Brandenburgische Technische Universität Cottbus-Senftenberg
Institut für Informatik
Fachgebiet Praktische Informatik/Softwaresystemtechnik

Masterarbeit

Integrationstesten von GraphQL mittels Prime-Path Abdeckung

Integration testing of GraphQL using Prime-Path Coverage

Tom Lorenz
MatrikelNr.: 3711679
Studiengang: Informatik M. Sc.

Datum der Themenausgabe: 16.05.2023
Datum der Abgabe: (hier einfügen)

Gutachter 1: Prof. Dr. rer. nat. Leen Lambers
Gutachter 2: Prof. Dr. rer. nat. Gerd Wagner
Betreuer: M. Sc. Lucas Sakizloglou

Eidesstattliche Erklärung

Der Verfasser erklärt, dass er die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt hat. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht. Wörtlich und inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitens zitiert. Die Arbeit ist nicht in gleicher oder vergleichbarer Form (auch nicht auszugsweise) im Rahmen einer anderen Prüfung bei einer anderen Hochschule vorgelegt oder publiziert worden. Der Verfasser erklärt sich zudem damit einverstanden, dass die Arbeit mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate überprüft wird.

.....	.....
Ort, Datum	Unterschrift

Inhaltsverzeichnis

1 Kurzfassung

Im Zuge der digitalen Transformation nimmt die Anzahl von Softwareanwendungen rasant zu [1]. Insbesondere durch das Internet of Things und die generelle fortschreitende Vernetzung diverser Geräte nimmt die Netzwerklast stark zu [2]. Bisheriger Standard für Kommunikation von Geräten über das Internet waren REST-APIs [3, vgl. Introduction]. Diese haben gewisse Limitierungen wie zum Beispiel: Ineffizienz durch Overfetching/Underfetching, Anzahl an Requests, Versionierung, Komplexität und vieles mehr [3]. Mit der Veröffentlichung von GraphQL in 2015 wurde ein Konkurrent zu REST ins Leben gerufen, der diese Probleme beheben kann.

Durch GraphQL lässt sich insbesondere die Netzwerklast reduzieren, da eine GraphQL-Request, im Gegensatz zu REST, mehrere Anfragen in einer einzigen HTTP-Request zusammenfassen kann [4] und dabei auch nur die wirklich gewünschten Daten überträgt [3, vgl. Advantages of GraphQL APIs]. Dadurch, dass jedoch die wachsende Anzahl von Softwareanwendungen auch in immer kritischere Bereiche des Lebens vordringt, ist es wichtig, die Qualität der Software sicherzustellen [5, S. 16]. Eine Methodik zum Sicherstellen der korrekten Funktionalität von Software ist das Testen von Software im Sinne von Validierungstests, die sicherstellen sollen, dass die Software vorher definierte Szenarien nach Erwartung behandelt. Für REST-APIs existieren zahlreiche Tools, die solche Validierungstests automatisch übernehmen können, wohingegen es noch einen Mangel an Tools dieser Art für GraphQL gibt [6, vgl. Introduction].

Im Rahmen der internationalen Konferenz für Automatisierung von Softwaretests IEEE/ACM 2021 wurde ein Paper veröffentlicht, das eine Methode vorstellt, wie GraphQL-APIs mithilfe von *Property-based Tests* [6] automatisch getestet werden können. Property-based bezieht sich darauf, dass die Eigenschaften eines Objektes genutzt werden, um diese zu testen. Diese Methode generiert, der Datenstruktur angepasste, zufällige Tests und bietet so eine Möglichkeit, Fehler zu entdecken, ohne ein tiefgreifendes Wissen des zu testenden System zu besitzen [6, vgl. Proposed Method]. Die zufallsbasierte Testgenerierung weist allerdings einige Schwachstellen auf. So kann sie nicht garantieren, dass die generierten Tests zu jeder Zeit eine gute Abdeckung der GraphQL-API haben, denn es können einzelne Routen der API komplett ausgelassen werden. Es sind Testszenarios denkbar, die sehr viele False-Positives durchlassen und somit die Qualität der Software nicht ausreichend sicherstellen können. GraphQL ermöglicht außerdem einen potenziell unendlichen Suchraum für die Tests. Um den potenziell unendlichen Suchraum einzugrenzen wurde ein Rekursionslimit eingeführt, dass die Testlänge limitiert. Diese Limitierung führt dazu, dass die Testabdeckung nur bis zu einem bestimmten Komplexitätsgrad des zu testenden System ausreichend ist. Mit dieser Arbeit wurde

ein anderer Ansatz für die automatisierte Testgenerierung untersucht, um die Testabdeckung verlässlich zu verbessern. Ein Schwerpunkt der Arbeit lag hierbei darin, zuerst die theoretische Verknüpfung von GraphQL mit der Graphentheorie herzustellen.

Das gewonnene Wissen wird für eine Analyse verwendet, welches Graphabdeckungskriterium sich gut für Testgenerierung eignen würde. Die erlangten Kenntnisse halfen bei der Entwicklung einer Methode zum Generieren von Tests. Um die Methode zu validieren wurde ein Prototyp entwickelt, der die Methode automatisiert. In zwei Experimenten konnte mit dem Prototyp gezeigt werden, dass die entwickelte Methode in der Lage ist, Fehler in GraphQL-APIs zu finden. Insgesamt war es möglich, einen Punkt aus [6, VI. Future Work] zu erweitern, indem für eine bessere Graphabdeckung gesorgt wurde.

2 Einleitung

In diesem Kapitel wird an das Thema herangeführt und die Motive werden dargestellt. Es wird definiert, welche Ziele in dieser Arbeit verfolgt werden. Abschließend folgt eine Übersicht über die Kapitelstruktur.

2.1 Motivation

Mit einer steigenden Nutzung von GraphQL wird es immer wichtiger, Tests für GraphQL-Schnittstellen zu entwickeln, damit eine gute Softwarequalität sichergestellt werden kann. Die Entwicklung von Tests kann manuell oder automatisch geschehen. Bei Unit-Tests, also Tests für einzelne Funktionen, kann ein Programmierer selbst entscheiden, ob er diese manuell erstellt oder von einem Tool automatisch generieren lassen will. Integrations-Tests, also Tests, die Kombinationen von Interaktionen in Modulen miteinander testen, haben oft einen sehr großen Testraum, sodass ein manuelles Erstellen dieser Tests fehleranfällig und langwierig ist. Im Folgenden wird der Begriff API öfters genutzt werden, die Klärung des Begriffs findet sich in Kapitel 3.2. Für REST-APIs existieren schon automatische Integrationstesttools wie zum Beispiel: EvoMaster [7], QuickREST [8] oder RESTTESTGEN [9]. GraphQL-APIs haben leider noch einen Mangel an solchen automatischen Testtools. Im Rahmen der internationalen Konferenz für Automatisierung von Softwaretests IEEE/ACM 2021 wurde mit *Automatic Property-based Testing of GraphQL APIs* [6] eine Methode vorgestellt, die diesen Mangel angehen soll. Es wurde eine Methode entwickelt, die aus dem GraphQL-Schema, also der Beschreibung der Datenstruktur der API, Tests zufällig generiert und damit versucht, Fehler in der Programmierung zu finden. Die entwickelte Methode arbeitet nach dem in Abbildung 2.1 gezeigten Prinzip.

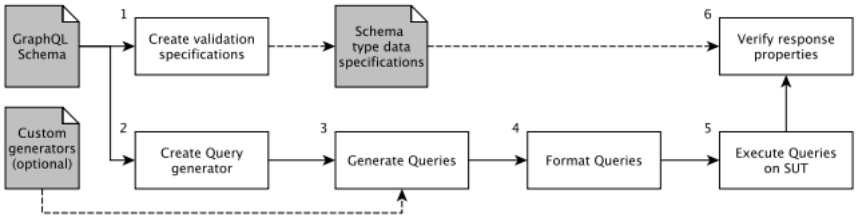

Abbildung 2.1: Methode von [6]

Es wird aus einem GraphQL-Schema ein Testgenerator entwickelt. Dieser kann aus der Typspezifikation, die GraphQL vorgibt, valide GraphQL-Queries entwickeln und diese mit verschiedenen Argumenten anreichern. Die generierten Queries stellen die entwickelten Tests dar. Das Besondere an GraphQL ist jedoch, dass es einen Graphen umsetzt. Ein einfaches Schema lässt sich in den Graphen aus Abbildung 2.2 übersetzen.

```
type Query {
  book(id: ID): Book
}

type Book {
  id: ID
  title: String
  sequel: Book
}
```

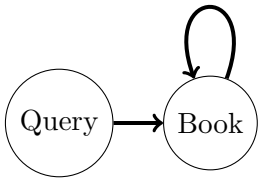

Abbildung 2.2: GraphQL-Schema als Graph

Generell starten alle GraphQL-Queries im Query-Type. Erlaubte Anfragen sind dann alle Pfade, deren Ursprung im Query Knoten liegt, wobei Limitierungen implementiert werden können. Property-based Testing nimmt die definierten Felder im Query-Type und geht die Pfade, welche sich im Graphen ergeben, zufällig ab. Nach einer bestimmten Anzahl an zufälligen Iterationen wird die Query aus dem erlangten Pfad generiert und ausgeführt. Wird jedoch ein wesentlich größeres Schema, zum Beispiel die GraphQL-API von GitLab [10] genutzt, so ist schnell zu erkennen, dass der Graph so komplex wird, dass eine zufällige Pfadgenerierung zu unzuverlässig und ineffizient ist, um eine große Struktur zuverlässig zu testen. Ein ähnlicher Sachverhalt findet sich in der Testgenerierung für Programmcode. Dieser kann sehr komplex werden und es müssen Strategien gefunden werden, um diese effizient und zuverlässig zu testen. Ein häufig verwendeter Ansatz ist es, den Code in einen Kontrollflussgraphen zu überführen, bei dem die Knoten Anweisungen oder Operationen darstellen und die Kanten den möglichen Pfaden entsprechen, die während der Ausführung des Programms genommen werden können. Hierbei wurde schon erhebliche Arbeit geleistet und diverse Kriterien entwickelt, wie eine gute Testabdeckung erreicht wird. An dieser Stelle sei insbesondere an *Introduction to Software Testing* [5] verwiesen. In [5] wird die Graphabdeckung erarbeitet und praktisch gezeigt, wie sie helfen kann, um Tests zu generieren. Es werden verschiedene Kriterien vorgestellt, die den Graphen auf unterschiedliche Art und Weise betrachten. Ziel ist es, zu zeigen, dass das in [5] erarbeitete Wissen nicht nur für die Testentwicklung von Programmcode zielführend ist, sondern auch für die Testgenerierung von anderen Graphstrukturen verwendet werden kann, in diesem Fall für die Testgenerierung für GraphQL-APIs. Der Fokus wird sich auf die PrimePfad-Abdeckung [5, vgl. Criterion 2.4] richten, da zu vermuten ist, dass diese Abdeckung einen guten Mittelweg zwischen Testgenauigkeit, Fehlerfindung und Effizienz bietet.

2.2 Umsetzung

Zuallererst wird in dieser Arbeit die grundlegende Theorie in Kapitel 3 definiert und in Bezug zueinander gesetzt. Es wird mit der Definition einiger Konzepte aus der Graphentheorie in Abschnitt 3.1 begonnen. Darauffolgend kommt eine präzise Betrachtung von GraphQL in Absatz 3.3. Die Erkenntnisse beider Absätze werden dann in Absatz 3.4 kombiniert. Abschließend für die grundlegende Theorie wird in Absatz 3.5 das Thema Softwaretests eingeführt. Die zuvor erarbeitete Theorie wird dann im Kapitel 4 genutzt, um einen Zugang zu schaffen, der zeigt, dass Graphabdeckungskriterien nutzbar für GraphQL sind. Ein Überblick über den aktuellen Stand der Forschung wird in Kapitel 5 gegeben. Darauffolgend wird mit dem gewonnenen Wissen in Kapitel 6 eine Methode entwickelt, die fähig ist, GraphQL automatisiert zu testen. Die entwickelte Methode wird mit einem Prototypen verifiziert, wobei die Entwicklung von diesem in Kapitel 7 betrachtet wird. Um die Fähigkeiten des Prototypen nachzuweisen, sind verschiedene Experimente in Kapitel 8 zu finden. Ein Ausblick für zukünftige Weiterarbeit findet sich in Kapitel 9. Abschließen wird die Arbeit in Kapitel 10 mit einem Fazit, wo noch einmal die ganze Arbeit kurz rekapituliert wird.

3 Grundlagen

Das automatisierte Testen von GraphQL-APIs erfordert ein spezifisches Domänenwissen in verschiedenen Teilbereichen der Informatik und Mathematik, insbesondere die Graphentheorie und das Softwaretesten. Dieses Domänenwissen wird in den folgenden Abschnitten auf Grundlage von zwei Lehrbüchern [5], [11] erarbeitet und in Kontext gesetzt. Die Graphentheorie wird benötigt, um die Struktur von GraphQL auf einer abstrakten Ebene besser verstehen zu können. Außerdem setzen die in [5] vorgestellten Überdeckungskriterien ein grundlegendes Wissen über Graphentheorie und Softwaretests voraus.

3.1 Graphentheorie

Da GraphQL es ermöglicht, dass komplexe Beziehungen innerhalb eines Datenmodells in Form von Graphen modelliert werden [12, vgl. Modelling with Graph(QL)] wird die Graphentheorie benötigt, da diese Methoden liefert, um Graphen zu definieren und zu analysieren. Des Weiteren sind die Testabdeckungskriterien, die später genutzt werden, eng mit der Graphentheorie verbunden. Die folgenden Absätze sind eher theoretisch gehalten. Die Zusammenhänge zwischen der Graphentheorie und dem Testen von GraphQL-APIs werden sich später erschließen.

3.1.1 Ungerichteter Graph

Ein Graph ist ein mathematisches Modell. Es kann dazu verwendet werden, Beziehungen zwischen Objekten darzustellen. Nach [11] ist ein ungerichteter Graph wie folgt definiert:

Definition 1 *Ein ungerichteter Graph ist ein Paar G = (V, E) zweier disjunkter Mengen mit* $E \subseteq V^2$.[11, vgl. S.2 0.1 Graphen]

Die Elemente der Menge V heißen Knoten (Vertices). Verbindungen zwischen den Knoten sind Elemente der Menge E und diese werden Kanten (Edges) genannt. In dieser Definition spielt die Ordnung der Elemente von E keine Rolle, daher werden solche Graphen auch ungerichtete Graphen genannt. Um Graphen darzustellen, gibt es verschiedene Ansätze. Der geläufigste Ansatz ist es, Knoten als Punkte und Kanten als Verbindungslinien zu zeichnen. Häufig wird auch eine Adjazenzmatrix genutzt, bei dieser wird mit $0, 1$ aufgeschlüsselt, welche Knoten eine Verbindung haben. Bei 0 existiert keine Kante zwischen den Knoten und bei 1 existiert eine.

Beispiel 1 *Ein Graph sei definiert mit* $V = \{1, 2, 3\}$ *und* $E = \{(1, 1), (1, 2), (1, 3)\}$
Mögliche Darstellungen des Graphen sind in Abbildung 3.1 und 3.2 gezeigt.

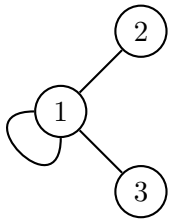

Abbildung 3.1: Gezeichneter Graph

	1	2	3
1	1	1	1
2	0	0	0
3	0	0	0

Abbildung 3.2: Adjazenzmatrix

3.1.2 Gerichteter Graph

Gerichtete Graphen sind die Grundlage vieler Überdeckungskriterien [5, vgl. 2.1 Overview]. Daher werden sie hier definiert.

