrac{1}{7}\right) \\ B \ &\rightarrow \ yA \mid z && \left(\tfrac{5}{6}; \tfrac{1}{6}\right) \end{aligned}$$

Wie an diesem Beispiel deutlich wird, wird die Eingabemenge der Grammatik nicht umfassend abgedeckt, da die Ableitung zum Wort w zu 0% vorgenommen wird. Ein Grund dafür kann sein, dass diese Ableitungen in den verwendeten Eingaben nie oder nur sehr selten zum Einsatz kommen. Wird dieses Verhalten in die probabilistischen Informationen übernommen, so werden auch die daraus generierten Programmeingaben dieses Muster aufweisen. Diese Eingaben gelten als *gewöhnliche* Eingaben [PSH+18].
Wie Pavese et al. vorstellen, kann die probabilistische Grammatik *invertiert* werden, um daraus *seltene* Eingaben zu generieren [PSH+18]. Beim *Invertieren* der Grammatik wird für jede anotierte Wahrscheinlichkeit ihre Gegenwahrscheinlichkeit gebildet. Die aus der invertierten Grammatik erstellten Eingaben werden als *inputs from hell* bezeichnet.
Wenden wir dieses Verfahren auf die verwendete Beispielgrammatik an, so erhalten wir folgende probabilistische Werte:

$$\begin{aligned} S \ &\rightarrow \ A \mid B \mid w \quad && (0; 0; 1) \\ A \ &\rightarrow \ xB \mid y && \left(\tfrac{1}{7}; \tfrac{6}{7}\right) \\ B \ &\rightarrow \ yA \mid z && \left(\tfrac{1}{6}; \tfrac{5}{6}\right) \end{aligned}$$

Mit diesen *inputs from hell* bezeichneten, Grammatik, kann durch die probabilistischen Informationen die Eingaben w abgeleitet werden.
Mittels dieses Verfahrens lassen sich durch die probabilistische Grammatik gewöhnliche Eingaben erstellen. Durch die zweite *invertierte* Grammatik können ungewöhnliche und seltene Eingaben erstellt werden.
Alternativ dazu bietet sich auch die Möglichkeit, durch geschickte Auswahl weiterer Testeingaben, die Wahrscheinlichkeiten umfassender zu verteilen. Damit kann die Eingabemenge bereits durch eine Grammatik abgedeckt werden. Bei diesem Vorgehen würden wir eine weitere Testeingabe w hinzufügen. Würde nun die Ableitung und Annotation der Wahrscheinlichkeiten, mit den erweiterten Testeingaben, auf die ursprünglichen Grammatik angewandt, so ergäbe sich diese Änderung der ersten Regel zu:

$$S \rightarrow A \mid B \mid w \; - \; \left(\frac{1}{3}; \frac{1}{3}; \frac{1}{3}\right)$$

Nach dieser Änderung deckt diese probabilistische Grammatik den vollständigen Eingabebereich ab. Als Einbuße sind die von dieser Grammatik generierten Eingaben nicht mehr vollständig realitätsnah.

2.5. Testfallauswertung

Werden zum Testen von Programmen zufällig erzeugte Eingaben verwendet, so kann der Test nicht ohne weiteres evaluiert werden, da das korrekte Programmverhalten für zufällige Eingaben nicht bekannt ist. Somit muss ein anderes Vorgehen verwendet werden um den Ausgang der Tests bestimmen zu können. Diese Entscheidungsstrategien werden als *Orakel* bezeichnet. Für diese Problemstellung werden im Folgenden zwei mögliche Vorgehensweisen vorgestellt.
Bei **Crashtesting** wird das Programm mit den entsprechenden Eingabedaten gestartet. Gibt es einen Programmabsturz für einen Test t_i, so wird dieses Ergebnis als $CT_{t_i} = 0$ bezeichnet und gilt als nicht erfolgreicher Abschluss. Startet das Programm ohne abzustürzen, unabhängig von der Programmausgabe, so gilt der Test t_i als erfolgreich. Dieser Fall wird als $CT_{t_i} = 1$ bezeichnet.
Dieser Auswertungsmodus kann nur eingeschränkte Aussagen über die Korrektheit des Programms machen, da auch die Menge an Fehlern, welche erkannt werden können, begrenzt ist. Durch diese Variante erkennbare Fehler lassen sich in Speicherfehler, Programm Assertions und nicht weiter spezifizierte Programmabstürze einteilen [Ney08]. Tritt der Fall $CT_{t_i} = 1$ ein, so kann das Programm trotzdem logische Fehler enthalten, welche in diesem Modus nicht erkannt werden können. Am Beispiel eines Taschenrechners kann durch dieses Verfahren nicht geprüft werden, ob das Ergebnis einer eingebenen Rechenaufgabe mathematisch korrekt ist.
Eine zweite Vorgehensweise ist die im Folgenden als **Observation Testing** bezeichnete Technik. Dabei wird das Programm mit den Eingabedaten gestartet. Zur Auswertung werden eine oder mehrere Kontrollinstanzen verwendet, die mit den selben Eingabedaten gestartet werden. Zur Auswertung des Tests werden alle vorliegenden Ergebnisse

miteinander verglichen. Weichen dabei die Ergebnisse des zu testenden Programms von denen der Kontrollinstanzen ab, so kann von einem Fehler im Programm ausgegangen werden. Ein Nachteil dieser Vorgehensweise ist, dass selbst bei Übereinstimmung aller Ergebnisse nicht darauf geschlossen werden kann, dass es keinen Fehler gibt. Es besteht die Möglichkeit, dass auch die Kontrollinstanzen fehlerhaft sind. Wenn wir wieder das Beispiel des Taschenrechners heranziehen und als gewählte Kontrollinstanz einen zweiten Taschenrechner verwenden, so können beide Ergebnisse von einander abweichen. Dieses Orakel macht an keiner Stelle eine Aussage, welche Instanz fehlerhaft ist. Nur bei einem Programmabsturz kann eine fehlerhafte Instanz identifiziert werden. Es könnte auch die Kontrollinstanz fehlerhafte Ergebnisse liefern.
Ein weiterer Nachteil dieses Vorgehens ist, dass für jedes zu testende Programm eine geeignete Kontrollinstanz zur Verfügung stehen muss. Dies ist bei besonders umfangreichen oder inhaltlich einzigartigen Programmen nicht trivial und teilweise nicht möglich.

2.6. OpenDocument Standard

Beim OpenDocument Format (ODF) handelt es sich um einen von Sun Microsystems und OASIS entwickelten internationalen Standard für Bürodokumente [Wei09]. Mit diesem Dateiformat können Texte, Tabellen, Präsentationen, Zeichnungen und weitere Dateien gespeichert werden. Das ODF basiert auf der *Extensible Markup Language* (kurz: XML). Die erste Version des ODFs wurde 2005 von OASIS veröffentlicht. Im Jahr 2011 wurde die Version 1.2 vorgestellt, welche bis zum Zeitpunkt des Schreibens der Arbeit die aktuelle Version des Formats darstellt.[5] Bei einem Dokument, welches dem ODF folgt, handelt es sich um ein Zip-Dateiarchiv[6]. Die beispielhafte Ordnerstruktur für eine OpenDocument Text (*.odt) Datei ist in Abbildung 4 dargestellt.

META-INF/manifest.xml	Diese Datei enthält eine Liste aller nicht standardmäßig enthaltenen Dateien des Dokuments und derer Internet Medientypen [FB96]. (kurz: *mimetypes*).
Thumbnails/thumbnail.png	Stellt das Vorschaubild des entsprechenden Dokuments dar. Es wird aus der ersten Seite des Dokuments berechnet.
content.xml	Diese Datei ist das zentrale Element des Dokuments. Darin wird der gesamte Inhalt abgelegt.
manifest.rdf	In dieser Datei werden alle standardmäßig in einem Dokument des ODF enthaltenten Dateien und deren *mimetypes* gespeichert.
meta.xml	Als Metadaten werden in dieser Datei Informationen über das Dokument gespeichert. Dazu zählen Erstellungszeit oder Name des Autors.