Definition 2 *Ein gerichteter Graph ist ein Paar G = (V, E) zweier disjunkter Mengen mit zwei Funktionen init: E →V und ter: E →V, die jeder Kante e eine Anfangsecke init(e) und eine Endecke ter(e) zuordnen [11, S.26 0.10 Verwandte Begriffsbildungen].*

Bei einem gerichteten Graphen ist die Sortierung der Kantenpaare wichtig. Die Funktionen *init* und *ter* können am einfachsten durch die Sortierung der Elemente eines Kantenpaares umgesetzt werden. Hierbei ist das erste Element des Kantenpaares die Anfangsecke und das zweite Element ist die Endecke. Die Kanten in einem gerichteten Graphen werden mit einem Pfeil gezeichnet. Dabei zeigt der Pfeil stets in Richtung Endecke. Der in Beispiel 1 definierte ungerichtete Graph wird in Beispiel 2 als gerichteter Graph angepasst.

Beispiel 2 *Ein Graph sei definiert mit* $V = \{1, 2, 3\}$ *und* $E = \{e1, e2, e3\}$
Die Funktionen init und ter sind definiert als:
$init(e1) = 1$ *und* $ter(e1) = 1$
$init(e2) = 1$ *und* $ter(e2) = 2$
$init(e3) = 1$ *und* $ter(e3) = 3$

Die Sortierung der Kanten für *init* und *ter* ergibt für $e1 = (1, 1)$, $e2 = (1, 2)$ und $e3 = (1, 3)$.

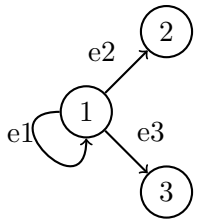

Abbildung 3.3: ein gerichteter Graph

3.1.3 Gewichteter Graph

Für die spätere Entwicklung des Testentwurfs ist es wichtig, gewichtete Graphen zu definieren. Zuvor wurde eine Kante in Definition 1 und Definition 2 als ein Tupel (x, y) eingeführt. Ein gewichteter Graph weist jeder Kante ein Kantengewicht zu. Dies ist im Allgemeinen eine positive, reelle Zahl [13, vgl. S. 251]. Das Kantengewicht kann sowohl ungerichteten als auch gerichteten Kanten zugewiesen werden.

Definition 3 *Ein gerichteter/ungerichteter Graph G = (V,E) mit einer Abbildung g : E →$\mathbb{R}_{>0}$ heißt gerichteter/ungerichteter gewichteter Graph. Die Abbildung g heißt Gewichtsfunktion. Für e ∈ E heißt g(e) das Gewicht von e. Das Gewicht von G ist die Summer der Gewichte aller Kanten, g(G) = $\sum_{e \in G} g(e)$.*
[13, *vgl. Definition* 6.1 *S.* 251]

Für spätere Anwendungszwecke muss die Definition jedoch ein wenig allgemeiner gefasst werden. Die Felder eines GraphQL-Typens sollen später als Gewicht genutzt werden um Tests zu entwerfen. Um dies zu erleichtern, soll die Definition 3 verallgemeinert werden durch eine Anpassung der Gewichtsfunktion. Die neue Definition soll vorerst *allgemein gewichteter Graph* genannt werden.

Definition 4 *Ein Graph G = (V,E) mit einer Abbildung g: E →X heißt allgemein gewichteter Graph. Die Menge X ist frei wählbar.*

An dieser Stelle sei verwiesen an Kapitel 6 wo diese Definition genutzt wird, um Felder eines GraphQL-Schemas den Kanten eines Graphens zuzuordnen.

3.1.4 Pfad

Ein Pfad, oft auch Weg genannt, ist eine Sequenz von Knoten, die nachfolgend durch Kanten miteinander verbunden sind [11, vgl. S. 7 0.3].

Definition 5 *Ein Weg ist ein nicht leerer Graph P = (V,E) der Form $V = \{x_0, x_1, ..., x_k\}$ und $E = \{x_0x_1, x_1x_2, \ldots, x_{k-1}x_k\}$ wobei die x_i paarweise verschieden sind [11, vgl. S. 7].*

Ein Weg wird oft durch die Folge seiner Knoten beschrieben also $P = x_0x_1 \ldots x_k$ [11, vgl. S.7] Die Länge eines Weges ist die Anzahl der Kanten die dieser besucht [11, vgl. S. 7]. In gewichteten Graphen ist das Gewicht eines Pfads die Summe aller Gewichte der einbezogenen Kanten [13, vgl. 7.2 kürzeste Wege].

Beispiel 3 *Es sei ein gerichteter Graph G definiert mit $V = \{n1, n2, n3, n4, n5, n6, n7, n8\}$ und $E = \{(n1, n2), (n2, n3), (n3, n4), (n4, n1), (n2, n5), (n3, n6), (n5, n7), (n6, n8)\}$. Die Sortierung der Elemente von E ist fest. Es gilt, dass init(x) das erste Element des Kantenpaares ist. ter(x) ist das zweite Element des Kantenpaares. Ein möglicher Pfad von Knoten n1 zu n8 ist der Pfad $p = \{(n1, n2), (n2, n3), (n3, n6), (n6, n8)\}$. Der Pfad p ist*

in Abbildung 3.4 rot markiert

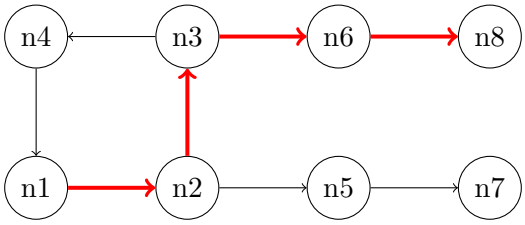

Abbildung 3.4: Pfad von n1 zu n8

3.1.5 Kreis

Ein Kreis in einem Graphen ist ein Weg, bei dem gilt: $Anfangsknoten = Endknoten$ [11, vgl. S. 8] Die Größe eines Kreises ist die Länge des Wegs, den dieser Kreis bildet. Der kürzeste Kreis eines Graphens nennt sich $Taillenweite\ g(G)$ und der längste Kreis ist der Umfang [11, vgl. S.8]. Der Graph aus Beispiel 3.4 hat einen Kreis der Länge 4 und ist in Abbildung 3.5 rot eingezeichnet.

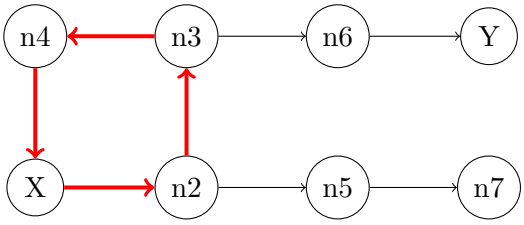

Abbildung 3.5: Ein zyklischer Graph

Im Kontext des Testentwurfs sind Kreise besonders interessant, da diese für einen potenziell unendlich großen Testraum sorgen. In einem azyklischen gerichteten Graphen, also einem gerichteten Graphen, der keinen Kreis besitzt, ist die Menge aller möglichen Pfade endlich. Bei einem Graphen mit Zyklen ist die Menge aller möglichen Pfade unendlich. Dies folgt aus der Tatsache, dass jeder Pfad, der den Kreis beinhaltet, diesen Kreis ein weiteres Mal ablaufen kann und somit stets ein neuer Pfad generiert wird.

3.2 APIs

Eine API ermöglicht es, dass unabhängige Anwendungen miteinander kommunizieren und Daten austauschen können [14]. Im Allgemeinen funktioniert diese Kommunikation mit dem HTTP-Protokoll über das Internet [4, vgl.]. Dabei stellt ein System per HTTP eine Anfrage und die API liefert eine Antwort. Es ist nicht festgelegt, auf welche Art das System die Antwort für die API erzeugt. Dies hängt stark von der zugrundeliegenden Implementierung ab.

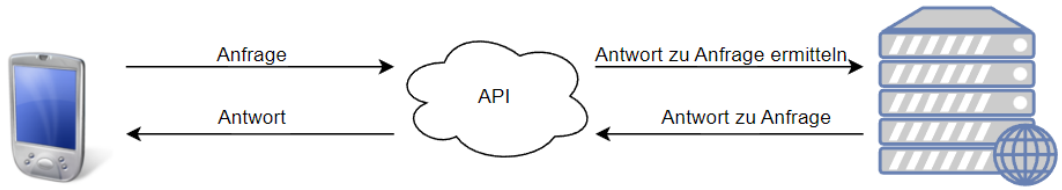

Abbildung 3.6: simple API-Kommunikation

Es gibt verschiedene Entwurfsmuster, die für ein API Design genutzt werden können. Die verschiedenen Muster werden durch Programmcode umgesetzt und sind somit fehleranfällig.

3.3 GraphQL

GraphQL [15] ist eine Open-Source Query-Language (Abfragesprache) und Laufzeitumgebung, die von Meta (ehemals Facebook) entwickelt wurde [15, vgl. Introduction]. Die Besonderheiten von GraphQL sind, dass mit nur einer einzelnen Anfrage mehrere Ressourcen gleichzeitig abgefragt werden können [16, vgl. No More Over- and Underfetching] und die Daten in einem Schema durch einen Typgraphen definiert sind [16, vgl. Benefits of a Schema & Type System]. So lässt sich die Effizienz stark erhöhen, indem weniger Anfragen gestellt werden, welche zeitgleich eine höhere Informationsdichte haben. GraphQL erleichtert außerdem die Kommunikation von Schnittstellen, indem die gewünschten Felder schon in der Query definiert sind und direkt den erwarteten Datentyp zusichern. Hier liegt auch der große Vorteil im Vergleich zum technologischen Konkurrenten REST API [4, vgl. Welche REST-Einschränkungen versucht GraphQL zu überwinden?]. Bei REST-APIs sind für verschiedene Ressourcen jeweils eigene Anfragen nötig [16, vgl. No More Over- and Underfetching] und die Typsicherheit ist nicht so stark gegeben wie bei GraphQL-APIs [4, vgl. Zusammenfassung der Unterschiede: REST vs. GraphQL]. Diese beiden großen Vorteile sorgen dafür, dass GraphQL an Popularität gewinnt und zunehmend eingesetzt wird [17, vgl. Continued growth and the road ahead].

GraphQL ist eine Abfragesprache und Spezifikation, dies bedeutet, dass GraphQL selbst keine konkrete Implementierung für eine Schnittstelle ist. Implementierungen der GraphQL-Spezifikation sind in GraphQL-Servern umgesetzt, die in verschiedenen Programmiersprachen existieren. Eine umfassende Auswahl verschiedenster Implementierungen findet sich in [18]. Besonderer Beliebtheit erfreuen sich ApolloServer [19], Express GraphQL [20] und HyGraph [21]. Da jedoch alle Server die GraphQL-Spezifikation umsetzen müssen und die hier entwickelten Tests aus GraphQL Anfragen bestehen, ist es irrelevant, welche konkrete GraphQL-Serverimplementierung das zu testende System verwendet. Im Kontext dieser Arbeit ist ein tiefgreifendes, technologisches Verständnis von GraphQL essenziell.

3.3.1 Schema & Typen

Grundlage jeder GraphQL-API ist ein GraphQL-Schema [15, vgl. Core Concepts]. Das Schema definiert exakt, wie die Daten in der API aufgebaut sind und welche Informationen existieren [15, vgl. 3.2 Schema]. Ein GraphQL-Schema ist eine Sammlung von einzigartigen Typen und definiert die Einstiegspunkte der API. Es gibt drei Einstiegspunkte: *query* zum Daten abfragen, *mutation* zum Daten verändern und *subscription* um über Datenänderungen informiert zu werden [15, vgl. 3.2.1 Root Operation Types]. *query* ist dabei als einziger verpflichtend, die anderen sind optional [15, vgl. 3.2.1]. Die drei Einstiegspunkte sind als Typen definiert. Ein Typ ist die fundamentale Einheit eines jeden GraphQL-Schemas [15, vgl. 3.4 Types].

Es gibt 6 verschiedene Typdefinitionen, diese sind:

Typ	Beschreibung
ScalarTypeDefinition	Primitive Datentypen, welche keine Verbindungen zu anderen Typen haben dürfen (Strings, Integer, ...).
ObjectTypeDefinition	Komplexere Datentypen, die Verbindungen untereinander haben und in Feldern ihre Verbindungen definieren.
InterfaceTypeDefinition	Ein abstrakter Datentyp, der die Struktur für andere Typen vorgibt.
UnionTypeDefinition	Ein Typ, der die Vereinigung verschiedener Typen ist.
EnumTypeDefinition	Ein Typ, der nur eine feste Anzahl an vorher festgelegten Werten hat.
InputObjectTypeDefinition	Zusammensetzung von Scalar-Types, um komplexere Eingabeargumente zu bilden.

Tabelle 3.1: GraphQL Typen [15, vgl. 3.4 Types]

Ein Typ hat einen einzigartigen Namen und definiert alle Informationen über sich, hierbei wird für jede Information ein Feld angelegt. Das Feld setzt jeweils einen Typen um (InputObjectTypeDefinition ist dabei ausgeschlossen). Ein sehr einfaches Schema wäre die Beziehung zwischen Büchern und Autoren. Ein Buch hat einen Titel und einen Autor. Der Autor hat einen Namen und ein Geburtsdatum. Ein zugehöriges Schema für dieses Beispiel ist in Abbildung 3.7 dargestellt.

```
1  type Buch {
2      title: String
3      author: Autor
4  }
5  type Autor{
6      name: String
7      geburtsdatum: Date
8  }
```

Abbildung 3.7: Minimales Schema mit zwei Types

Es lässt sich also festhalten, dass ein GraphQL-Typ immer als ein Tupel `(Name, Felder)`

definiert wird, wobei die Felder eine Liste an Tupeln `(Feldname, Feldtyp, Datentyp)` sind [15, vgl. 3.6 Objects]. Hierbei gelten Einschränkungen für die Elemente des Tupels.

Feldname ist ein eindeutiger Feldbezeichner

Feldtyp gibt Einschränkungen vor, zum Beispiel nicht Null (durch `!`), Listentyp (durch `[]`) etc.

Datentyp ist der explizite Typ den das Feld hat, kann Standarddatentyp oder anders definierter Type sein

3.3.2 Query

Der Query-Type ist der Einstiegspunkt in alle Abfragen. Da somit auch jeder Test im Query-Type beginnt, soll dieser hier näher betrachtet werden. Abfragen können mit und ohne Eingabeparameter angegeben werden. Informationen darüber finden sich im Schema. Die definierten Anfragen haben, wie jeder Typ, einen eindeutigen Bezeichner, welcher dann in der zustellenden Abfrage benutzt wird. Wenn das abgefragte Feld vom Typ $SCALAR$ ist, dann wird es direkt ausgegeben. Ist das Feld vom Typ $OBJECT$, dann muss definiert werden, welche Felder des $OBJECTS$ gewünscht sind. Eine korrekte Anfrage muss stets mindestens ein Feld des Typens $SCALAR$ enthalten. Der Query-Type für das Schema aus Abbildung 3.7 kann umgesetzt werden durch Abbildung 3.8.

```
1         type Query{
2             # liste aller b cher
3             getBooks: [Book]
4             # ein zuf lliges buch
5             getBook: Book
6             # author zum jeweiligen Buch
7             getBookByTitle(String title): Author
8         }
```

Abbildung 3.8: Query Type für Buch und Autor

Eine solche API ist in der Lage, drei verschiedene Anfragen zu beantworten. Es muss beachtet werden, dass die Anfragen alle zwingend einen $SCALAR$ Type aufweisen müssen. Somit ist die Anfrage `getBookByTitle("Beispieltitel"){ author }` nicht gültig und muss erweitert werden zu `getBookByTitle("Beispieltitel"){ author{ name } }`.