[5] https://docs.oasis-open.org/office/v1.2/OpenDocument-v1.2-part1.html
[6] https://www.oasis-open.org/committees/download.php/12572/OpenDocument-v1.2-os.pdf

mimetype	In dieser Datei wird der *mimetype* des Dokuments festgelegt. Dieser setzt sich aus der Programmbezeichnung und dem Dateiformat zusammen [ODF11].
settings.xml	Enthält die Einstellungen, welche die Büroanwendung beim Öffnen des Dokuments lädt.
styles.xml	Beschreibt die stilistischen Einstellungen des Dokuments, unter anderem die verschiedenen Schriftarten.

Die *.xml Dateien in einem Dokument folgen dabei einem vorgegebenen Schema, in Form einer kontextfreien Grammatik. Das beschreibende Schema dieser Grammatik wird in der aktuellen Version von OASIS als *.pdf und als *.rng zur Verfügung gestellt [ODF11].

```
test.odt
+-META-INF
| +-manifest.xml
+-Thumbnails
| +-thumbnail.png
+-content.xml
+-manifest.rdf
+-meta.xml
+-mimetype
+-settings.xml
+-styles.xml
```

Abbildung 4: Beispielhafter Aufbau eines OpenDocument Texts (*.odt)

2.7. Verwandte Forschung

Das ursprüngliche Konzept der Grammatik wurde bereits Ende der 50er Jahre von Chomsky entwickelt [Cho56]. Jedoch enstammt die Idee, Grammatiken als Modelle für Testeingaben von Programmen zu verwenden, den frühen 70er Jahren. Purdom führte dazu Experimente durch, in welchen er Testeingaben aus einer kontextfreien Grammatik generierte [Pur72]. Dieser Teilbereich des Fuzzings wird als *Grammar Based Fuzzing* bezeichnet. Damit können auch Sonderfälle der Eingabemenge abgedeckt werden, wie auch Godefroid et al. besprechen [GKL08]. Grammatiken mit der Verwendung von probabilistischen Informationen werden häufig in Teilgebieten der Informatik eingesetzt: Dazu zählen neben der Testfallgenerierung auch die Computerlinguisitik [MMS99] und die Biochemie, in der unter anderem DNA und RNA Sequenzen analysiert werden [SBH+94]. Ein weiterer Ansatz zur Testfallgenerierung ist die Kombination von probabilistischen Grammatiken und einem suchbasierten Testansatz, bei welchem genetische Algorithmen verwendet werden [KTT14]. Dieser von Kifetew et al. beschriebene Ansatz bietet eine große Pfadabdeckung in Abhängigkeit der gewählten Zielfunktion, nach welcher die genetischen Algorithmen ihre Ausgaben optimieren [KTT17].

Mit der Verwendung von Grammatiken kann beim Fuzzing sichergestellt werden, dass die Eingaben den vordefinierten Regeln folgen. Wie Le in seiner Arbeit zum Vergleich von zufälligem und probabilistischem *Grammar Based Fuzzing* zeigt, ergibt sich eine weitaus höhere Robustheit und Realitätsnähe bei der Verwendung von probabilistischen Informationen [Le17]. Darüber hinaus gehen Pavese et al. in ihrer Arbeit *Inputs from Hell: Generating Uncommon Inputs from Common Samples* auf eine Strategie ein, mit welcher durch die Nutzung von probabilistischen Informationen auch seltene und ungewöhnliche Eingabedaten erzeugt werden können [PSH+18].
Die in dieser Arbeit entwickelte Anwendung ermöglicht das Generieren von zufallsbasierten und probabilistischen Eingaben für *LibreOffice*. Sie kann leicht mit dem von Pavese et al. beschriebenen Vorgehen erweitert werden.

3. Vorgehensweise

Das Ziel war es, eine Software zu entwickeln, welche Fuzzing für LibreOffice als Schwerpunkt hat. In diesem Kapiel wollen wir den Aufbau der Software sowie das Vorgehen in den einzelnen Teilabschnitten genauer betrachten. Wie in Abbildung 5 dargestellt, besteht die entwickelte Software aus fünf Komponenten. Im ersten Abschnitt werden wir auf den

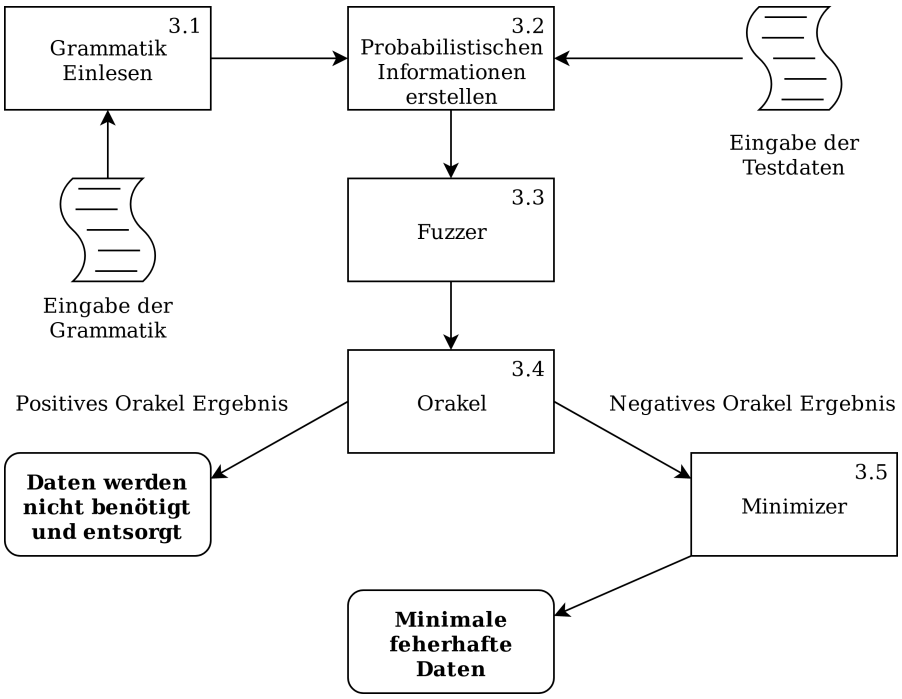

Abbildung 5: Aufbau der einzelnen Softwarekomponenten

Aufbau der Grammatik und das Vorgehen beim Einlesen dieser Grammatik eingehen. Der darauf folgende Abschnitt wird das Erstellen der probabilistischen Informationen zum Fokus haben. Der dritte Abschnitt wird vom Fuzzer handeln und das Vorgehen desselben genauer beleuchten. Als Entscheidungsoperator wird im darauf folgenden Abschnitt das *Orakel* behandelt und erklärt. Den letzten Abschnitt des Programms bildet der Minimizer. Im Anschluss, zur Erläuterung des Programmaufbaus, werden zum Ende des Kapitels die Ergebnisse des entwickelten Programms präsentiert.
Verfasst wurde die Software in Java unter der Version 11.0.3. Zur Ausführung wird darüber hinaus eine Installation von LibreOffice benötigt, welche für Tests und Entwicklung der Software in Version 6.0.7.3[7] installiert war. Während der Entwicklung wurde die Grammatikversion 1.2 des OpenDocument Formats[8] verwendet. Um ausschließlich Text Dokumente (*.odt) zu generieren, wurde die Grammatik geringfügig angepasst: Diese Anpassung ist in Abbildung 14 im Anhang der urpsrünglichen Fassung in Abbildung 15 im Anhang gegenübergestellt.