3.3.3 Resolver

Bisher wurde die Strukturierung und Typisierung von GraphQL und den zugrundeliegenden Daten dargestellt. Es ist noch unklar, wie GraphQL die zugrundeliegenden Daten abfragt und woher diese kommen. Dies wird durch Resolver umgesetzt.

Ein Resolver ist in GraphQL eine Funktion, die zuständig für die Datenabfragen und Strukturierung ist [19, vgl.]. Im Schema wird definiert, in welcher Form die Daten sein sollen. Resolver sorgen dafür, dass die richtigen Daten in der richtigen Form zur Verfügung gestellt werden. Die Resolver sind nicht, wie alle vorher benannten Teile von GraphQL offen einsehbar, sondern sind Funktionen einer Programmiersprache. Sie setzen die Schnittstellenprogrammierung wie in Kapitel 3.2 angesprochen um. Für jeden Typ im Schema muss ein Resolver implementiert werden. Das umfasst insbesondere die Query, Mutation und Subscription-Typen, aber auch alle selbstdefinierten Typen [19, vgl.]. Die konkrete Implementierung der Resolver hängt von verschiedenen Dingen ab, insbesondere jedoch davon, welcher GraphQL-Server genutzt wird. Ein Resolver für die Query eines Buches anhand seines Titels mit ApolloServer in Javascript kann die Syntax aus Abbildung 3.9 haben.

```
const resolvers = {
    Query: {
        book: (parent, args, context, info) => {
            return getBookByID(args.id);
        },
    },
};
```

Abbildung 3.9: Ein einfacher Resolver

Es ist zu beachten, dass alle Argumente, die mitgegeben werden, im `args` Argument gespeichert sind. Die Funktion $getBookByID$ gibt ein dem Schema entsprechendes Json-Objekt zurück. Da die Resolver konkrete Implementierungen außerhalb von GraphQL sind und die einzelnen Resolver untereinander aufrufen können, bedarf es hier einer Reihe an Tests, damit die erwartete Funktionsweise des Programmcodes nachgewiesen werden kann. Ein häufiger Fehler in GraphQL ist es, einzelne Attribute in einem Resolver zu vergessen. Ein Entwickler sollte für jeden Resolver, der eine Funktion darstellt, einen beziehungsweise mehrere Tests zur Verfügung stellen. Da sich die einzelnen Resolver aber auch untereinander aufrufen können, kann sich ein teils riesiger Testraum ergeben. Hierzu mehr in Kapitel 3.4.2. Um sicherzustellen, dass der Kombinationsaufruf der Resolver fehlerfrei ist, wird im Folgenden eine Methode entwickelt, welche gewährleistet, dass die möglichen Kombinationen ausreichend durch Tests abgedeckt sind und so die Qualität und Zuverlässigkeit von GraphQL-APIs erhöht wird.

3.4 Zusammenhang Graphentheorie und GraphQL

Da ein grundlegendes Wissen über Graphentheorie und GraphQL geschaffen wurde, muss noch gezeigt werden, dass Graphentheorie auch anwendbar ist auf GraphQL. Die Verknüpfung wird später genutzt, um Algorithmen, die für Graphen gedacht sind, für GraphQL zu nutzen.

3.4.1 Schema als Graph

Das GraphQL-Schema ist, wie im Kapitel 3.3.1 gezeigt, eine Komposition von Typen. Ein Typ definiert Felder in sich. Jedes Feld eines Typens zeigt seinerseits wieder auf ein anderes Feld [12, vgl. Modelling with GraphQL]. Somit wird jedes Feld eines Typens zu einer ausgehenden Kante. Es sei ein einfacher Typ *Buch* mit zwei Feldern vom Typ *SCALAR* in Abbildung 3.10 definiert. Der zugehörige Graph ist dargestellt in Abbildung 3.11.

```
type Buch {
    id: Int
    title: String
}
```

Abbildung 3.10: Buch-Typ

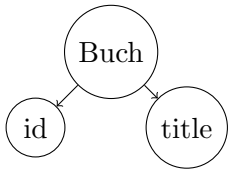

Abbildung 3.11: Graph des Typ-Buch

Der Typ *Buch* besitzt zwei ausgehende Kanten, zu den jeweils beiden definierten Feldern *id* und *title*. In Kapitel 3.3.1 wurde festgestellt, dass Skalare Typen keine eigenen Beziehungen haben. Daher können die Felder *id* und *title* selbst keine ausgehenden Kanten haben. Das generelle Prinzip der Graphbildung ist:

1. Erstelle Knoten aus jedem Typen und den definierten Feldern des Knotens.
2. Ziehe für jeden Typ seine Kanten, indem alle Felder des Typens zum entsprechenden Knoten verbunden werden.

Nach diesem Prinzip kann aus einem beliebig großen Schema ein gerichteter Graph gebildet werden.

3.4.2 Abfragen im Graphen

Im Folgenden soll geklärt werden, wie GraphQL Anfragen intern behandelt und dies abbildbar in einem Graphen ist. Hierfür wird in Abbildung 3.12 ein Schema definiert.

```
type Query {
    buch: Buch
    autor: Autor
}
type Buch {
    id: Int
    title: String
    autor: Autor
    verleger: Verlag
}
```

```
type Author {
    id: Int
    name: String
    schrieb: [Buch!]
}
type Verlag {
    name: String
}
```

Abbildung 3.12: Schemadefinition

Das in Abbildung 3.12 definierte Schema wird durch den Graphen in Abbildung 3.13 visualisiert.

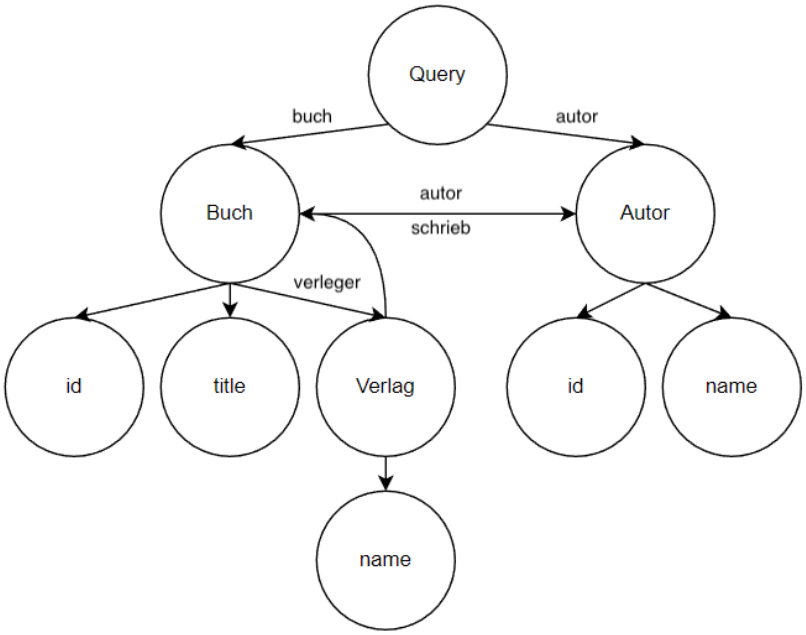

Abbildung 3.13: Graph für Schemadefinition aus Abbildung 3.12

Valide Anfragen an eine GraphQL-API sind alle Pfade, die mit Startknoten *Query* beginnen und mindestens einen Knoten beinhalten, der keine ausgehenden Kanten hat (also ein Scalar-Type) [12, vgl. Modelling with GraphQL]. Auf jeder Ebene des Graphens werden

Resolver ausgeführt, welche für die Datenbereitstellung verantwortlich sind. Es ist wichtig zu betonen, dass der aufgespannte Graph ein gerichteter Graph ist. Hierdurch wird ermöglicht, dass die *SCALAR* Felder stets Endknoten sind da sie selbst keine ausgehenden Kanten besitzen dürfen. Soll von einem GraphQL-Server mit dem zuvor definierten Schema ein Buch mit id, Titel und Verlag mit Verlagsnamen abgefragt werden, so kann diese Query genutzt werden: `{ buch { id title verleger { name }}}`.

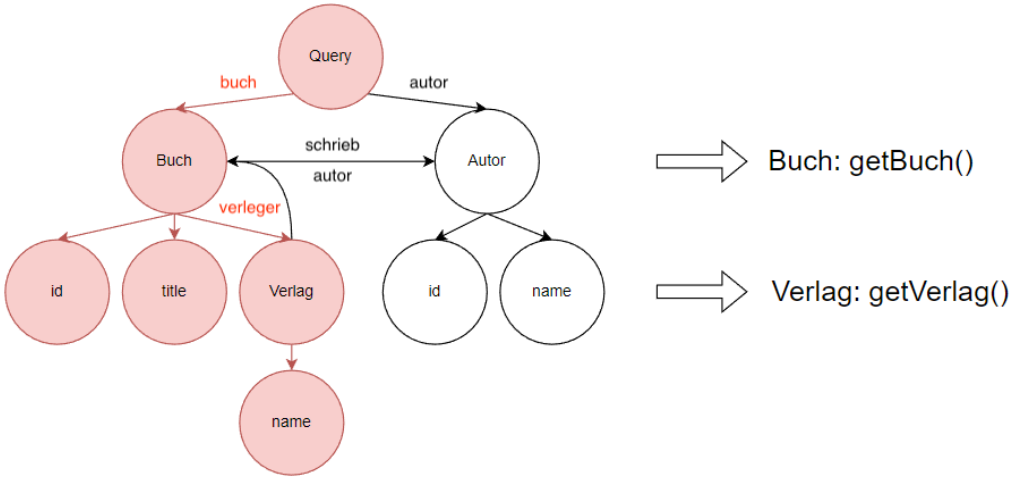

Abbildung 3.14: Graph für Abfrage nach [12]

Mit Ausführung dieser Query wird zuerst der Resolver $getBuch()$ ausgeführt. Liefert dieser ein valides Ergebnis zurück, also gibt es ein solches Buch, dann wird der nächste Resolver $getVerlag()$ ausgeführt [12, vgl. Resolver]. Dadurch, dass die Typen *Buch* und *Autor* aufeinander verweisen, ist der Graph des Schemas zyklisch und die Pfadmenge, das heißt die erlaubten Abfragen, die gestellt werden können, unendlich.

3.5 Testen

Indem technische Geräte und somit auch Software im umfangreichen Maßstab Einzug nehmen in nahezu alle Bereiche des Lebens, ist es wichtig, die Sicherheit, Qualität und Zuverlässigkeit von Software sicherzustellen [5, vgl. Introduction]. Um dies sicherzustellen, sind systematische Tests von Software nötig. Dabei ist das Ziel, dass die Software bestimmten Anforderungen und Spezifikationen entspricht. Hierbei werden diverse Techniken und Ansätze verfolgt, die im Folgenden kurz vorgestellt werden.

3.5.1 Sichtweisen auf Testsysteme

Es gibt verschiedene Sichtweisen auf das zu testende System. Die Sichtweisen können intern oder extern bestimmt sein. So sind Faktoren wie möglicher Zugriff auf Quellcode oder Architekturdetails des Systems essenziell wichtig, um zu entscheiden, welche Art von Tests angewandt werden sollen. Es gibt zwei generelle Sichtweisen und eine Mischform. Das zu testende System wird als Box betrachtet. Diese Box kann aus verschiedenen Sichten gesehen werden, die im Folgenden näher erläutert werden. Die Ansätze unterscheiden sich vor allem in den zur Verfügung stehenden Informationen über das System.

White-Box Testing

Im White-Box Testing stehen alle Informationen über das System zur Verfügung [22, vgl. 1.4.2 Code-Based Testing]. Der Tester hat Zugriff auf Code, Architekturdetails und besitzt Kenntnisse über alle möglichen Details des Systems [22, vgl. 1.4.2 Code-Based Testing]. Somit kann der Tester auf alle möglichen Informationen über das System zugreifen und damit seine Tests generieren. Die erstellten Tests fundieren dann auf einem Niveau, welches durch das Domänenwissen über das System begründet wird. Verschiedene Techniken zur Analyse des Domänenwissens wurden entwickelt, um die Informationen für die Testentwicklung zu nutzen. In Kapitel 3.6 wird eine dieser Techniken näher untersucht.

Black-Box Testing

Im Black-Box Testing hat der Tester keinen Zugriff auf interne Funktionsweisen der Software. Schwerpunkt des Testens ist es, dass die Software das tut, was in den Anforderungen verlangt wird [22, vgl. Specification-Based Testing]. Da der Quellcode nicht einsehbar ist, muss darauf verlass sein, dass die Anforderungen treffend formuliert wurden [23, vgl.]. Das Black-Box Testing hat einen methodischen Bezug zu Property-based Testing [6].

Grey-Box Testing

Das Grey-Box Testing ist eine Mischform von White-Box und Black-Box Tests [23, vgl.]. Es sind in dieser Sicht Teile der Software bekannt, aber es gibt keinen umfassenden Einblick wie im White-Box Testing. Es werden sowohl funktionale als auch strukturelle Testansätze verfolgt, je nachdem, wie viele Informationen über das System tatsächlich

verfügbar sind [24, vgl.]. Das System wird aus der Sicht des Endbenutzers getestet [23, vgl.], jedoch mit zusätzlichem Wissen über Teile des internen Aufbaus.

3.5.2 Arten von Tests

Neben verschiedenen Transparenzen auf dem zu testenden System gibt es verschiedene Granularitätsebenen der Tests. Die Tests können abgeleitet werden von Anforderungen, Spezifikationen, Designartifikaten und dem Programmcode [5, vgl. 1.1.1 Testing Levels Based on Software Activity]. Dabei können verschiedene Level an Tests definiert werden. Die Level sind eng mit den Entwicklungsaktivitäten einer Software verbunden [5, vgl. 1.1.1].

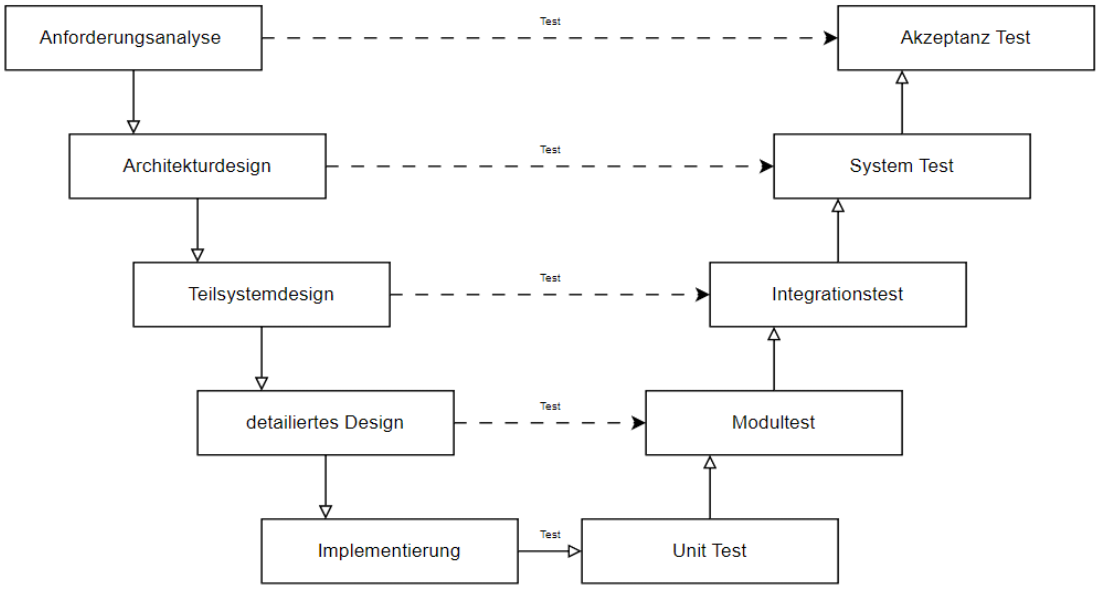

Abbildung 3.15: Softwareentwicklung und Test-Levels im V-Modell [5, vgl. Figure 1.2]

Die verschiedenen Testebenen sollen schon im Designprozess beachtet werden, denn die Formulierung von Tests kann dabei helfen, Designfehler zu finden, noch bevor die Software entwickelt wird [25, vgl. 1.1.1]. Die verschiedenen Ebenen der Tests spiegeln auch verschiedene Systemsichten in den Tests. In Abbildung 3.15 sind im V-Modell die verschiedenen Schritte der Softwareentwicklung abgebildet. Dazu passend findet sich im Folgenden die Erklärung der einzelnen Ebenen.