[7]Build-ID: 1:6.0.7-0ubuntu0.18.04.6
[8]https://docs.oasis-open.org/office/v1.2/OpenDocument-v1.2-part1.html

3.1. Grammatik

Die Grammatik, welche von der Software verwendet werden kann, muss im auf XML basierenden Relax NG Format[9] (kurz RNG)[Cla01] vorliegen. In diesem Format werden die Produktionsregeln, sowie die Startregel angegeben. Die, der Grammatik zugehörigen, *Nichtterminalsymbole*, werden in diesem Format nicht separat angegeben. Diese erschließen sich ausschließlich durch ihre Verwendung. Auch die *Terminalsymbole* werden im Relax NG Format nicht separat aufgeführt. Im Gegensatz zu den *Nichtterminalsymbolen* lassen sich die *Terminalsymbole* jedoch in drei Kategorien aufteilen.
In die erste Kategorie lassen sich die *Terminalsymbole* einteilen, welche wörtlich in dem Ableitungsregeln genannt werden. Im RNG Format werden diese durch einen `value`-Tag eingeleitet. Dazu zählen beispielsweise:

- `<value>true</value>`
- `<value>false</value>`

Diese werden im Folgenden als **Wertterminale** bezeichnet.

Die zweite Kategorie der *Terminalsymbole* lässt sich durch die Art der Ableitung gruppieren. Diese *Terminale* werden nicht direkt abgeleitet, sondern enthalten ausschließlich Informationen über die Generierung von Werten, zu denen abgeleitet wird. Beispiele für diese Kategorie sind:

- `<param name="pattern">-?[0-9]+,-?[0-9]+([ ]+-?[0-9]+,-?[0-9]+)*</param>`
  Dieses *Terminalsymbol* wird bei Ableitung zu einem, dem regulären Ausdruck entsprechenden, String abgeleitet.
- `<data type="string"><param name="length">1</param></data>`
  Dieses *Terminal* gibt an, dass hier zu einem einelementigen String abgeleitet werden soll. Dies entspricht dem Datentyp *Character*, welcher einzelne Zeichen speichert.
- `<data type="positiveInteger"/>`
  Bei diesem *Terminalsymbol* handelt es sich um die Ableitungsanweisung zu einem Wert des Datentyps `positiveInteger`. ieser Datentyp wird mit anderen im W3C Report von Biron und Malhotra spezifiziert [BM04].
- `<text/>` leitet zuzüglich weiterer Paramenter zu einem Text ab.

Terminalsymbole dieser Kategorie werden im Folgenden als **generierte Terminale** bezeichnet.
In die dritte Kategorie fallen *Terminalsymbole*, welche direkt abgeleitet werden. Dazu zählen:

[9] https://relaxng.org/

- `<empty/>` leitet zum leeren Symbol (ε) ab.
- `<element [...]>` generiert einen neuen XML Knoten
- `<attribute [...]>` generiert ein Attribut im nächsten übergeordneten Element-Knoten.

Diese *Terminalsymbole* werden im Weiteren als **direkte Terminale** bezeichnet.

Im RNG Format sind in den Ableitungsregeln zusätzlich zu *Terminal-* und *Nichtterminalsymbolen* auch **Metasymbole** zulässig. Diese *Metasymbole* beinhalten die Steueroperatoren des Regelsatzes. Folgende *Metasymbole* entsprechen einem Äquivalent in den Operatoren regulärer Ausdrücke.

- Der Sternhüllenoperator (`*` Operator) wird durch `<zeroOrMore>` dargestellt.
- Die Plushülle (`+` Operator) folgt dieser Konvention als `<oneOrMore>` Tag.
- Eine Entscheidung (`|` Operator) wird als `<choice>` bezeichnet.
- Der Optionaloperator (`?` Operator) erhält den `<optional>` Tag als äquivalent.

Darüber hinaus gibt es weitere *Metasymbole*, welche zur genaueren Spezifikation von *.xml Dateien beitragen. Zu diesen zählen:

- Der `<interleave>`-Tag erlaubt Kommutativität in den ihm untergeordneten Elementen
- Der `<except>`-Tag exkludiert ihm untergeordnete Wörter aus dem Eingabebereich des Elternelements
- Der `<notAllowed>`-Tag hebt das jeweilige Elternelement auf. Davon ausgenommen ist die Konstellation bestehend aus einem `<choice>` Tag, welcher als Kindknoten ein `<notAllowed>` Element und mindestens ein anderes Element hat.

Diese werden im Verlauf der Arbeit als **spezielle Metasymbole** bezeichnet.

Um eine Grammatik in dieser Form für diese Arbeit einlesen zu können, wird sie als XML Baumstruktur betrachtet. Dabei werden alle einem Element x direkt untergeordenten Kindelemente x_{kind_i} als Teilbaum des Elements[10] x angeordnet. Im nächsten Schritt wollen wir der eingelesenen Grammtik probabilistische Informationen hinzufügen.

[10]Bei Betrachtung einer Baumstruktur auch Knoten genannt

3.2. Probabilistische Informationen

Um Daten zu erzeugen, welche eine höhere Realitätsnähe aufweisen sollen, bietet sich die Möglichkeit an, neben dem Verwenden von Zufallswerten auch probabilistische Informationen heranzuziehen. Diese Daten müssen vor Beginn des Fuzzings eingelesen und verarbeitet werden.
Die für diese Arbeit herangezogene Menge an Daten wurde der Arbeit von Le entnommen [Le17]. Diese Daten wurden mittels eines Apache Webcrawlers über einen Zeitraum von 3 Monaten erhoben.
Für diese Arbeit wurde sich dafür entschieden, die kontextfreien Informationen des Datensatzes zu verwenden. Dafür wurden zunächst alle Dateien des Datensatzes nacheinander betrachtet.
Im ersten Schritt der Verarbeitung wurden die Häufigkeiten verschiedener Knotenarten ermittelt. Dabei mussten die Namen der XML-Elemente und die Namen der XML-Attribute gezählt werden. Darüber hinaus wurden die verwendeten Werte jedes Attributes, nach Attributtyp geordnet gesammelt.
Im zweiten Schritt des Erfassungsprozesses wurden die Ergebnisse aller verarbeiteten Dateien zusammengetragen.
Im letzten Schritt wurden schließlich die probabilistischen Informationen aus den erfassten Daten zusammengestellt. Dabei musste die verwendete Grammatik durchlaufen werden. Trat bei diesem Durchlauf einer der folgenden Knoten `<choice>`, `<zeroOrMore>`, `<oneOrMore>` & `<optional>` auf, so wurden für diesen Knoten die probabilistischen Informationen zusammengetragen und annotiert. Für die Knotentypen wurden jeweils die unter Abbildung 6 dargestellten Informationen abgespeichert. Dabei konnte die Position eines Knotens durch die Struktur seiner Teilbäume identifiziert werden.

`<choice>`	Die absolute Häufigkeit jeder Option
	Die absolute Häufigkeit des Choice-Tags
`<zeroOrMore>`	Die kleinste Anzahl an Wiederholungen
	Die größte Anzahl an Wiederholungen
	Die durchschnittliche Anzahl an Wiederholungen
`<oneOrMore>`	Die kleinste Anzahl an Wiederholungen
	Die größte Anzahl an Wiederholungen
	Die durchschnittliche Anzahl an Wiederholungen
`<optional>`	Die absolute Häufigkeit der Fälle, wo der Teilbaum gewählt wurde
	Die absolute Häufigkeit dieses Choice-Tags

Abbildung 6: Knotentyp und jeweils dazu gespeicherte probabilistische Informationen

3.3. Fuzzer

Beim Fuzzer handelt es sich um den zentralen Teil der Software. In diesem Teil der Arbeit wollen wir seinen Aufbau und dessen Funktionsweise veranschaulichen. Wie im Abschnitt *Einleitung in Fuzzing* 2.3 geschildert, stehen mehrere Varianten des Fuzzings zur Verfügung. In Anbetracht der zugrunde liegenden Daten, wurde sich in dieser Arbeit für den Ansatz des *Modelbased Blackbox-Fuzzing* entschieden. Als Modell wird die, in Abschnitt 2.6 der Arbeit beschriebene, OpenDocument Grammatik im Relax NG Format verwendet.
Zu Beginn des Fuzzings werden die Startregel und alle weiteren Regeln der Grammatik indiziert. Die Startregel wird dabei, wie in Abbildung 16 im Anhang dargestellt, in einem `<start>`-Tag angegeben. Alle weiteren Regeln werden durch einen `<define name"[...]">`-Tag eingeleitet. Das *Nichtterminalsymbol* auf der linken Regelseite entspricht dabei dem Namen, welcher im `<define>`-Tag angegeben ist. Ein beispielhafter Auszug aus den Ableitungsregeln ist im Anhang unter Abbildung 17 dargestellt.
Sobald dieser Vorgang erfolgreich abgeschlossen wurde, kann mit dem Fuzzing begonnen werden. Dazu werden, mit der Startregel beginnend, alle aufgerufenen Regeln rekursiv abgeleitet. Dabei kann der Knoten im jeweils aktuellen Ableitungsschritt auf sechs verschiedene Weisen behandelt werden:

1. Handelt es sich beim aktuellen Knoten um ein *Wertterminal*, so wird dieses direkt zum String im `<value>`-Tag abgeleitet.