Akeptanz-Test - Betrachten der Software hinsichtlich der Anforderungen

System-Test - Betrachten der Software hinsichtlich der Architektur

Integrations-Test - Betrachten der Software hinsichtlich der Teilsysteme

Modul-Test - Betrachten der Software hinsichtlich detailliertem Design

Unit-Test - Betrachten der Software hinsichtlich korrekter Implementierung

[5, vgl. 1.1.1]

Im Folgenden wird jede einzelne Testebene präziser betrachtet. Die Reihenfolge der Betrachtung ist dabei von feingranular zu grobgranular.

Unit-Tests

Als feingranularste Testebene ist das Ziel des Unit-Tests, den entwickelten Code zu testen. Eine einzelne *Unit* ist in Objekt-Orientierter Programmierung eine Funktion oder Methode [22, vgl. Unit Testing]. Der einzelne Unit-Test konzentriert sich dabei auf eine Funktion oder Methode.

```
1         def add(a, b):
2             return a + b
```

Abbildung 3.16: Eine einfache Python-Funktion

```
1     def test_add_positive():
2         self.assertEqual(add(3, 5), 8)
3
4     def test_add_negative():
5         self.assertEqual(add(-3, -5), -8)
6
7     def test_add_mixed():
8         self.assertEqual(add(5, -3), 2)
```

Abbildung 3.17: Drei Unit-Tests für die add-Funktion

Er prüft, ob die gegebene Einheit bei bekannten Eingaben die erwartete Ausgabe liefert. Für diese Tests werden häufig Testframeworks genutzt, die bei der Entwicklung und Ausführung der Tests helfen [5]. In Abbildung 3.16 wurde eine Funktion definiert und in Abbildung 3.17 sind drei Unit-Tests mit PyTest [26] definiert. Die Tests führen verschiedene Methoden aus und prüfen, ob das Ergebnis mit der Erwartung übereinstimmt. Das *assertEqual* übernimmt dabei die Auswertung.

Modul-Test

Eine Granularitätsebene höher ist der Modul-Test. Ein Modul ist eine Sammlung von Units [5, vgl. S. 6]. Ziel ist es, die Interoperabilität der einzelnen Units in einem Modul sicherzustellen [5, vgl. S. 6].

```
1         def add(a, b):
2             return a + b
3         def sub(a, b):
4             return a - b
5         def mul(a, b):
6             return a * b
7         def quo(a, b):
8             return a / b
```

Abbildung 3.18: Ein Python Rechenmodul

In Abbildung 3.18 ist ein Rechenmodul definiert. Dieses wird getestet, indem verschiedene Funktionen miteinander kombiniert werden und dann geprüft wird, ob Erwartung und Ergebnis übereinstimmen. Ein Modul-Test für das Modul aus Abbildung 3.18 ist in Abbildung 3.19 definiert.

```
1     def test_rechenmodul():
2         testResult = (mul(add(2,3), sub(3,2)))
3         self.assertEqual(testResult, 5)
```

Abbildung 3.19: Ein Modul-Test

Es ist anzumerken, dass die Interoperabilität im Modul-Test nur innerhalb eines Moduls getestet wird [5, vgl. S. 6].

Integrations-Test

Die Integrations-Tests übernehmen das Testen von Interoperabilität zwischen verschiedenen Modulen [5, vgl. S. 7]. Es wird davon ausgegangen, dass die einzelnen Module zuvor korrekt getestet worden sind und die einzelnen Module korrekt arbeiten [5, vgl. S. 7]. Testobjekte sind die Schnittstellen der einzelnen Module und somit auch die Kommunikation zwischen den Modulen.
In Abbildung 3.20 ist ein Modul definiert, das einen neuen Nutzer erstellen kann. Abbildung 3.21 definiert ein Modul, das Nutzer speichern und finden kann. Ein Integrations-Test beider Module sollte testen, ob ein neu angelegter Nutzer ordentlich gespeichert wird und ob die dabei zugewiesenen Daten auch korrekt bleiben. In Abbildung 3.22 ist ein solcher Integrations-Test zu finden.

```
def newUser(name, gb, email):
    pw = generateRandomPw()
    return new User(name, gb, email, pw)
```

Abbildung 3.20: Modul 1

```
def saveUser(User user):
    db.save(user)
def getUser(name):
    db.findUser(name)
```

Abbildung 3.21: Modul 2

```
def testAddNewUserAndSaveUserAndGetUser():
        user = newUser("Peter", "01.04.1980", "p@test.de")
        saveUser(user)
        self.assertEqual(getUser("Peter"), user)
```

Abbildung 3.22: Integrations-Test zwischen Modul 1 und Modul 2

Im Kontext dieser Arbeit gilt es zu untersuchen, wie Integrations-Tests automatisiert für GraphQL erstellt werden können. Hinter jedem verschiedenen Typen von GraphQL steckt ein Resolver, welcher als ein eigenes Modul gesehen werden kann. Die Interoperabilität dieser Module gilt es im Folgenden automatisiert zu testen.

System-Test

Um zu testen, ob nicht nur einzelne Teile des Systems gut miteinander funktionieren, sondern auch das ganze System als Solches, werden die System-Tests genutzt [5, vgl. S. 6]. Die Tests werden auf Grundlage der Spezifikation des Systems erstellt. Es wird davon ausgegangen, dass die einzelnen Module hier wie erwartet funktionieren. Im Vordergrund dieser Testebene steht, dass die Software die Spezifikation, also die Erwartungen an sich, erfüllt.

Akzeptanz-Test

Die finale Ebene des Testprozesses ist der Akzeptanz-Test. Auf dieser Ebene wird die Software aus Sicht des Endnutzers geprüft. Dieser ist oft auch Teil dieses Prozesses [5, vgl. S.6]. Ziel ist es, zu verifizieren, dass die Analyse und Umsetzung des Problems erfolgreich ist und der Nutzer mit der entwickelten Lösung zufrieden ist [5, vgl. S.6].

3.5.3 Testabdeckung

Bisher ungeklärt ist, wann ausreichend getestet wurde und ob überhaupt genügend getestet werden kann. Hierzu werden die formalen Abdeckungskriterien eingeführt, wie sie in [5] definiert sind. Mithilfe dieser Abdeckungskriterien wird es möglich, sinnvolle Testfälle zu entwickeln und zu entscheiden, wann ausreichend getestet wurde. Die Notwendigkeiten für solche Abdeckungskriterien zeigt sich schnell. Ein Ausprobieren aller Kombinationen ist in der heutigen Zeit unmöglich. Als Beispiel sei hier eine simple Addition von 2 64-bit Integer genannt. Für eine komplette Testung dieser Addition gibt es 2^{64} Kombinationen. Mit einem 3GHz-Prozessor wäre eine vollständige Testung nach etwa 69 Tagen erledigt. Betrachtet sei ein Java-Compiler, der Eingaberaum von Programmen, die zum Test stehen, ist effektiv unendlich und somit nicht testbar [5, vgl. 1.3 Coverage Criteria for Testing].

Abdeckungskriterien

Da ein vollständiges Testen, also ein Ausprobieren aller Möglichkeiten unmöglich ist, müssen Kriterien geschaffen werden, die eine hinreichende Testqualität zusichern. Hierfür werden die Abdeckungskriterien eingeführt. Diese liefern einen Ansatz, der dabei hilft, sinnvolle Tests zu entwickeln und abzuschätzen, wann ausreichend getestet wurde. Im Folgenden werden die Abdeckungskriterien auch als Testanforderungen [5, vgl. 1.3 Coverage Criteria for Testing] gesehen.

Definition 6 *Eine Testanforderung ist ein spezifisches Element eines Softwareartifkates das einen Testfall erfüllen muss [5, Def. 1.20].*

Dabei ist ein Abdeckungskriterium erfüllt, wenn alle seine Testanforderungen erfüllt sind.

Definition 7 *Ein Abdeckungskriterium ist eine Regel oder eine Sammlung von Regeln, die Testanforderungen an eine Menge von Testfällen stellen [5, Def. 1.21].*

Um eine Aussage darüber zu machen, wie gut eine Menge an Tests das Abdeckungskriterium umsetzt, wird die Abdeckung eingeführt. Einerseits ist es manchmal sehr schwierig ein Abdeckungskriterium vollständig zu erreichen, andererseits kann dann gemessen werden, ob das Kriterium erfüllt wurde [5, vgl. S. 18]. Durch die Definition 8 wird messbar, wie gut eine Menge an Tests das Abdeckungskriterium umsetzt und ob das Kriterium erfüllt ist.

Definition 8 *Gegeben sei eine Menge an Testanforderungen TR für ein Abdeckungskriterium C. Eine Menge an Testfällen T erfüllt C wenn gilt, dass für jede Testanforderung mindestens ein Test existiert der diese Testanforderung erfüllt [5, vgl. Def. 1.22].*

Es gibt diverse Abdeckungskriterien, die auf verschiedenen Annahmen beruhen. Ist das Ziel, dass alle Entscheidungen in einem Programm abgedeckt sein sollen (Branch-Coverage), so führt jede Entscheidung zu zwei Testanforderungen, eine Anforderung

für den positiven und eine für den negativen Entscheidungsfall [5, vgl. S. 17]. Soll jede Methode mindestens einmal aufgerufen werden (Call-Coverage), so führt jede Methode zu einer Testanforderung, um diese Methode abzudecken [5, vgl. S. 17]. Im folgenden Kapitel 3.6 werden Abdeckungskriterien speziell für Graphen eingeführt.

3.6 Graphabdeckung

Die Graphabdeckung führt verschiedene Kriterien ein, die es ermöglichen, aus Graphen Pfade für Tests zu generieren. GraphQL kann, wie im Kapitel 3.4 gesehen, durch einen Graphen repräsentiert werden. Hinter jedem Knoten des Graphen steckt ein Resolver, der potenziell einem eigenen Modul angehört. Anfragen an GraphQL können Resolver beliebig miteinander kombinieren. Um die möglichen Kombinationen zu testen, sollen Graphabdeckungskriterien genutzt werden. Da in GraphQL zyklische Graphen erlaubt sind, ist der Testraum potentiell unendlich groß. Dieses Problem kann durch die Verwendung von Graphabdeckung gelöst werden. Zuerst wird aber weitere Theorie eingeführt. Um Graphcoverage zu nutzen, wird die allgemeine Definition 2 von gerichteten Graphen erweitert. Im Testkontext definiert sich ein gerichteter Graph wie in Definition 9 angegeben.

Definition 9 *Ein gerichteter Graph G ist definiert als*

Menge N *von Knoten*

Menge N_0 *von Anfangsknoten, wobei $N_0 \subseteq N$*

Menge N_f *von Endknoten, wobei $N_f \subseteq N$*

Menge E *von Kanten, wobei $E \subseteq N \times N$. Dabei ist die Menge als init(x) $\times$ target(y) definiert.*

[5, 2.1 Overview]

Das *target* aus Definition 9 ist gleichzusetzen mit *ter* aus Definition 2. Mithilfe dieser Definition können zum Beispiel Kontrollflussgraphen abgebildet werden, indem die Einstiegspunkte die Anfangsknoten sind und die Endknoten die Austrittspunkte [5, vgl. S. 46]. Ein Pfad innerhalb von eben definierten Graphen der in einem Knoten $x \in N_0$ startet und in einem Knoten $y \in N_f$ endet, nennt sich Testpfad [5, vgl. Def 2.31]. Ziel ist es mithilfe von Abdeckungskriterien Testpfade zu ermitteln die einen Graphen ausreichend abdecken. Ein Graph gilt als ausreichend abgedeckt wenn die Definition 10 gilt.

Definition 10 *Gegeben sei eine Menge TR von Testanforderungen für ein Graphabdeckungskriterium C. Eine Menge Tests T erfüllt C auf Graphen G wenn gilt: Jedes Element von TR ist durch mindestens einen Pfad p abgedeckt. Ein Pfad p deckt ein Element von TR ab, wenn das Element von TR ein Teilpfad von p ist. Dabei kann auch gelten, dass das Element von TR = p ist. [5, vgl. Def. 2.32]*

Hierfür werden in [5] verschiedene Kriterien eingeführt. Diese sollen im Folgenden definiert und erklärt werden.

3.6.1 Graphabdeckungskriterien

Ein Graphabdeckungskriterium ist eine Sammlung von Testanforderungen gemäß Definition 6, wobei die Anforderungen an Graphen gestellt werden. Im Folgenden werden einige Graphabdeckungskriterien vorgestellt, so wie sie in [5] definiert werden.

Knotenabdeckung

Wird erwartet, dass beim Testen jede definierte Methode zumindest einmal ausgeführt wird, so handelt es sich hierbei um Knotenabdeckung. Dieses Kriterium ist weithin geläufig als *Blockabdeckung* [5, vgl. 2.2.1]. Eine Menge T an Tests erfüllt die Knotenabdeckung, wenn gilt, dass jeder erreichbare Knoten durch zumindest einen Test $t \in T$ besucht wird. Formal definiert wird dies in Definition 11:

Definition 11 ***Knotenabdeckung****: TR enthält jeden erreichbaren Knoten in G [5, vgl. Criterion 2.1].*

Kantenabdeckung

Eine Granularitätsebene höher ist die Kantenabdeckung. In diesem Kriterium wird gefordert, dass jede erreichbare Kante mindestens einmal in einer gegebenen Menge an Tests besucht wird. Dabei wird die Pfadlänge aus Kapitel 3.1.4 benötigt.

Definition 12 ***Kantenabdeckung****: TR enthält jeden erreichbaren Pfad der Länge bis zu 1 (Kanten), in G [5, vgl. Criterion 2.2].*

Dadurch ist eingeschlossen, dass auch jeder Knoten besucht wird. Es folgt daher, dass die Kantenabdeckung ein stärkeres Kriterium ist, da dieses die Knotenabdeckung automatisch beinhaltet. Diese Gegebenheit wird sich weiterhin fortführen, sodass die Kriterien im Generellen stärker, aber auch schwerer zu berechnen werden.

Kanten-Paar Abdeckung

Die Kantenabdeckung betrachtet nur die einzelnen Pfade des Graphens. Im Testkontext ist aber durchaus die zuvor ausgeführte Operation auch wichtig und muss im Testprozess berücksichtigt werden. Um dem Rechnung zu tragen, wird die Kanten-Paar Abdeckung eingeführt. Diese setzt voraus, dass eine Menge an Tests T alle möglichen Kantenpaare durch mindestens einen Test abgedeckt hat.