2. Ist der aktuelle Knoten ein *generiertes Terminal*, so werden dieser und der ihm untergeordnete Teilbaum an eine separate Funktion übergeben. Diese Funktion extrahiert alle vorhandenen Parameter für den Terminalstring aus dem Knoten und leitet diese an einen Generator weiter. Der Generator liefert daraufhin einen, anhand der Parameter generierten, Terminalstring zurück. Das *generierte Terminal* wird nun zum Text abgeleitet.

3. Ist der Knoten dieser Gruppe der *direkten Terminale* zuzuordnen, so wird dieser zusammen mit einem vordefinierten Schema abgeleitet. Die Ableitung eines Attributs erfolgt dabei in der Form `name="value"`, während ein Element zu Text der Form `<name [1]>[2]<name>` abgeleitet wird. An der Stelle `[1]` können dabei noch beliebig viele Attribute eingefügt werden und an der Stelle `[2]` können weitere Kindelemente untergeordnet werden.

4. Im Falle eines *Metasymbols* muss eine Entscheidung für den weiteren Ableitungsverlauf getroffen werden. Dazu wird, je nach Einstellung des Fuzzers, auf die probabilistischen Informationen zurückgegriffen. Je nach Art des Knotens können dabei die unter Abbildung 6 aufgeführten Informationen abgerufen und verwendet werden. Sollten für den jeweiligen Knoten keine Daten vorhanden sein oder sollte auf diese nicht zugegriffen werden, so werden Standardeinstellungen des Programms[11] verwendet. Diese Informationen

[11] Einstellbar in der Datei `Software/res/settings.xml`.

werden im Verlauf des *Fuzzings* genutzt, um Knotenabhängig eine Entscheidung für den weiteren Ableitungsverlauf zu erstellen. Dazu werden die zugeordneten probabilistischen Wahrscheinlichkeiten ausgewertet.

5. Handelt es sich beim aktuellen Knoten um ein *spezielles Metasymbol*, so wird dieses nach Art des Knotens abgearbeitet. Im Falle eines `<interleave>`-Tags werden alle Kindknoten zufällig neu angeordnet, um die mögliche Kommutativität zu simulieren.

6. In allen anderen Fällen werden auftretende Knoten ignoriert, da diese nicht relevant für den *Fuzzer* sind. Die Ausnahme bildet dabei der `<ref name="[...]">`-Tag. Dieser ist das Gegenstück des `<define name"[...]">`-Tags und stellt den Aufruf der definierten Regel mit dem selben Namen dar.

Um der Problematik vorzubeugen, dass der Fuzzer beim Generieren in eine endlose Ableitungsschleife verfällt, wird die maximale Tiefe des Ableitungsbaums und die maximale Anzahl der Schleifenwiederholungen durch einstellbare Parameter begrenzt. Nachdem ein Dokument vom Fuzzer generiert wurde, wird dieses an das *Orakel* weitergeleitet.

3.4. Orakel

Als Orakelverfahren wurde sich in diesem Projekt für das in 2.5 beschriebene *Crashtesting* entschieden. Grund dafür war, dass keine ausreichende Menge an Kontrollinstanzen zur Verfügung standen, welche im *Observation Testing* hätten verwendet werden können.
Da *LibreOffice* keine direkte Eingabeschnittstelle für Programmtests zur Verfügung stellt, wurde sich dafür entschieden, die Schnittstelle zum Linuxterminal zu nutzen.
Dazu wurde ein Bash Script verfasst, welches die vom Fuzzer generierte *content.xml* in eine vorbereitete Dateistruktur, siehe 2.6, einbettet und zu einem validen *.odt Dokument packt. Ein Auszug aus dem Bash Script ist dafür in Abbildung 7 dargestellt.

```
3  # Eingabeparameter
4  ######################################
5  # $1 Pfad zum gefuzzten Dokument
6  # $2 Name des gefuzzten Dokuments
7  # $3 Pfad zum LibreOffice Templateordner
8  # $4 Timeout für LibreOffice
```

```
13 cp $1/$2 $3/content.xml
14 cd $3
15 zip -r test.odt *
```

Abbildung 7: Packen der Daten zu einer *.odt Datei
Auszug aus `Data/in.pack.sh`

Im nächsten Schritt wird das fertig gepackte Dokument gegen die Konsolenschnittstelle von LibreOffice getestet. Um diesen Ablauf bestmöglich zu automatisieren, werden dabei folgende Parameter verwendet:

Parameter	Bedeutung
–headless	Startet das Programm ohne User Interface.
–invisible	Das Programm ist auch während des Starts nicht sichtbar.
–nologo	Das LibreOffice Logo wird beim Programmstart nicht angezeigt.
–norestore	Deaktiviert den Wiederherstellungsprozess nach einem Absturz.

Da *LibreOffice* keine maschinell verwertbare Rückmeldung liefert, ob eine Datei erfolgreich geöffnet wurde, wird der Fehlercode des Programms ausgewertet. Eine Liste der möglichen Fehlercodes ist im Anhang unter A.4 aufgelistet. Zusätzlich zu den Fehlercodes aus *LibreOffice* liefert auch die Unix `timeout` Funktion Rückgabewerte. Diese werden vom *Orakel* ignoriert, da sie nicht von *LibreOffice* stammen. Darüber hinaus lässt sich nicht feststellen, ob *LibreOffice* den Ladeprozess der zu öffnenden Datei bereits beendet hat. Zu diesem Zweck, wird die Prozesslaufzeit von *LibreOffice* durch einen einstellbaren Timer begrenzt. Wir gehen dabei davon aus, dass *LibreOffice* nach Ablauf des Timers[12] die Datei vollständig öffnen konnte.
Im erfolgreichen Testfall wird die vom Fuzzer erzeugte Eingabedatei verworfen. In diesem konnte von dem gewählten Orakel kein fehlerhaftes Verhalten festgestellt werden. Schlägt der Test jedoch fehl und *LibreOffice* liefert einen Fehler zurück, dann wird das erzeugte Dokument weiter von unserem *Minimizer* verarbeitet.

3.5. Minimizer

Der Minimizer ist dafür vorgesehen, bereits als fehlerhaft erkannte Dokumente auf die fehlerhaften Knoten zu reduzieren. Dazu wird der Syntaxbaum der *content.xml* Datei geladen. Im folgenden Schritt werden alle Knoten des Syntaxbaums durchlaufen. Der aktuelle Knoten wird dabei als *elter* und die Kindknoten als $kind_{0...n}$ bezeichnet. Für jeden Knoten e wird in einer Schleife über die Kindknoten iteriert. In einer Kopie des Syntaxbaums wird dabei jeweils der Teilbaum unter $kind_i$ temporär entfernt. Anschließend wird der kopierte Syntaxbaum erneut vom Orakel evaluiert.
Ist dabei der Ausgang des Tests **positiv**, so kann daraus gefolgert werden, dass sich alle verbleibenden Fehlerquellen im Teilbaum unter $kind_i$ befanden. Um die Fehler weiter einschränken zu können, fügen wir den Knoten $kind_i$ wieder an seine vorherige Position ein. Er wird in der nächsten Knotenebene weiter untersucht werden.
Führt der Test jedoch zu einem **negativen** Ergebnis, so besteht noch mindestens eine weitere Fehlerquelle im verbleibenden Syntaxbaum. Folglich hat der entfernte Teilbaum $kind_i$ keinen Einfluss auf den noch bestehenden Fehler. Als Konsequenz wird der Teilbaum $kind_i$ nicht wieder zum Syntaxbaum hinzugefügt. Er wurde erfolgreich reduziert.

[12] In der Entwicklung auf 15 Sekunden festgelegt.

Im Folgenden wollen wir das Vorgehen einmal auf einem beispielhaften Syntaxbaum anwenden. In dem Beispiel, dargestellt unter den Abbildungen 10 – 13 im Anhang, liegt der eine fehlerhafte Knoten im **Teilbaum 1b**. Als Ausgangsituation gehen wir vom Zustand in Abbildung 10 aus.
Zu Beginn wählen wir den $Wurzelknoten$ als aktuellen Knoten $elter$. Die Kindknoten $kind_{0,1}$ sind dabei $Teilbaum1$ und $Teilbaum2$. Nun wird über die Kindknoten iteriert.

i = 0 Wir betrachten nun $Teilbaum1$. Wie in Abbildung 11 dargestellt, wird dieser Teilbaum temporär entfernt.
Im Anschluss wird der Syntaxbaum vom Orakel getestet. Das Ergebnis des Orakels ist in diesem Beispiel positiv.
Wir fügen also den $Teilbaum1$ wieder dem Syntaxbaum hinzu. Dabei wird die Ausgangssituation aus Abbildung 10 wiederhergestellt.

i = 1 In dieser Iteration betrachten wir den $Teilbaum2$. Auch hier wird der Teilbaum temporär entfernt. Dabei erhalten wir die Situation, wie in Abbildung 12 dargestellt.
Das Orakel liefert für diese Variante des Syntaxbaums ein negatives Ergebnis.
Daraus wird gefolgert, dass der entfernte Teilbaum keinen Zusammenhang zum fehlerhaften Teilbaum 1b hat. Somit wird der Teilbaum nicht wieder dem Syntaxbaum hinzugefügt.
Abschließend erhalten wir den in Abbildung 13 dargestellten Syntaxbaum.

4. Ergebnisse

Da im Fuzzing zwangsläufig mit Zufallszahlen gearbeitet wird, können die Ergebnisse bei Reproduktion des Vorgehens abweichen. Die erarbeitete Software verwendet dazu annähernd gleichverteilte Pseudozufallszahlen im Intervall $[0.0, 1.0)$.[13] Zur Analyse wurden 30 Dateien mit zufälligem Fuzzing und 30 weitere Datein mit probabilistischen Fuzzing erzeugt.
Im Verlauf der Arbeit wurde das entwickelte Programm in verschiedenen Konfigurationen verwendet, um Fehler in *LibreOffice* zu identifizieren. Im ersten Versuchsdurchlauf des Programms wurden zufällige Eingaben gewählt. Dazu wurde das Programm mit `FUZZING_MODE="R"` gestartet. Die von dieser Konfiguration erzeugten Dateien waren syntaktisch korrekt. Dabei wurden jedoch keine Fehler in *LibreOffice* festgestellt. Alle generierten Eingaben konnten ohne das Auftreten von Fehlern geöffnet werden.
Vor Beginn des zweiten Tests wurden der Grammatik probabilistische Informationen hinzugefügt. Als Datengrundlage für diese wurden *content.xml* Dateien aus der Arbeit von Le[Le17] verwendet. Mit dieser Vorbereitung wurde eine zweite Konfiguration des Programms getestet. Dazu wurde die Einstellung `FUZZING_MODE="P"` gewählt. Auch diese Dateien wurden von *LibreOffice* als syntaktisch Korrekt akzeptiert. Es konnten auch mit dieser Konfiguration keine Fehler in *LibreOffice* provoziert werden.
Bei der Analyse der vom Fuzzer generierten Daten konnten einige gravierende Unterschiede zu den Daten aus dem ersten Versuchsdurchlauf festgestellt werden. So wiesen die Daten aus dem zweiten Versuchsdurchlauf meist eine geringere Größe auf, wie im Boxplot in Abbildung 8 dargestellt ist. Darüber hinaus war zu beobachten, dass Dateien des probabilistischen Datensatzes eine wesentlich flacherere Struktur aufwiesen. Dies wird in Abbildung 9 dargestellt. Auf die Gründe für diese unterschiedliche Struktur wollen wir im folgenden Abschnitt genauer eingehen.

[13] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Math.html#random()

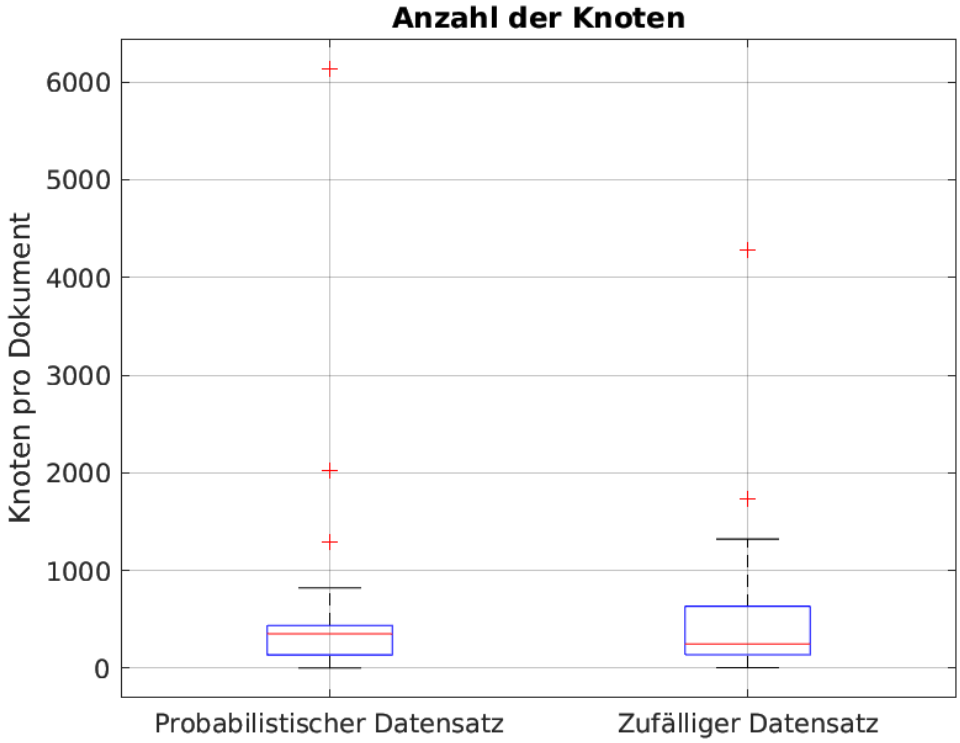

Abbildung 8: Anzahl der Knoten aus dem probabilistischen und dem zufälligen Datensatz

5. Diskussion der Ergebnisse

Wie der Vergleich von rein zufällig erzeugten Datensätzen und jenen, welche mit probabilistischen Informationen erzeugt wurden zeigt, weisen beide Varianten deutliche Unterschiede in ihrer Struktur auf. So sind die zufällig generierten Dateien (folgend als *zufällige Datensätze* bezeichnet) durchschnittlich größer als jene, welche durch die Verwendung probabilistischer Informationen generiert wurden (folgend als *probabilistische Datensätze* bezeichnet). Diese Ergebnisse sind in Abbildung 8 grafisch dargestellt. Ein Grund dafür ist, dass für die *zufälligen Datensätze* hohe Wahrscheinlichkeiten für optionale Knoten und Schleifendurchläufe verwendet wurden. Die Wahrscheinlichkeit für die Auswahl mittels des `<choice>`-Tags sind gleichverteilt. Darüber hinaus kann auch die Einschränkung durch die probabilistischen Wahrscheinlichkeiten Einfluss auf die Größe der *probabilistischen Datensätze* haben.
Eine weitere Beobachtung beim Vergleich von *zufälligen* und *probabilistischen Datensätzen* waren die stark von einander abweichenden Ableitungstiefen. So wiesen *probabilistische Datensätze* eine wesentlich geringere Ableitungstiefe auf, als es bei den *zufälligen Datensätzen* der Fall war. Dies ist in Abbildung 9 dargestellt. Dies ist bei den *zufälligen Datensätzen* größtenteils darauf zurückzuführen, dass an vielen Stellen der Grammatik zyklische Ableitungen möglich sind. Wie im beispielhaften Auszug, unter A.2 dagestellt, leicht zu sehen ist, wiederholen sich die meisten dargestellten Knoten mehrfach. Dies war bereits bei der Konstruktion des Fuzzers eine Herausforderung. So wurde, um diesem