Definition 13 ***Kanten-Paar Abdeckung****: TR enthält jeden erreichbaren Pfad der Länge bis zu 2 (Kanten), in G [5, vgl. Criterion 2.3].*

PrimePfad Abdeckung

Während die zuvor definierten Kriterien darauf achten, dass Knoten und Kanten(-paare) abgedeckt werden, muss auch in Beachtung bezogen werden, dass im Testkontext durchaus alle Pfadkombinationen die existieren relevante Testfälle darstellen können. Die Anzahl an allen azyklischen Pfadkombinationen wird jedoch selbst bei kleinen Programmen schnell groß[5, vgl. S. 35]. Gleichzeitig sind viele Pfadkombinationen Teile von längeren Pfaden und somit uninteressant im Testkontext [5, vgl. S. 35]. Um dieses Problem zu lösen, wird die PrimePfad Abdeckung in Definition 14 eingeführt.

Definition 14 ***PrimePfad Abdeckung****: TR enthält alle PrimePfade in G [5, vgl. Criterion 2.4].*

Ein PrimePfad ist definiert in Definition 15.

Definition 15 *Ein Pfad von n_l zu n_i ist ein PrimePfad, wenn gilt, dass dieser keinen Knoten doppelt enthält (mit Ausnahme von Start- und Endknoten) und der Pfad nicht Teilpfad eines anderen Pfades ist.*

Es sollen also die längsten, einfachen Pfade im Graphen abgedeckt werden. Ein einfacher Pfad ist ein Pfad, der keinen Knoten doppelt enthält. Ausnahme ist hierbei, dass Start und Endknoten gleich sein dürfen. So wird eine umfassendere Abdeckung als in den vorherigen Abdeckungskriterien gewährleistet, da auch längere Pfadkombinationen dabei berücksichtigt werden.

Vollständige Pfadabdeckung

Idealerweise sollten Tests die gesamte Software abdecken. Wie jedoch in Kapitel 3.5.3 schon gezeigt ist dies oft nicht möglich. Insbesondere wenn der Graph zyklisch ist, ist der Pfadraum unendlich und kann somit nicht vollständig abgedeckt werden [5, vgl. S. 36]. Der Vollständigkeit halber soll dieses Kriterium hier dennoch definiert werden.

Definition 16 ***Vollständige Pfadabdeckung****: TR enthält alle Pfade in G [5, vgl. Criterion 2.7].*

Sollte der zu testende Graph azyklisch sein, kann dieses Kriterium jedoch durchaus genutzt werden.

3.6.2 Vergleich der Kriterien

Die verschiedenen Abdeckungskriterien variieren in ihrer Granularität. Ein kurzer Überblick soll hier gegeben werden.

Kriterium	Granularität
Knoten	jeder Knoten in mindestens einem Testpfad
Kanten	jede Kante in mindestens einem Testpfad
Kanten-Paar	jedes Kantenpaar in mindestens einem Testpfad
PrimePfad	jeder PrimePfad in einem Testpfad
vollständige Pfade	Alle möglichen Pfade

Tabelle 3.2: Vergleich der Graphabdeckungskriterien

Die vorgestellten Abdeckungskriterien können hierarchisch sortiert werden. Mit Erfüllung eines höher sortierten Kriteriums sind automatisch die tieferen Kriterien auch erfüllt [5, vgl. Figure 2.15]. Die Hierarchisierung der Abdeckungskriterien ist durch Abbildung 3.23 dargestellt.

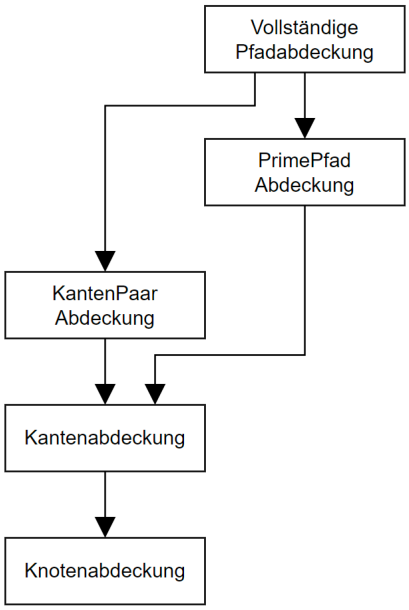

Abbildung 3.23: Subsumtion der Abdeckungskriterien

Hierdurch gilt, dass zum Beispiel mit Kantenabdeckung automatisch auch Knotenabdeckung gewährleistet ist. Es ist nicht gewährleistet, dass PrimePfad-Abdeckung auch KantenPaar-Abdeckung umsetzt. Im Kontext des Integrationstestens sei jedoch gefolgert, dass die PrimePfad-Abdeckung ein stärkeres Abdeckungskriterium ist. Dies folgt, weil die PrimePfad-Abdeckung längere Testpfade erzeugt was nach [6, vgl. RQ 1] für eine höhere Abdeckung sorgt. Es sei außerdem angemerkt, dass die PrimePfad-Abdeckung fast, aber nicht vollständig die KantenPaar-Abdeckung umsetzt [27, vgl. S. 20]. Ein

Auftreten von Eigenkanten, also Kanten von einem Knoten zu sich selbst, zeigt einen Unterschied zwischen PrimePfad und Kantenpaar-Abdeckung [27, vgl. S. 21]. Da im Sinne des Integrationstests jedoch davon ausgegangen wird, dass die einzelnen Module getestet sind, ist diese Eigenkante kein relevanter Integrationstest da das Modul nur mit sich selbst arbeitet und somit keine Kombination von Modulen durch solch eine Kante getestet wird.

4 Graphabdeckung für GraphQL

Ziel dieses Kapitels ist es, einen theoretischen Zugang zu schaffen, um die Abdeckungskriterien für Graphen aus Kapitel 3.6 nutzbar zu machen für Testgenerierung. Grundlage des Zugangs ist die Arbeit aus Kapitel 3.4. Abbildung 3.14 stellte den initialen Zugang dar. Hierbei waren Knoten, die ausgehende Kanten haben, als Resolver zugeordnet. Ein Resolver ist wie in Kapitel 3.3.3 festgestellt eine Funktion eines Moduls, das es zu testen gilt. Durch die Graphstruktur eines GraphQL-Schemas sollen Pfade generiert werden, die sich später in Tests umwandeln lassen. Der abzudeckende Graph wird durch das GraphQL Schema definiert wobei nach Definition 9 festzulegen ist, dass die spezielle Menge $N_0 = \{Query - Type\}$ ist und $N_f = N$ (N ohne Query-Type) gilt. Dies folgt, da jeder Knoten entlang eines Pfades des Graphens eine valide Anfrage darstellt. Einzige Bedingung ist hierbei, dass der Endknoten eines Pfades stets ein $SCALAR$ Knoten sein muss. Im Sinne des Integrationstesten ist es wünschenswert, möglichst viele Kombinationen einzelner Module durch diese Pfade abzubilden. Die einzelnen Abdeckungskriterien sind dahingehend zu untersuchen.

4.1 Knotenabdeckung für GraphQL

Die Knotenabdeckung zielt darauf ab, dass jeder Knoten in mindestens einem Testpfad berücksichtigt ist. In GraphQL sind Knoten als Type definiert. Jeder Type definiert seinen eigenen Resolver. Dadurch ist mit der Knotenabdeckung sichergestellt, dass jeder Resolver einmal ausgeführt wird. Ein Type hat jedoch ausgehende Kanten welche ihn mit anderen Resolver verbinden. Der Graph in Abbildung 3.13 vom Schema aus Abbildung 3.12 wäre abgedeckt durch den Pfad $Query \rightarrow Autor \rightarrow Buch \rightarrow Verlag$ Dabei wird allerdings die definierte Kante *autor* ausgelassen welche im Query-Type definiert wurde. Diese kann potenziell Fehler aufweisen und soll daher auch getestet werden. Gleiches gilt für die Kante *autor* des Typen *Buch*. Die Folgerung ist, dass dieses Abdeckungskriterium unzureichend ist um Integrationstest abzubilden.

4.2 Kantenabdeckung für GraphQL

Zuvor wurde deutlich, dass die Abdeckung aller Kanten essenziell ist, um GraphQL gut zu testen. Mit der Kantenabdeckung wird darauf geachtet, dass jede Kante in mindestens einem Testpfad berücksichtigt ist. Die Kantenabdeckung findet auch Anwendung in *Property-based Testing*[6, vgl. D-RQ1]. Allerdings sind im Testkontext von GraphQL alle Kombinationen von Kanten interessant. Durch Kantenabdeckung wäre der Testpfad $Query \rightarrow Buch \rightarrow Autor \rightarrow Buch \rightarrow Verlag$ im Graphen aus Abbildung 3.12

nicht berücksichtigt, obwohl dieser interessant wäre um zu sehen, ob der Kreis richtig aufgelöst wurde. In [6] wurde aber gezeigt, dass mithilfe dieses Abdeckungskriteriums Fehler gefunden werden können.

4.3 Kanten-Paar Abdeckung für GraphQL

In der Kanten-Paar Abdeckung werden alle Kantenpaare betrachtet. Dadurch wird eine bessere Abdeckung der Funktion erreicht als bei der Kantenabdeckung. Das Kriterium stellt eine Verbesserung der Kantenabdeckung dar, allerdings werden eben nur aufeinanderfolgende Kantenpaare abgedeckt. GraphQL kann aber wesentlich tiefere und komplexere Strukturen abbilden. Somit ergibt sich, dass die Kanten-Paar Abdeckung noch nicht ausreichend ist, da insbesondere stark verschachtelte Anfragen hier nicht als Test generiert werden, obwohl diese wahrscheinlich besonders interessant für Tests sind.

4.4 PrimePfad Abdeckung für GraphQL

Die PrimePfad-Abdeckung ermittelt nach Definition 14 die längsten, einfachen Pfade. Dadurch, dass die längsten einfachen Pfade ermittelt werden, wird eine andere Abdeckung als in der Kanten-Paar Abdeckung erreicht. Im Prinzip enthält diese Abdeckung alle möglichen Kantentupel ohne Wiederholungen von Knoten [5, vgl. S. 42]. Dadurch wird erreicht, dass andere Kombinationsmöglichkeit von Knoten und Kanten mit mindestens einem Test berücksichtigt werden. Da GraphQL den Pfad der Anfrage sequentiell abarbeitet, limitiert die Länge der Anfrage die Testausführung nicht.

4.5 Vollständige Pfadabdeckung für GraphQL

Die vollständige Pfadabdeckung verlangt, dass alle möglichen Pfade abgedeckt werden. Da GraphQL jedoch Zyklen erlaubt und das Verbot von Zyklen eine zu starke Einschränkung darstellen würde, ist dieses Kriterium nicht geeignet. Sollte das Schema nativ azyklisch sein, so wäre eine Umsetzung dieses Kriteriums denkbar. Dies soll jedoch keine Voraussetzung sein.

4.6 Schlussfolgerung

Die beiden geeignetsten Abdeckungskriterien sind die vollständige Pfadabdeckung und PrimePfad Abdeckung, wobei die vollständige Pfadabdeckung im Allgemeinen nicht verwendet werden kann, da sie im Fall von Zyklen im Graphen unberechenbar wird. In der Praxis hat sich gezeigt, dass der Großteil der GraphQL-Schemen Zyklen hat. Deswegen ist die vollständige Pfadabdeckung im Allgemeinen nicht zu nutzen und das nächst schwächere Testkriterium ist zu nutzen. Wie in Kapitel 3.6.2 gezeigt gilt, dass die PrimePfad-Abdeckung besser geeignet ist für die Entwicklung von Integrationstests als

die KantenPaar-Abdeckung. Mithilfe der PrimePfad-Abdeckung soll im Folgenden ein Prototyp für Integrationstests von GraphQL-Servern entwickelt werden.

5 Verwandte Arbeiten

Da GraphQL eine stetig wachsende Beliebtheit verzeichnet [17][vgl. Language Features], steigt auch der Bedarf und das Interesse an Testmethoden. Aktuell gibt es für GraphQL noch eine Lücke an produktionsreifen Testtools, insbesondere automatischen Testtools. Eine wachsende Anzahl an Forschungsprototypen beziehungsweise untersuchten Methoden ist allerdings zu verzeichnen. In diesem Kapitel sollen diese Methoden benannt werden und Verwandheiten, Unterschiede oder thematische Überschnitte von dieser und anderen Arbeiten benannt werden.

5.1 Property Based Testing

In *Automatic Property-based Testing of GraphQL APIs* [6] wird der Ansatz des Property-based Testing verfolgt, um Integrationstests zu erstellen. Property-based Testing ist laut dem Paper heute Synonym mit *Random Testing* [6][vgl. 2B], wobei zufällig hierbei meint, dass die Eingabedaten und Pfade zufällig generiert werden. Wie zuvor erwähnt soll diese Arbeit als Motivation für die hier zu entwickelnde Methode dienen, indem die Graphabdeckung verbessert wird. Es sollen Lösungen für einige Limitierungen des *Property-based Testing* gefunden werden. Im Folgenden soll das *Property-based Testing* näher betrachtet werden und die Probleme konkret gezeigt werden. Der allgemeine Funktionsablauf der Testgenerierung laut Paper teilt sich in 6 Schritte auf.

1. Vom Schema, generiere Typ-Spezifikationen
2. Generiere einen Generator, der zufällig eine Liste an Query-Objekten erstellen kann
3. Generiere n Queries
4. Transformiere die Queries in GraphQL-Format
5. Führe die Queries auf dem SUT (System under Test) aus
6. Evaluiere die Ergebnisse auf ihre Properties

[6, vgl. 3. Proposed Method]

Punkt 2 ist der Hauptunterscheidungspunkt beider Arbeiten, denn hier werden zwei gänzlich unterschiedliche Konzepte umgesetzt. Das Konzept beim *Property-based Testing* ist es, einen Query-Generator zu erstellen. Der Query-Generator ist in der Lage, korrekte Anfragen für eine GraphQL-API zu generieren, aus dem zugrundeliegenden Schema. Mithilfe der Clojure-Bibliothek Serene [28] wird es möglich, Clojure.Specs [29] zu generieren. Aus den Clojure.Specs werden mithilfe der Bibliothek Malli[30] dann

Queries zufällig generiert. Malli übernimmt dabei die Aufgabe des Generierens von Argumenten. *Property-based Testing* arbeitet auf Graphstrukturen, nutzt jedoch in der tatsächlichen Verarbeitung nicht diese Gegebenheit. Die hier erarbeitete Methode wird sich davon gänzlich unterscheiden. Im Sinne vom Property-based Testing ist die Herangehensweise allerdings sinnvoll gewesen, da Malli[30] de-facto Standard für Property-based Testing in Clojure ist. Hauptbeitrag der Arbeit war, einen Generator für Malli zu schreiben, der in der Lage ist, GraphQL-Queries zu generieren. Wird davon ausgegangen, dass das Ziel eine möglichst große Abdeckung des Graphens ist, so ist diese Herangehensweise nicht die beste [6][vgl. 3C]. Dies folgt, da die benutzte Bibliothek Malli eben keine Pfade als solches kennt, sondern nur zufällige Nachfolger eines Typens. Daher wird ein Rekursionslimit zwingend benötigt, weil es anders nicht möglich ist, Zyklen in der Abhängigkeit der Typen aufzulösen. Laut dem Paper gilt:*ein größeres und mehr rekursives (GraphQL)-Schema würde nicht skalieren und der (zufällig) iterative Ansatz ist besser als eine Breitensuche* [6][vgl. 3C]. Diese Behauptung lässt Zweifeln, es soll im Folgenden gezeigt werden, dass es möglich ist Algorithmen zu entwickeln, die skalieren und jedes GraphQL-Schema abdecken können.

5.2 Suchbasiertes Testen

EvoMaster[7] ist ein Open-Source Tool, welches sich automatisiertes Testen von Rest-APIs und GraphQL APIs zur Aufgabe gemacht hat. Aktuell kann durch EvoMaster sowohl WhiteBox Testing als auch BlackBox Testing durchgeführt werden, jedoch ist ein Whitebox Test mittels Vanilla-EvoMaster nur für Rest-APIs möglich, die mit der JVM lauffähig sind. Im Paper *White-Box and Black-Box Fuzzing for GraphQL APIs* [31] wurde eine Erweiterung für EvoMaster erstellt, welche GraphQL Tests generieren kann. Hierbei soll sowohl WhiteBox als auch BlackBox Testing möglich sein. Das erstellte Framework des Papers arbeitet nach dem Prinzip, das in Abbildung 5.1 dargestellt ist. WhiteBox Testing ist möglich, insofern Zugang zum GraphQL-Schema und zum Source Code der API gegeben ist. Andernfalls ist nur BlackBox Testing möglich.
Zur Testgenerierung wird ein genetischer Algorithmus genutzt. Ein genetischer Algorithmus ist ein Optimierungsalgorithmus, der von der natürlichen Evolution inspiriert ist. Dabei werden verschiedene Lösungen eines Problems in Generationen erstellt, verändert und nach bestimmten Bedingungen ausgewählt, sodass das gewünschte Ergebnis stetig besser wird [32, vgl.]. Während ein genetischer Algorithmus sich einer Lösung annähert, berechnet er diese jedoch nicht zuverlässig ideal [32, vgl. Fazit]. Im Gegensatz dazu ist der Ansatz dieser Arbeit ein iterativer Algorithmus, der eine ideale Lösung im ersten Durchlauf erreicht. Die ideale Lösung bezieht sich hierbei auf ein Abdeckungskriterium, das die Testpfade erfüllen müssen, die durch den Algorithmus erfüllt werden. Ein genetischer Algorithmus kann das Abdeckungskriterium irgendwann auch erfüllen, jedoch kann keine allgemeine Aussage darüber gemacht werden, wann er dieses erreicht. Während die BlackBox-Tests mit ähnlichen Hürden zu kämpfen haben wie das *Property-based Testing* zeigt sich, dass ein WhiteBox Ansatz eine Verbesserung bringen kann. Hierbei muss allerdings starkes Domänenwissen in den Suchalgorithmus eingearbeitet werden, sodass

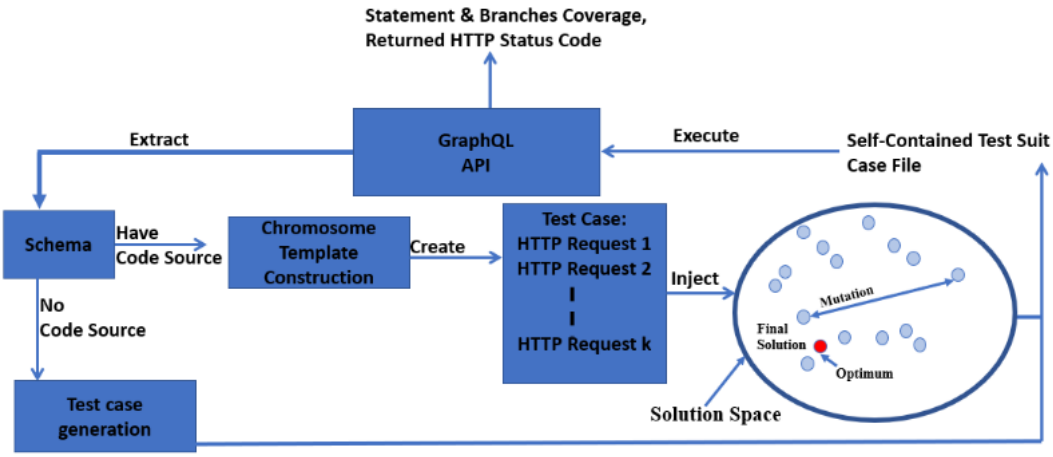

Abbildung 5.1: Arbeitsweise EvoMaster

der Ansatz noch fern von einer WhiteBox-Testautomatisierung ist [31, vgl. Discussion and Future Directions].

5.3 Deviation Testing

Da GraphQL dynamisch auf Anfragen reagiert und es somit möglich ist, in seiner Anfrage einzelne Felder mit einzubeziehen oder auch auszuschließen, ist dies im Grunde genommen ein einzelner Testfall. Im Paper *Deviation Testing: A Test Case Generation Technique for GraphQL APIs* [33] wird diese Gegebenheit benutzt und aus einer selbstdefinierten Query werden verschiedene Variationen erzeugt [33, vgl. 3]. Da Deviation Testing jedoch nur bestehende Tests erweitert, um mögliche Felder mitzutesten, werden hier keine neuen Tests im Sinne der Pfadabdeckung generiert. Durch Deviation Testing werden bestehende Tests nur erweitert. Die durch Variation erstellten Tests stellen stets immer den gleichen Pfad im Graphen dar und nur die Auswahl der verschiedenen *SCALAR* Felder wird verändert. Somit kann Deviation Testing maximal dazu dienen, die Codeabdeckung eines einzelnen Pfads zu maximieren, jedoch nicht die Graphabdeckung insgesamt zu erhöhen.

5.4 Query Harvesting

Klassisches Testen von Anwendungen beinhaltet, dass nach Möglichkeit das komplette System getestet wird, bevor es verwendet wird. Im Paper *Harvesting Production GraphQL Queries to Detect Schema Faults* [34] wird ein gänzlich anderer Ansatz verfolgt. Hierbei ist es nicht wichtig, dass die gesamte GraphQL-API vor der Veröffentlichung getestet ist, sondern echte Queries, die in Produktion ausgeführt wurden, zu sammeln.

Der Ansatz, der hierbei verfolgt wird, begründet sich so, dass ein Testraum für GraphQL potenziell unendlich sein kann und es sehr wahrscheinlich ist, dass nur ein kleiner Teil der API wirklich intensiv genutzt wird, sodass auch nur dieser Teil wirklich stark durch Tests abgedeckt werden muss. Der vorgestellte Prototyp AutoGraphQL läuft hierbei in zwei Phasen, wobei in der ersten Phase alle einzigartigen Anfragen gesammelt werden. In der zweiten Phase werden dann aus den gesammelten Anfragen Tests generiert. Dabei wird für jede gesammelte Anfrage genau ein Test-Case erstellt. Bei dieser Art des Testens wird insbesondere darauf Wert gelegt, dass Veränderungen im GraphQL-Schema zu keinem Fehler führen. Während in dieser Arbeit überprüft wird, dass Queries im laufenden Produktlebenszyklus nicht zu einem Fehler führen, wird außer Acht gelassen, dass eine Query korrekt für AutoGraphQL sein kann, aber trotzdem falsch, indem zum Beispiel falsche Daten zurückgegeben werden durch falsche Referenzierung oder Ähnliches. Somit eignet sich AutoGraphQL vor allem als Monitoring-Software, die gleichzeitig dafür sorgt, dass die Integrität der GraphQL-API bei Veränderungen testbar ist.

5.5 Vergleich der Arbeiten

Folgender Vergleich soll die eben vorgestellten Arbeiten noch einmal kurz einordnen.

Arbeit / Kriterium	**Property Based Testing**	**heuristisch suchenbasiertes Testen**	**Deviation-Testing**	**Query Harvesting**
Generierungsart	Zufallsbasierte Routengenerierung	Heuristische Suche	Erweiterung von bestehenden Tests	Sammeln von Queries und daraus Tests generieren
Überdeckung	Zufällig, stark abhängig von Schema	abhängig, ob Zugang zu Source Code, zufällig aber optimaler	stark abhängig von selbst geschriebenen Tests	stark abhängig von Nutzeranfragen
Orakel	simples Raten	mit Source Code: Analyse	aus entwickelten Tests	aus gestellten Queries
Ausführzeit	vor Produktion	vor Produktion	vor Produktion	Verifikation / Wartung
Use-Case	allgemeines Testen	allgemeines Testen	allgemeines Testen	Testen bei Code-Änderung

Tabelle 5.1: Vergleich der verwandten Arbeiten

6 Testprozess

Nachdem festgestellt wurde, dass die PrimePfad Abdeckung potenziell das sinnvollste Abdeckungskriterium ist, soll im Folgenden eine Methodik entwickelt werden, die es erlaubt, mithilfe des Abdeckungskriteriums Tests für GraphQL zu entwerfen. Die zu entwickelnde Methodik wird in einigen Teilen stark an der Methode aus [6] orientiert sein, das ist jedoch an den betreffenden Stellen kenntlich gemacht. In diesem Kapitel wird die Methodik konzeptionell erstellt und im folgenden Kapitel 7 ein Prototyp entwickelt, der die Methodik umsetzt und validiert. Die zu entwickelnde Methode arbeitet grob nach dem in Abbildung 6.1 gezeigten Muster.

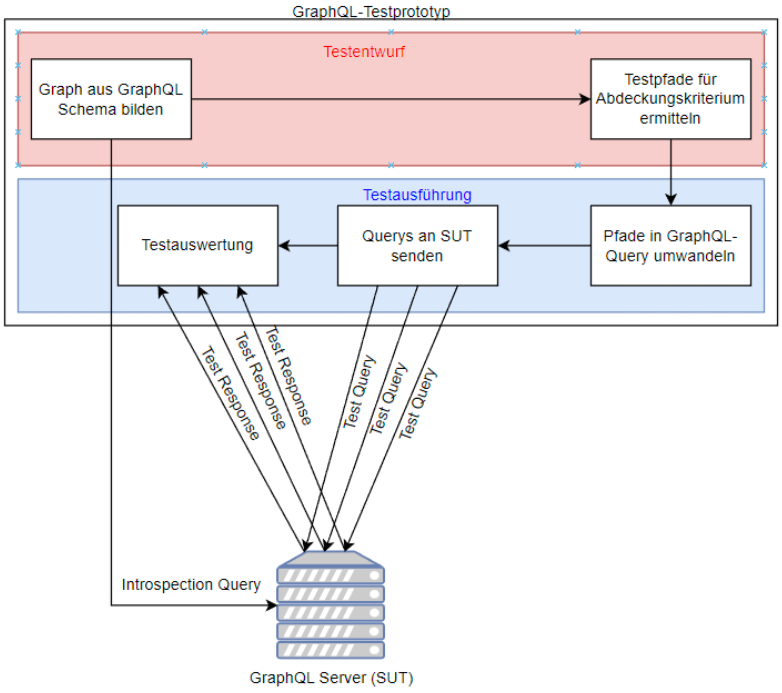

Abbildung 6.1: Grober Ablauf des Testprozesses

Wie in Abbildung 6.1 zu sehen, ist der gesamte Testprozess in zwei Teile aufgeteilt. Einerseits in den Testentwurf, andererseits in die Testausführung. Der Testentwurf basiert auf den zuvor erarbeiteten Theorien und die Testausführung orientiert sich stark am Property-based Testing [6, vgl. Method].

6.1 Testentwurf

Der erste Abschnitt des Testprozesses erarbeitet die Pfadgenerierung nach gewähltem Abdeckungskriterium. Wie zuvor festgestellt, wird die PrimePfad Abdeckung im Folgenden ermittelt. Die Methode erlaubt allerdings auch einen Wechsel des Abdeckungskriteriums, da im Endeffekt nur die Pfade für die weiteren Prozesse genutzt werden können. Bevor jedoch ein Abdeckungskriterium genutzt werden kann, muss das GraphQL-Schema in einen Graphen übersetzt werden.

6.1.1 GraphQL-Schema in Graph abbilden

Laut GraphQL-Specification [15] erlaubt ein GraphQL Server, dass Abfragen über die Schemastruktur des Servers erlaubt sind [15, vgl. 4. Introspection]. Mithilfe der Introspection-Query 12 lässt sich das gesamte Schema eines GraphQL-Servers abrufen. Die Introspection-Query existiert in verschiedenen Varianten. Hier wird die exakt gleiche Version genutzt, wie sie auch von [6] verwendet wird. Ergebnis der Introspection Query ist ein JSON-Objekt mit einer Struktur wie in Listing 6.1.1 gezeigt.

```
1   {
2       "data": {
3           "__schema": {
4               "queryType": {},
5               "mutationType": {},
6               "subscriptionType": {},
7               "types": [],
8           }
9       }
10  }
```

Listing 6.1: Schema-Response

Der Eintrag *queryType* gibt den Namen des Typens an, der Startpunkt jeder Query ist, so wie in Kapitel 4 festgelegt. Im Eintrag *types* ist eine Liste aller Typen enthalten, wobei jeder Eintrag der Struktur im Listing 6.1.1 entspricht.

```
1           {
2             "kind": "",
3             "name": "",
4             "description": "",
5             "fields": [],
6             "inputFields": [],
7             "interfaces": [],
8             "enumValues": [],
9             "possibleTypes": []
10          }
```

Listing 6.2: Type-Field

Um aus dem Schema einen Graphen zu erstellen, werden die Felder *kind*, *name* und *fields* benötigt. Die Angabe *kind* gibt an, von welchem Typ das Feld ist. Hierbei gibt es 9 Möglichkeiten, die dieses Feld annehmen kann.

- **ObjectTypeDefinition (OBJECT):** Repräsentiert ein Objekt mit Feldern.
- **ScalarTypeDefinition (SCALAR):** Eingebaute oder benutzerdefinierte Typen wie `Int`, `Float`, `String`, `Boolean` und `ID`.
- **InputObjectTypeDefinition (INPUT_OBJECT):** Erlaubt das Übergeben komplexer Objekte als Argumente.
- **InterfaceTypeDefinition (INTERFACE):** Repräsentiert eine Liste von Feldern, die andere Objekttypen enthalten müssen.
- **UnionTypeDefinition (UNION):** Kann einen von mehreren Arten von Objekttypen repräsentieren.
- **EnumTypeDefinition (ENUM):** Ein Skalartyp, der auf eine bestimmte Liste von Werten beschränkt ist.
- **ListTypeDefinition (LIST):** Repräsentiert eine Liste von Werten eines bestimmten Typs.
- **NonNullTypeDefinition (NON_NULL):** Ein Modifikator, der angibt, dass der angewandte Typ nicht null sein kann.
- **DirectiveDefinition (DIRECTIVE):** Passt das Verhalten von Feldern oder Typen des Schemas an.

Aus allen Feldern des Typen $OBJECT$ kann ein Graph gebildet werden. Die Menge aller Objekte vom Typ $OBJECT$ ist die Menge aller Knoten des Graphens. Kanten werden dem Graphen hinzugefügt indem die einzelne Typdefinition näher betrachtet wird. Wie in $Type - Field$ gesehen, definiert ein Type immer ein Feld $fields$. In diesem Feld $fields$ verbirgt sich die Informationen aller Kanten, die ausgehend von diesem Knoten sind. Das Feld $fields$ beeinhaltet Objekte folgender Struktur:

```
{
  "name": "",
  "description": "",
  "args": [],
  "type": {},
  "isDeprecated": "",
  "deprecationReason": ""
}
```

Listing 6.3: Type-Field

Wobei für die Ermittlung der Kanten das Feld *type* besonders wichtig ist. Das Feld ist nach Schema aus Listing 6.4 definiert.

```
{
    "kind": "",
    "name": "",
    "ofType": null
}
```

Listing 6.4: Type-Field

Wenn der Eintrag *kind* den Wert $OBJECT$ trägt, so ist klar, dass das hier definierte $OBJECT$ eine Kante zum Knoten *name* besitzt.