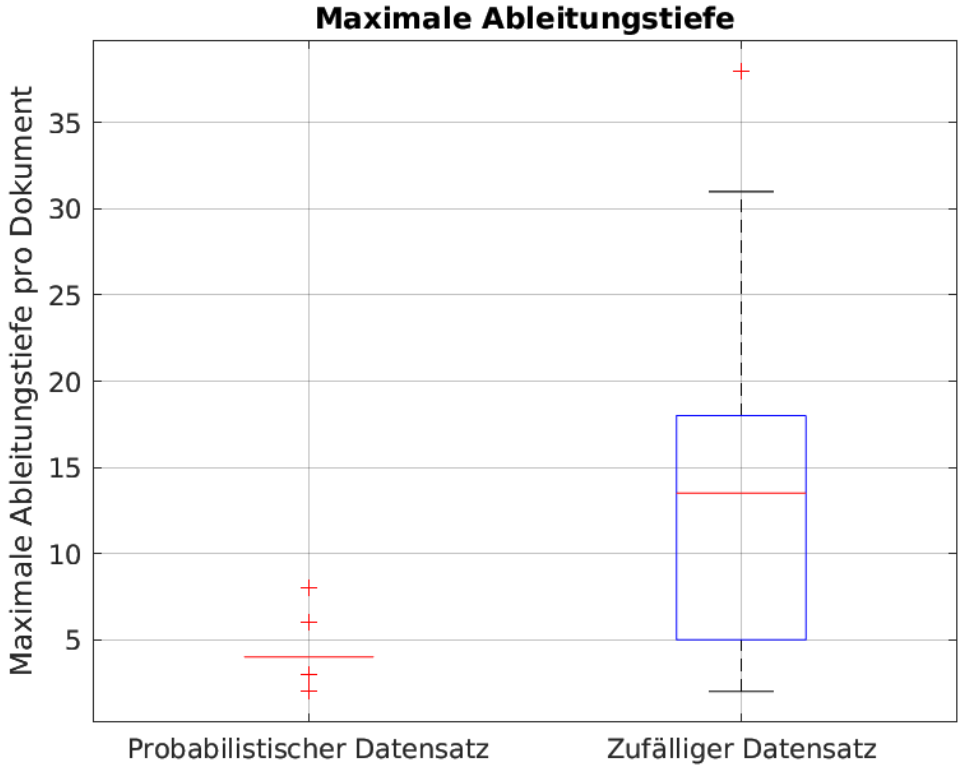

Abbildung 9: Maximale Ableitungstiefe des probabilistischen und des zufälligen Datensatzes

Problem vorzubeugen, eine Einstellungsmöglichkeit [14] für die maximale rekursive Ableitungstiefe und eine maximale Beschränkung der Schleifendurchläufe hinzugefügt. Bei *probabilistischen Datensätzen* ist ein solches Verhalten jedoch nicht zu beobachten. Dies lässt sich darauf zurückführen, dass auch in den probabilistischen Datensätzen, welche in 3.2 verwendet wurden, ein solches Verhalten nicht gegeben ist.
Im Folgenden wollen wir schließlich auf mögliche Bedenken in den einzelnen Entwicklungskomponenten eingehen. So zählen das *Orakel* und der *Minimizer* zu den kritischsten Komponenten, da in diesen Programmteilen das Abweichungspotenzial am größten ist.
So wird der *Orakel* Prozess nach Ablauf eines Timers als erfolgreich angenommen. Dies geschieht, da von *LibreOffice* keine Rückmeldung über das erfolgreiche Öffnen einer Datei erfolgt. Die verwendete Eintellung des Timers wurde dabei über ein Experiment ermittelt. Bei diesem Experiment, wurden aus den für die Ermittlung der probabilistischen Daten zur Verfügung stehenden Datensätzen, zufällig 20 Datensätze ausgewählt. Für diese wurde im Anschluss die Zeit zum Öffnen des Dokuments gemessen. Das maximale Ergebnis dieser Testreihe wurde auf den nächsten, ganzzahlig durch 5 teilbaren Messwert aufgerundet. In unserem Testfall ergab sich so ein resultierenden Wert von 10 Sekunden. Dieser Wert ist jedoch abhängig von den gewählten Datensätzen und der Systemleistung, er kann entsprechend für eine andere Auswahl von Datensätzen variieren. Zu diesem

[14] Einstellbar in der Datei `Software/res/settings.xml`

Zweck wurde als Zeitpuffer der Timerwert um 50% erhöht. Woraus ein Wert von 15 Sekunden resultierte.
Ein weiterer Aspekt des *Orakels* ist, dass mit der gewählten Variante ausschließlich Programmabstürze festgestellt werden können.3.4 Aufgrund der Verwendung von zufällig erstellten Eingabedaten bietet sich keine Möglichkeit auch passende korrekte Testfallergebnisse zu bestimmen. Dies ist nur möglich, falls bereits eine Kontrollinstanz existiert, welche dieses korrekte Testfallergebnis ermitteln kann. Somit wurde der bestmögliche Ansatz gewählt.
Im Weiteren wird auch der *Minimizer* betrachtet. Dieser bietet die Möglichkeit, Einzelfehler[15] eindeutig zu ermitteln. Da das *Orakel*, wie in 2.5 beschrieben, nur genau zwei Zustände ($CT_{t_i} = 0$ und $CT_{t_i} = 1$) zurück liefern kann, kann auch der *Minimizer* nur nach einem fehlerverursachenden Knoten minimieren. Dies stellt jedoch keine Einschränkung des Programms dar, da selbst beim Auftreten von Einzelfehlern die Datei als fehlerhaft identifiziert wird.

[15]Dateien, welche nur genau einen fehlerverursachenden Knoten aufweisen.

6. Zusammenfassung & Ausblick

Das Ziel dieser Arbeit war die Erstellung einer Anwendung. Dieses Programm hatte die Aufgabe, durch *Fuzzing* erstellte Eingabedaten für *LibreOffice* zu generieren. Dabei wurde Wert darauf gelegt, dass das Sammeln und Verwenden von probabilistischen Daten unterstützt wird. Ein weiteres Augenmerk lag auf der Isolation von möglichen Fehlern. Dazu wurde ein Minimizer integriert, welcher potenziell fehlerhafte Datenknoten erkennen kann. Zum Sammeln der probabilistischen Informationen wurde ebenfalls eine entsprechende Funktionalität in die Software integriert.
Wie die Tests der Anwendung zeigten, konnten bereits ohne die Verwendung von probabilistischen Daten valide Eingabedaten erzeugt werden. Diese Daten wurden bereits als syntaktisch korrekt durch das *Orakel* akzeptiert. Wie sich im Verlauf der Arbeit herausstellte, erwiesen sich diese Daten jedoch als nicht sehr realitätsnah. Häufig bildeten diese Dateien unrealistischen Knotenfolgen (Vgl. A.2). Nach der Implementation und Verwendung von probabilistischen Informationen stieg die Qualität der Eingaben an. Durch Optimierungsarbeiten an der Software konnte der Fuzzer weiter verbessert werden. Mit der im Rahmen dieser Arbeit erstellten Software können minimierte fehlerhafte *LibreOffice* Dokumente automatisiert erstellt werden. Der Folgeprozess, das endgültige Identifizieren von Fehlern, ist dabei bis zum Abschluss der Arbeit ein nicht automatisierter Prozess. Dieses manuelle Vorgehen könnte im Rahmen weiterer, folgender Forschungsprojekte bearbeitet werden. Darüber hinaus kann das Programm auch um die Möglichkeit erweitert werden, die von Pavese et al. vorgestellten *inputs from hell* zu verwenden.
Mit dieser Arbeit konnte gezeigt werden, dass die Entwicklung automatisierter Testgenerierungssoftware ein umfassendes Forschungsfeld darstellt. Mit weiteren Ansätzen, wie der Verwendung genetischer Algorithmen oder einer automatisierten Fehlerklassifikation, bietet dieses Themengebiet ausreichend Potenzial für weitere Arbeiten.