6.1.2 PrimePfade ermitteln

Aus dem zuvor ermittelten Graphen sollen Testpfade ermittelt werden, welche die PrimePfad-Abdeckung erfüllen. Hierzu wird der in [5, Finding Prime Test Paths] vorgestellte Algorithmus genutzt. Dabei werden zuerst die einfachen Pfade ermittelt und dann gefiltert. Dies sind Pfade ähnlich zu Definition 15 mit der Lockerung, dass diese Pfade auch Teilpfad eines längeren Pfads sein können [5, vgl. S. 35]. Der längste einfache Pfad kann maximal so lang sein wie die Anzahl der Knoten des Graphens [5, vgl. S.41]. Es wird von jedem Knoten aus expandiert und eine Liste über alle Pfade geführt. Pfade werden nicht weiter expandiert, wenn diese einen Knoten doppelt enthalten. Das Endergebnis ist dann eine Liste aller einfachen Pfade. Diese Liste wird gefiltert, indem alle Pfade, die Teilpfad eines anderen Pfads sind, verworfen werden. Nach Definition 14 wird so die PrimePfad-Abdeckung erlangt. Dies folgt, da die Menge der PrimePfade eine echte Teilmenge der einfachen Pfade ist [5, vgl. S. 35]. Mit der Einschränkung von GraphQL, dass valide Queries stets im Query-Knoten starten müssen, muss sichergestellt werden, dass die PrimePfade dort starten. Um dies umzusetzen, wird festgelegt, dass der kürzeste Weg vom Query Knoten zum Startknoten des PrimePfades zu ermitteln ist und an den PrimePfad anzuhängen, sodass aus diesem später eine valide Query generiert werden kann. Es lässt sich also festhalten, dass zuerst die Pfade für eine PrimePfad Abdeckung berechnet werden müssen und diese dann zu gültigen Testpfaden erweitert werden sollen, sodass sie GraphQL Konventionen einhalten.

6.2 Testausführung

Die ermittelten Pfade werden zu validen GraphQL-Queries umgewandelt und dann ausgeführt um den Test zu validieren. Die Pfadumwandlung in eine valide Query ist noch methodisch stark abweichend zu [6]. Spätere Schritte, also Test ausführen und auswerten sind methodisch gleich zu [6].

6.2.1 Pfade in Query umwandeln

Die Umwandlung eines Pfads in eine Query erfolgt durch die Verwendung der Typinformationen aus dem GraphQL-Schema. Der Pfad wird beginnend im Query-Knoten abgelaufen. Das GraphQL-Schema enthält Informationen darüber, welche Informationen nötig sind, um zum nächsten adjazenten Knoten des Pfads zu kommen. Die Informationen darüber sind im Eintrag *fields* enthalten, wie in Listing 6.1.1 gesehen. Im *fields* Eintrag steht, ob eine Kante Argumente benötigt und welchen Typ das Rückgabeobjekt hat. Das Rückgabeobjekt des *fields* steht dabei aber schon fest, da dieser exakt gleich sein muss mit dem nächsten Knoten des Pfads. In jedem Schritt der Query-Generierung werden stets alle Felder vom Typ *SCALAR* hinzugefügt, damit sichergestellt werden kann, dass der Typ alle Felder implementiert hat. Je nach Implementierung können durch die Feldauswahl weitere Funktionen abgefragt werden, daher ist alles zu inkludieren. Felder vom Typ *OBJECT* werden nur zur Query hinzugefügt, wenn der Typ des *OBJECT* dem nächsten Knoten entspricht. Im Allgemeinen lässt sich das Verfahren durch den Pseudocode in Listing 6.5 darstellen.

```
1    path = (A , B , ..... , Y)
2    query = {}
3
4    while path not empty:
5        knoten = pfad.pop()
6        ScalarFields = getScalarFields(knoten)
7        query.addScalars(ScalarFields)
8        edge = pfad.peek()
9        args = checkForEdgeArgs(edge)
10       query.addArgs(args)
11   return query
```

Listing 6.5: Pseudocode für Pfadgenerierung

Die Argumente, die in einer Query verwendet werden, sind stets nur *SCALAR* Types und somit einfache Datentypen. Es gibt verschiedene Arten die Argumentgeneratoren umzusetzen, vorerst werden diese jedoch methodisch exakt wie in [6] umgesetzt. Dabei wird der Typ des Arguments genutzt, um zufällig ein Argument des entsprechenden Datentyps zu generieren. Ergebnis des Prozesses ist schließlich eine valide GraphQL-Query. In einer konkreten Implementierung ist die Syntax von GraphQL zu beachten, diese ist einsehbar in [15, 2.3 Language Operations].

6.2.2 Queries an Server senden

Die generierten Queries stellen die konkreten Tests für den GraphQL-Server dar. Im Folgenden wird der GraphQL-Server vermehrt SUT (System under Test) genannt. Eine Auswertung der Tests geschieht, indem alle generierten Queries per HTTP-POST an den GraphQL-Server geschickt werden. Die gelieferten Antworten sind zu speichern, dies ist analog zu [6]. Es ist wünschenswert, dass die generierten Queries in einem Testframework abgebildet und gespeichert werden. Dadurch werden die Tests reproduzierbar und können später verwendet werden, um etwaige Fehlerbehebungen zu verifizieren.

6.2.3 Testauswertung

Die Auswertung der Tests basiert auf denselben Annahmen, wie sie in [6] getroffen wurden. Dabei werden die HTTP-Codes der Antworten (im folgenden oft Response) und die existierenden Keys in der Response überprüft. Eine Antwort eines GraphQL-Servers liefert stets einen Statuscode **200**, wenn kein kritischer Fehler auftrat. Kritische Fehler sind stets ein Statuscode **500** [15, vgl. 7. Response]. Daher wird jede Antwort mit einem Code **500** als gefundener Fehler und fehlerhafter Test betrachtet. Eine Antwort mit einem Statuscode **200** kann jedoch auch Fehler aufweisen. Dies wird ersichtlich durch einen zweiten Haupteintrag *errors* in einer Antwort, ersichtlich in Listing 6.6.

```
1  {
2      "data": {}
3      "errors": {}
4  }
```

Listing 6.6: fehlerhafte Antwort

Die Einträge in *errors* müssen jedoch manuell geprüft werden, ob es sich um einen wirklichen Programmierfehler handelt oder gewünschtes Verhalten, da die Zufallsargumente teilweise dafür sorgen, dass Konventionen nicht eingehalten werden können. Die Zufallsargumente sorgen allerdings auch dafür, dass die errechnete PrimePfad Abdeckung nicht praktisch erreicht wird. Sehr häufig kommt es vor, dass zufällig generierte Argumente schon in den Anfängen des Pfads nicht passend zu den vorhandenen Daten sind. Dadurch folgt, dass ein Großteil der Testpfade, die theoretisch eine gute Abdeckung aufweisen, praktisch diese Abdeckung nicht erreicht. Um messen zu können, ob ein Pfad seine theoretische Abdeckung auch praktisch erreicht, wird eine Abschätzung eingeführt.

Abschätzung der Pfadlängen

Mit einer Abschätzung werden die Testergebnisse nicht besser, allerdings können so Informationen darüber gewonnen werden, wie viel vom getesteten Pfad tatsächlich abgedeckt wurde. Dadurch lässt sich der Erfolg der Tests besser abschätzen, da so messbar wird, ob die Queries wirklich die Funktionen ausgeführt haben. Hierzu wird die Pfadlänge des Pfads, der zur Erstellung der Query genutzt wurde, als erwartete Pfadlänge angenommen. Die Pfadlänge der Antwort wird dann als tatsächliche Pfadlänge genommen. Der

Unterschied zwischen erwarteter und tatsächlicher Pfadlänge ist dann das Auswertungsmerkmal für diesen speziellen Test. Die Pfadlänge der Response ist die maximale Tiefe der JSON-Response verringert um 1.

$$\text{Tiefe des Pfades} = \text{Tiefe des JSON-Response-Objekts} - 1$$

Demnach hat die Response aus Listing 6.7 eine Tiefe von 2.

```
{
    "data": {
        "book": {
            id: "1",
            title: "Moby Dick"
            publisher: {
                id: "1",
                name: "Testverlag"
            }
        }
    }
}
```

Listing 6.7: vollständige Response

Eine leere Antwort, wie in Listing 6.8 gezeigt, hat eine Tiefe von 1.

```
{
    "data": {
        "book": null
    }
}
```

Listing 6.8: mangelhafte Response

Obwohl eine leere Response zulässig ist und nicht direkt auf einen Fehler hindeutet, signalisiert der Unterschied zwischen erwarteter und tatsächlicher Länge, ob die Query tatsächlich alle Resolver ausgeführt hat oder nur einen Teil davon. Mithilfe der Abschätzung kann die Qualität der Tests ausgewertet werden. Hierzu kann die Pfadlänge aller erwarteten Pfade addiert werden. Das gleiche wird mit den tatsächlichen Pfadlängen gemacht. Beide Zahlen können dann miteinander in Bezug gesetzt werden, um eine prozentuale Einschätzung zu erlangen, wie viel Prozent der Tests tatsächlich ausgeführt wurden.

Definition 17

$$\textit{Prozent der tatsächlichen Abdeckung} = \frac{\textit{tatsächliche Gesamtpfadlänge}}{\textit{erwartete Gesamtpfadlänge}} * 100$$

Ein Wert von 100% ist anzustreben. Dies würde bedeuten, dass die generierten Tests auch alle Funktionen getestet haben. In der Praxis wird der Wert jedoch sehr wahrscheinlich darunter liegen.

Abschließend soll erklärt werden, wie es möglich ist, den Wert der tatsächlichen Abdeckung zu erhöhen. Dies geschieht durch zwei verschiedene Methoden.

Zufallsgeneratoren der Argumente

Die zuvor vorgestellte Abschätzung liefert einen Hinweis darüber, wie gut die Queries tatsächlich getestet wurden. Ein Ansatz, der die Queries eine bessere tatsächliche Abdeckung erreichen lässt, ist das Anpassen der Generatoren für die Argumente. Bei der in Kapitel 6.2.1 vorgestellten Methode werden Argumente komplett zufällig für den zugrundeliegenden Datentyp erstellt. Dies bedeutet, dass zum Beispiel der Type ID als String gewertet wird. Eine ID unterliegt jedoch in den allermeisten Implementierungen bestimmten Konventionen. So wäre ein Generator für zufällige Strings nicht in der Lage, solche Konventionen zuverlässig abzubilden. Ein Anpassen des Argumentgenerators für die ID an die verwendete Konvention des SUT hilft dabei, dass signifikant bessere Ergebnisse erzeugt werden. Alternativ kann auch eine Liste aller existenten IDs angegeben sein und zufällig ein Element ausgewählt werden. Die Anpassung der Argumentgeneratoren an die zugrundeliegenden Daten ist höchst spezifisch und daher kann keine allgemeine Vorgehensweise ermittelt werden. Ziel der Anpassung ist es, dass die Chance erhöht wird, dass Argumente zufällig generiert werden, die dann tatsächlich zu den zugrunde liegenden Daten passen.

Anzahl der Queries

Die Wahrscheinlichkeit, passende Argumente zu generieren, steigt mit der Anzahl an generierten Queries. Wird die Anzahl an generierten Queries pro Pfad erhöht, so steigt auch die Wahrscheinlichkeit, dass zumindest ein Pfad eine gute Abdeckung erreicht. Hierfür muss die Methode aus Kapitel 6.2.1 so oft wie gewünscht wiederholt werden. In jeder Generierung muss jedoch das Ergebnis sein, dass verschiedene Argumente für die Query generiert wurden. Zusammen mit den angepassten Argumentgeneratoren kann so die zufallsbasierte Argumentgeneration begrenzt werden und es ist wahrscheinlicher, dass gute Tests entstehen.

6.3 Zusammenfassung der Methode

Die eben entwickelte Methode funktioniert so wie in Abbildung 6.1 gezeigt. Teile der Methode bedienen sich an Methoden die in [6] eingeführt wurden. Der komplette Prozess der Testgenerierung wurde in der hier entwickelten Methode jedoch verändert. So wurde die Graphstruktur von GraphQL genutzt, um aus dieser Testpfade zu generieren, die dann in GraphQL-Queries umgewandelt werden. Auf diese Weise war es möglich, die Limitierung des Rekursionslimits vom *Property-based Testing* zu beseitigen. In der Testauswertung wurde sich stark am *Property-based Testing* orientiert. Die Einführung der erwarteten Pfadlänge gegenüber der tatsächlichen Pfadlänge ist ein neuer Ansatz,

der die Qualität der zu testenden Queries messbar macht. Das fehlt im *Property-based Testing* .

Im Folgenden soll die entwickelte Methode in einem Prototypen umgesetzt werden. Dieser Prototyp soll dann in Experimenten zeigen, dass die entwickelte Methode in der Lage ist, Fehler in GraphQL-APIs zu finden. Ein Vergleich mit dem Prototypen von *Property-based Testing* soll zeigen, ob die Methode Verbesserungen bringt.

7 Testautomatisierung

Nach der Einführung der Methode im vorherigen Kapitel soll der entwickelte Prototyp erklärt werden. Der entwickelte Prototyp ist im GitHub und BTU-GitLab zu finden. Eine Anleitung findet sich in der Readme im Root-Verzeichnis. Voraussetzung zum Ausführen der Anwendung ist Python mit einigen Dritt-Bibliotheken, die in der Readme vermerkt sind.

7.1 Auswahl der Bibliotheken

Um die vorgestellte Methode umzusetzen, war insbesondere wichtig, dass eine einfache und mächtige Bibliothek für die Definition und Bearbeitung von Graphen zur Verfügung steht. Die erste Wahl fiel hierbei auf NetworkX, eine Graphenbibliothek für Python. Sie wurde ausgewählt, da der Ersteller schon einige Erfahrungen mit dieser Bibliothek hat und somit eine effiziente Umsetzung ohne langwierige Einarbeitung möglich war. Durch die Auswahl der Bilbiothek wurde gleichzeitig auch die Sprache Python festgelegt. Einige weitere Bibliotheken wurden benötigt, um den Applikationsstack zu vervollständigen. Insgesamt waren Bibliotheken in den Bereichen Graphen, API-Kommunikation, JSON-Bearbeitung und Argumentgenerierung nötig, um einen Prototypen umzusetzen. Es werden nicht alle Bibliotheken eine Berücksichtigung hier finden, sondern nur diese, die einen signifikanten Einfluss auf das Programm haben und besonders herausstechen.