Literatur

[BE96] Helmut Balzert and Christof Ebert. *Lehrbuch der Software-Technik*. Spektrum, Akad. Verlag, 1996.

[BFF08] Alexandre Blonce, Eric Filiol, and Laurent Frayssignes. Portable document format (pdf) security analysis and malware threats. Presentations of Europe BlackHat 2008 Conference, 2008.

[BM04] Paul V Biron and Ashok Malhotra. XML schema part 2: Datatypes second edition. "W3C recommendation", October 2004. "http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/".

[BP15] Marcel Böhme and Soumya Paul. A probabilistic analysis of the efficiency of automated software testing. *IEEE Transactions on Software Engineering*, 42(4):345–360, 2015.

[BPR17] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[Cla01] James Clark. Relax ng specification. *http://www.oasis-open.org/committees/relax-ng/spec-20011203.html*, 2001.

[DN81] Joe W Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*, pages 179–183. IEEE Press, 1981.

[FB96] Ned Freed and Nathaniel S. Borenstein. Multipurpose internet mail extensions (mime) part one: Format of internet message bodies. RFC 2045, RFC Editor, November 1996. `http://www.rfc-editor.org/rfc/rfc2045.txt`.

[FM00] Justin E Forrester and Barton P Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*, volume 4, pages 59–68. Seattle, 2000.

[GKL08] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[GLM+08] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[HHZ12] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 445–458, 2012.

[JLM92] Frederick Jelinek, John D Lafferty, and Robert L Mercer. Basic methods of probabilistic context free grammars. In *Speech Recognition and Understanding*, pages 345–360. Springer, 1992.

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KTT14] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. Combining stochastic grammars and genetic programming for coverage testing at the system level. In *International Symposium on Search Based Software Engineering*, pages 138–152. Springer, 2014.

[KTT17] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. Generating valid grammar-based test inputs by means of genetic programming and annotated grammars. *Empirical Software Engineering*, 22(2):928–961, 2017.

[Le17] Dinh Hung Le. Grammar-mining for opendocument files. Bachelorarbeit, Humboldt Universität zu Berlin, 2017.

[MMS99] Christopher D Manning, Christopher D Manning, and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

[ND12] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[Ney08] John Neystadt. Automated penetration testing with white-box fuzzing, 2008. http://msdn.microsoft.com/en-us/library/cc162782.aspx - Retrieved 02-06-2019.

[ODF11] Open document format for office applications, 2011. https://docs.oasis-open.org/office/v1.2/OpenDocument-v1.2-part1.html - Retrieved 07-02-2019.

[PSH+18] Esteban Pavese, Ezekiel Soremekun, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. Inputs from hell: Generating uncommon inputs from common samples. *arXiv preprint arXiv:1812.07525*, 2018.

[Pur72] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[RCA+14] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 861–875, 2014.

[RL10] Hassan Reza and Suhas Lande. Model based testing using software architecture. In *2010 Seventh International Conference on Information Technology: New Generations*, pages 188–193. IEEE, 2010.

[SBH+94] Yasubumi Sakakibara, Michael Brown, Richard Hughey, I Saira Mian, Kimmen Sjölander, Rebecca C Underwood, and David Haussler. Stochastic context-free grammers for trna modeling. *Nucleic acids research*, 22(23):5112–5120, 1994.

[Wei09] Rob Weir. Opendocument format: The standard for office documents. *IEEE Internet Computing*, 13(2):83–87, 2009.

[Wir13] Niklaus Wirth. *Compilerbau: Eine Einführung*, volume 36. Springer-Verlag, 2013.

[WPC06] Wen-Li Wang, Dai Pan, and Mei-Hwa Chen. Architecture-based software reliability modeling. *Journal of Systems and Software*, 79(1):132–146, 2006.

A. Anhang

A.1. Beispielhafte Baumstruktur

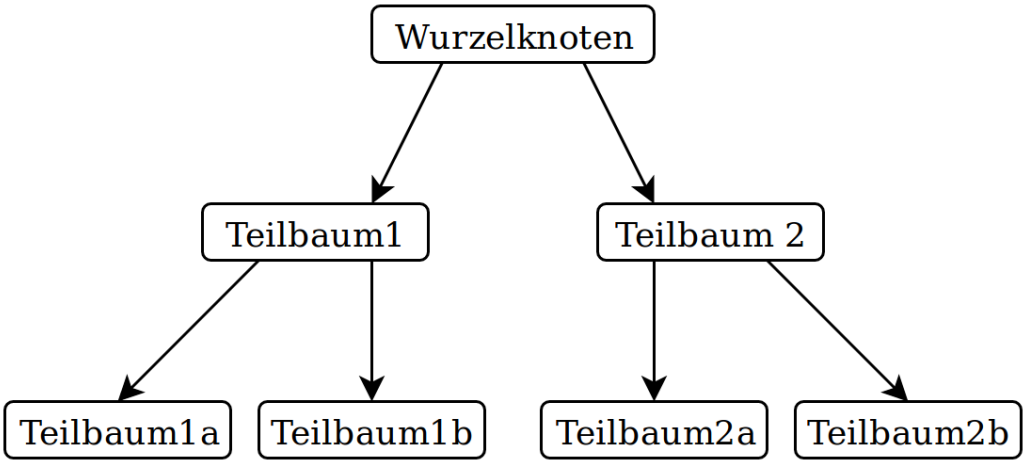

Abbildung 10: Vollständige Baumstruktur

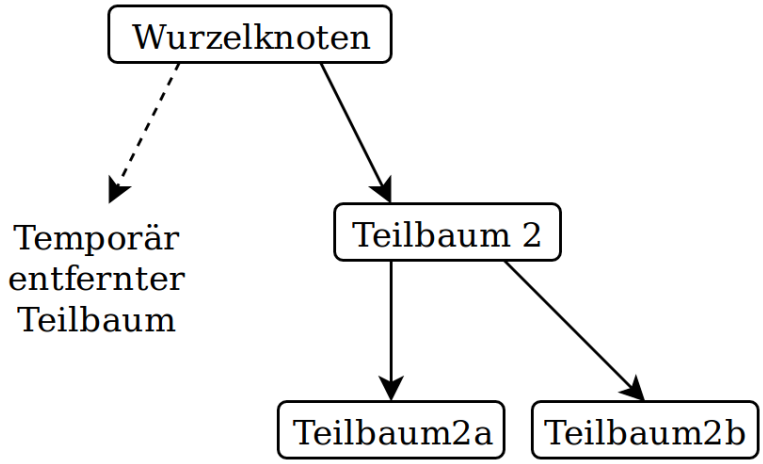

Abbildung 11: Der Teilbaum 1 wurde zu Testzwecken entfernt

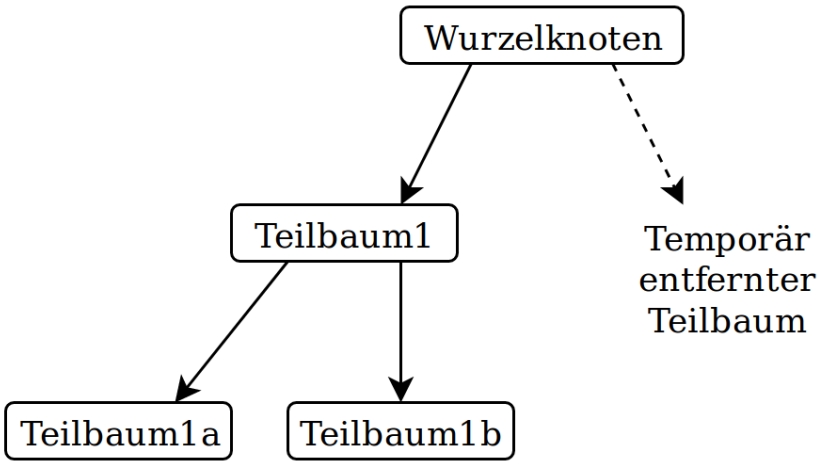

Abbildung 12: Der Teilbaum 2 wurde zu Testzwecken entfernt

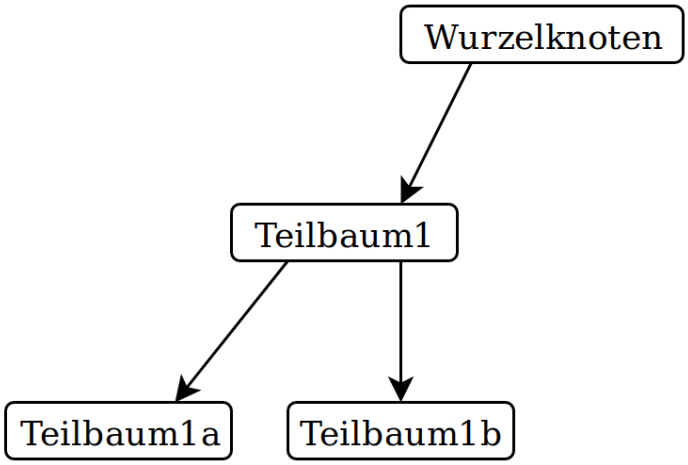

Abbildung 13: Zustand des Syntaxbaums nach Durchlauf der Schleife

A.2. Unrealistische Knotenstruktur

Quellcodeauszug einer durch den Fuzzer generierten Datei. Dabei wurden keine probabilistischen Daten verwenden. Alle Entscheidungen im Erstellungsprozess wurden dabei komplett zufällig getroffen.

```
<office:settings>
 <config:config-item-set xmlns:config="[...]">
  <config:config-item-map-named>
   <config:config-item-map-entry>
    <config:config-item-map-indexed>
     <config:config-item-map-entry>
      <config:config-item-set>
       <config:config-item-map-indexed>
        <config:config-item-map-entry>
         <config:config-item-map-indexed>
          <config:config-item-map-entry>
           <config:config-item-map-named>
            <config:config-item-map-entry>
[...]
            </config:config-item-map-entry>
           </config:config-item-map-named>
          </config:config-item-map-entry>
         </config:config-item-map-indexed>
        </config:config-item-map-entry>
       </config:config-item-map-indexed>
      </config:config-item-set>
     </config:config-item-map-entry>
    </config:config-item-map-indexed>
   </config:config-item-map-entry>
  </config:config-item-map-named>
 </config:config-item-set>
</office:settings>
```

Auszug aus `Data/random_samples/content0.xml`

A.3. Abänderung der Grammatik

```
73 <start>
74   <choice>
75     <ref name="office-document"/>
76   </choice>
77 </start>
```

Abbildung 14: Neue Fassung der verwendeten Grammatik.

```
73 <start>
74   <choice>
75     <ref name="office-document"/>
76     <ref name="office-document-content"/>
77     <ref name="office-document-styles"/>
78     <ref name="office-document-meta"/>
79     <ref name="office-document-settings"/>
80   </choice>
81 </start>
```

Abbildung 15: Ursprüngliche Fassung der verwendeten Grammatik.

A.4. LibreOffice Fehlercodes

Die Liste an möglicher Fehlercodes in LibreOffice Calc.[16] Diese Fehlercodes sind auch in *LibreOffice* Write gültig:

501 – Ein Zeichen in einer Formel ist ungültig.

502 – Das Argument einer Funktion ist ungültig.

503 – Eine Berechnung führt zu einem Überlauf des Wertebereichs.

504 – Ein Funktionsparamter ist nicht gültig.

508 – Eine geöffnete Klammer wurde nicht geschlossen.

509 – Ein Mathematischer Operator fehlt.

510 – Eine Variable in einer Rechnung fehlt.

511 – Eine Funktion erfordert mehr Varialben.

512 – Eine Formel ist zu lang. (Dies erzeugt einen Compilerfehler.)

513 – Eine Zeichenkette ist zu lang. (Dies erzeugt einen Compilerfehler.)

514 – Eine Sortieroperation überschreitet die größe es Rechenstapels.

516 – Im Rechenstapel wird eine Matrix erwartet, obwohl keine verfügbar ist.

517 – Es wurde Code gefunden.

518 – Eine Variable ist nicht verfügbar.

519 – Eine Formel gibt einen Wert zurück, der nicht der Definition entspricht.

520 – Der Compiler erzeugt unbekannten Code.

521 – Syntax Fehler; Kein Ergebnis vorhanden.

522 – Es gibt einen zirkulären Bezug innerhalb des Programms.

523 – Ein Rechenprozess konvergiert nicht zu einem Zielwert.

524 – Compilerfehler; Ein Beschreibungsname wurde nicht aufgelöst.

525 – Ein Bezeichner kann nicht aufgelöst werden.

526 – Ein veralteter Fehlercode, er wird nicht mehr verwendet.

527 – Beim Interpreter tritt ein Überlauf auf.

532 – Division durch Null.

[16] https://help.libreoffice.org/Calc/Error_Codes_in_Calc/de

Darüber hinaus können weitere Fehler auftreten, die vom Orakel nicht als *LibreOffice* Fehler eingeordnet werden können.

124 – Option `--preserve-status` wurde nicht gesetzt.

125 – Der Timeout ansich schlug fehl.

126 – Der auszuführende Befehl wurde gefunden, konnte aber nicht ausgeführt werden.

127 – Der auszuführende Befehl konnte nicht gefunden werden.

137 – Anstelle des `SIGTERM` Signals wurde `SIGKILL` gesendet.

255 – *LibreOffice* wurde mittels `SIGTERM` vom `timeout` Befehl beendet.

A.5. Auszüge der Grammatik

```
73    <start>
74      <choice>
75        <ref name="office-document"/>
76      </choice>
77    </start>
```

Abbildung 16: Die Startregel der Grammatik:
Auszug aus `Data/OpenDocument-v1.2-os-schema.rng`

```
354     <define name="office-image-content-prelude">
355       <empty/>
356     </define>
357     <define name="office-image-content-main">
358       <ref name="draw-frame"/>
359     </define>
360     <define name="office-image-content-epilogue">
361       <empty/>
362     </define>
363     <define name="office-settings">
364       <optional>
365         <element name="office:settings">
366           <oneOrMore>
367             <ref name="config-config-item-set"/>
368           </oneOrMore>
369         </element>
370       </optional>
371     </define>
372     <define name="config-config-item-set">
373       <element name="config:config-item-set">
374         <ref name="config-config-item-set-attlist"/>
375         <ref name="config-items"/>
376       </element>
377     </define>
378     <define name="config-items">
379       <oneOrMore>
380         <choice>
381           <ref name="config-config-item"/>
382           <ref name="config-config-item-set"/>
383           <ref name="config-config-item-map-named"/>
384           <ref name="config-config-item-map-indexed"/>
385         </choice>
386       </oneOrMore>
387     </define>
```

Abbildung 17: Beispielhaft dargestellte Regeln
Auszug aus `Data/OpenDocument-v1.2-os-schema.rng`

A.6. Dateizugriff

Alle Daten, Programme, Ergebnisse etc., welche im Rahmen dieser Arbeit erstellt wurden, sowie die Arbeit selbst in digitaler Fassung, stehen unter `https://scm.cms.hu-berlin.de/bucherda/ba-thesis` zur öffentlichen Einsicht zur Verfügung.

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 31. Juli 2019 .....