7.1.1 NetworkX

NetworkX ist eine Python-Bibliothek für *Erstellung, Manipulation und Untersuchung der Struktur, Dynamik und Funktionen komplexer Netzwerke* [35, vgl. Startseite] Mit *12.8k* [36] Sternen auf GitHub ist networkX eine sehr beliebte Bibliothek. NetworkX ist die ideale Wahl, um Graphen zu erstellen, denn es nimmt jeden möglichen Datentypen als Wert für einen Knoten und Kante. Somit kann sehr direkt ein Graph definiert werden. Für ein einfaches Beispiel von Author, Book, Publisher und deren Verbindungen wird nur der Code aus Listing 7.1 benötigt.

```
1 import networkx as nx
2
3 G = nx.Graph()
4 G.add_edge("Query", "Book", "book")
5 G.add_edge("Query", "Author", "author")
6 G.add_edge("Query", "Publisher", "publisher")
7
```

```
8  G.add_edge("Publisher", "Book", "book")
9  G.add_edge("Book", "Publisher", "publisher")
10
11 G.add_edge("Book", "Author", "author")
12 G.add_edge("Author", "Book", "book")
```

Listing 7.1: NetworkX-Graphen erstellen

Diese wenigen Zeilen reichen aus, um einen Graphen mit allen Knoten und Kanten zu definieren. Der Feldbezeichner kann dabei als Kantengewicht angegeben werden. So ist möglich, dass sofort geschlussfolgert werden kann welche Kante genutzt werden muss, um zum nächsten Typ des Pfads zu gelangen. Auf diesem Graphen können diverse Algorithmen ausgeführt werden. Dabei helfen diverse Hilfsfunktionen, die Effizienz der Programmierung zu erhöhen. Hierbei seien insbesondere folgende Hilfsfunktionen genannt:

draw

```
1 nx.draw(G, with_labels=True)
```

Listing 7.2: Ein Graphen visualisieren

Die Funktion zeichnet den erstellten Graphen in ein beliebiges Format. So ist eine Visualisierung eines Graphen möglich.

shortest_path

```
1 shortest_path = nx.shortest_path(G, Node1, Node5)
```

Listing 7.3: Der kürzeste Weg zwischen zwei Knoten

Die Funktion *shortest_path* gibt eine Liste von Kanten zurück, die den kürzesten Weg zwischen zwei Knoten angibt.

neighbors

```
1 G.neighbors(Node)
```

Listing 7.4: Alle Nachbarn eines Knoten

Diese Funktion liefert alle direkten Nachbarn eines Knotens.

simple_paths

Die Funktion aus Listing 7.5 berechnet alle einfachen Pfade, wie in Kapitel 6.1.2 gewünscht. So ist es direkt möglich, aus dem Ergebnis dieser Funktion die PrimePfade herauszufiltern. Dadurch wird die Bibliothek die Berechnung der Testpfade vereinfachen.

```
nx.all_simple_paths(G, source=start_node, target=end_node)
```

Listing 7.5: Alle einfachen Pfade zwischen zwei Knoten

7.1.2 Faker

Die gewählte Argumentgenerierungsbibliothek ist *Faker* [37]. Mit *16k* [37] Sternen auf GitHub ist Faker auch eine beliebte Bibliothek. Faker ist eine Bibliothek, die es sehr einfach macht, Daten zu generieren. Da im Kontext von GraphQL nur sehr einfache Datentypen als Argumente benötigt werden, reicht diese Bibliothek komplett aus, da sie es schafft, schnell und unkompliziert Daten in genau dem Format zu generieren, wie sie benötigt werden. Angenommen es wird ein String benötigt der 10 Zeichen lang ist. So reicht der Code aus Listing 7.6 aus.

```
random_string = fake.pystr(min_chars=10, max_chars=10)
```

Listing 7.6: Ein zufälliger String

Sollte eine Zufallszahl benötigt werden, so ist der Code aus Listing 7.7 ausreichend.

```
random_number = fake.random_int(min=1, max=1000)
```

Listing 7.7: Eine zufällige Zahl

Generell ist der Großteil aller Datentypen, die hier zufällig generiert werden müssen, Einzeiler. Daher wird diese Bibliothek für die Datengenerierung gewählt.

7.1.3 PyTest

Um die gewünschte Reproduzierbarkeit aus Kapitel 6.2.2 zu erreichen, wird das Testframework PyTest verwendet. PyTest ist ein Testframework für Python, welches eine simple und einfache Testdefinition ermöglicht. Ein Test für die Funktion *inc* kann mit *test_inc* umgesetzt werden wie in Listing 7.8.

```
def inc(x):
    return x + 1

def test_inc():
    assert inc(3) == 5
```

Listing 7.8: Unit mit Unit-Test

Die einfache Syntax von PyTest reicht für den hier benötigten Anwendungsfall aus. Gleichzeitig sind Imports in PyTest einfach umzusetzen, daher wird dieses Testframework genutzt.

7.2 Umsetzung der Methode

Für die Umsetzung der Methode werden die einzelnen Teile des Codes präsentiert und erklärt. Dabei wird chronologisch vorgegangen, so wie in den einzelnen Schritten der Methode definiert. Im Allgemeinen funktioniert der Prototyp, so wie in dem Sequenzdiagramm in Abbildung 7.1 gezeigt.

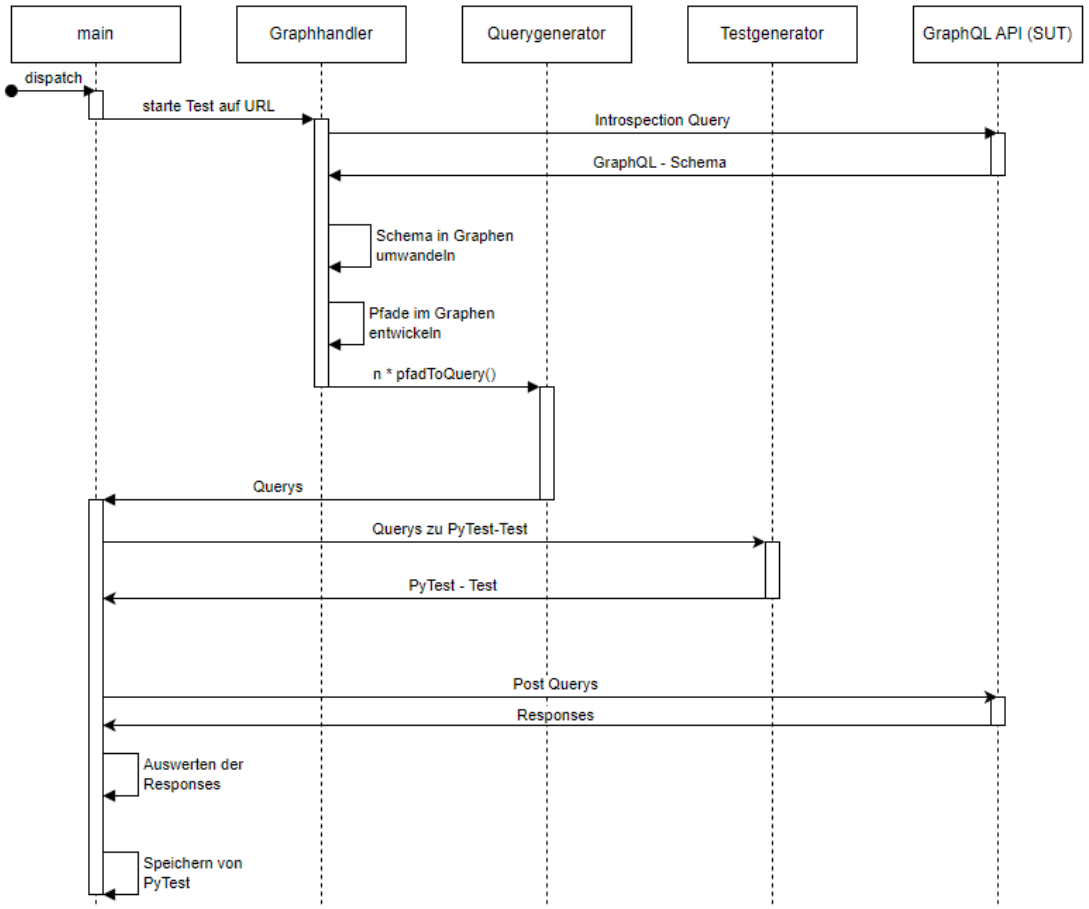

Abbildung 7.1: Sequenzdiagramm des Prototypens

Hierbei sind auch die einzelnen Module zu erkennen. Dabei sind die Module *main* (main.py), *Graphhandler* (graphhandler.py), *Querygenerator* (querygenerator.py) und *Testgenerator* (pytestgenerator.py) Teile des Prototypens. Das Modul *GraphQLAPI* stellt das zu testende System dar und ist extern. Die Module *Graphhandler*, *Querygenerator* und *Testgenerator* arbeiten chronologisch. Hierbei übernimmt der *Graphhandler* alle Teile des Testentwurfs also alle Teile des Kapitels 6.1. Der *Querygenerator* übernimmt die Umsetzung der Methode aus Kapitel 6.1. Im *Testgenerator* werden die PyTest-

Tests generiert, dies ist ein gewünschtes Feature aus Kapitel 6.2.2. Die *main* verwaltet die Zusammenarbeit der einzelnen Module und sorgt für die Umsetzung aller übrigen Funktionen.

7.2.1 Schema in Graph abbilden

Wie in der Vorstellung der Methode im Kapitel 6.1.1 erwähnt, muss das GraphQL-Schema in einem Graphen abgebildet werden. Der Graph wird in einem gerichteten NetworkX-Graphen abgebildet. In einem ersten Schritt wird jedoch die Introspection-Query 12 ausgeführt, um Informationen über das Schema des SUT zu erlangen. Es sei angemerkt, dass einige GraphQL APIs so eine Introspection-Query verbieten, sei es einerseits durch direktes Unterbinden oder ein Tiefenlimit in den Queries. Das SUT muss in jedem Fall die Introsepction-Query unterstützen, da sonst keine Informationen über das GraphQL-Schema erlangt werden können. Die Query wird mit einem simplen HTTP-POST an die zu testende URL gesendet.

```
1 r = requests.post(testUrl, json={'query': queries.
    introspection_query})
2 json_data = json.loads(r.text)
```

Listing 7.9: Absenden der Introspection-Query

Die Response erfolgt als JSON-Objekt. Erhaltene Daten werden in der Variable $json\_data$ gespeichert. Im *Graphhandler* Modul wird dann mit der Funktion *buildGraph* ein gerichteter NetworkX-Graph aus dem GraphQL-Schema erstellt. Diese generiert rekursiv einen Graphen von einem gegebenen Startknoten, einem leeren Graphen und dem Schema. Es werden nur erreichbare Knoten vom Startknoten berücksichtigt. Wird der Startknoten auf den Knoten *Query* gesetzt, so wird jeder erreichbare Teil des Graphens hinzugefügt, der vom *Query* Knoten erreichbar ist. Dies ist insofern sinnvoll, als andere Typen, wenn sie nicht von *Query* aus erreichbar sind, nicht Teil des Testraumes wären, da diese in keiner validen Anfrage vorkommen können. Die Funktion, die den Graphen generiert, ist in Abbildung 7.2 dargestellt.
Die Funktion buildGraph arbeitet rekursiv. Vom Startknoten aus ruft die Funktion alle Folgeknoten von Query auf. Dies sind alle Knoten, die den Type $OBJECT$ besitzen und nicht mit einem __ beginnen oder ein Basisdatentyp sind. GraphQL kann eigene Objekte definieren, welche mit __ starten. Diese werden explizit ausgeschlossen, genau wie alle $SCALAR$ Types. Jeder Knoten definiert in seinem $fields$ Eintrag, zu welchen Feldern er Beziehungen hat. Hierbei muss unterschieden werden, dass ein Eintrag entweder vom Type $OBJECT$, NON_NULL oder $LIST$ sein kann, um zulässig zu sein. Sollte es sich um einen $LIST$ oder NON_NULL Eintrag handeln, muss geprüft werden, von welchem Type diese sind. Wenn ein Knoten alle Bedingungen erfüllt, so wird dieser dem Graphen hinzugefügt und auf ihm selbst wird *buildGraph* ausgeführt. So wird der gesamte Graph rekursiv aufgebaut und jeder erreichbare Knoten vom Startknoten wird einbezogen.

```
def buildGraph(graph, type_name, type_dict):
    if type_name.startswith(nonSchemaTypePrefix) or
       type_name in baseDatatypes:
        pass
    else:
        for adjacentNode in type_dict[type_name]['fields']:
            if graph.has_edge(type_name, adjacentNode['type'
               ]['name']):
                return
            else:
                if adjacentNode['type']['name'] and
                   adjacentNode['type']['name'] not in
                   baseDatatypes:
                    graph.add_edge(type_name, adjacentNode['
                       type']['name'])
                    graph[type_name][adjacentNode['type']['
                       name']]["data"] = adjacentNode
                    buildGraph(graph, adjacentNode['type']['
                       name'], type_dict)
                if adjacentNode['type']['kind'] == 'LIST'
                   and adjacentNode['type']['ofType']['name'
                   ] not in baseDatatypes:
                    graph.add_edge(type_name, adjacentNode['
                       type']['ofType']['name'])
                    graph[type_name][adjacentNode['type']['
                       ofType']['name']]["data"] =
                       adjacentNode
                    buildGraph(graph, adjacentNode['type']['
                       ofType']['name'], type_dict)
```

Abbildung 7.2: Funktion die einen gerichteten Graphen aufspannt

7.2.2 Pfade aus Graph bilden

Der Graphhandler implementiert verschiedene Abdeckungskriterien. Das Tool benötigt im Verlauf eine Liste *paths* aller Pfade, die für die Testgenerierung berücksichtigt werden sollen.

```
1 paths = graphhandler.generate_prime_paths("Query", graph)
```

Abbildung 7.3: *paths* als Liste von Pfaden für ein Abdeckungskriterium

Wie im Kapitel 4.6 festgestellt, gilt, dass die PrimePfad-Abdeckung am geeignetsten ist und somit wird diese umgesetzt. Die PrimePfad Abdeckung ist durch die Funktion *generate_prime_paths* implementiert. Diese Funktion verknüpft dabei zwei andere Funktionen und ist in Listing 7.10 gezeigt.

```
1 def generate_prime_paths(startknoten, g):
2     return shortest_path_to_prime(g, startknoten,
          get_prime_paths(g, startknoten))
```

Listing 7.10: valide PrimePfad Generierung

Da PrimePfade nicht im Query Knoten starten müssen, wird zuerst die Funktion *get_prime_paths* aufgerufen. Diese gibt eine Liste aller PrimePfade zurück. Da jedoch ein Testpfad stets im Query-Knoten starten muss, wird anschließend *shortest_path_to_prime* ausgeführt. Diese Funktion ermittelt den kürzesten Weg vom Query-Knoten zum Startknoten des Prime-Pfades. So kann sichergestellt werden, dass die generierten PrimePfade stets korrekte Testpfade sind. Die Implementierung der Funktion *get_prime_paths* ist im Listing 7.11 gezeigt.

```
1 def get_prime_paths(G, start_node):
2     startPaths = [(n, ) for n in g.nodes()]
3     simplePaths = []
4     findSimplePath(g, startPaths, simplePaths)
5     primePaths = sorted(simplePaths, key=lambda a: (len(a),
          a))
6     return primePaths
```

Listing 7.11: PrimePfad Algorithmus

Von jedem Knoten ausgehend werden die einfachen Pfade ermittelt. Anschließend werden die Pfade so gefiltert, dass nur die längsten Pfade, die kein Teilpfad eines anderen Pfads sind, zurückgegeben werden. Dies entspricht laut Definition 15 einem PrimePfad und der Algorithmus ist Deckungsgleich mit [5, Finding Prime Test Paths S. 39]. Für Pfade die keinen Ursprung im Query-Knoten haben wird der kürzeste Weg vom Queryknoten zum Startknoten des Pfades ermittelt. So wird die Korrektheit der Testpfade sichergestellt.

7.2.3 Queries aus Pfad ermitteln

Aus den entwickelten Pfaden sollen Queries entwickelt werden, sodass diese an die GraphQL-API gestellt werden können. Eine Query beginnt in GraphQL immer im Query-Knoten und so beginnt jeder Pfad in diesem Knoten. Um die Wahrscheinlichkeit zu erhöhen, dass ein Pfad gut abgedeckt wird, werden pro Pfad mehrere Queries entwickelt. Hierfür wurde die Variable $testPerPath$ angelegt, diese legt fest, wieviele Queries pro Pfad generiert werden sollen. Standardmäßig gilt: $testPerPath = 5$. Die Generierung der Queries übernimmt die Methode *pathToQuery* vom Modul *Querygenerator*. In der Methode *pathToQuery* wird ein Pfad mithilfe eines Typedict und dem Graphen zu einer validen GraphQL-Query umgewandelt. Die Funktion *pathToQuery* entfernt den Startknoten Query und reichert den Pfad an. Ihre Implementierung ist in Listing 7.12 dargestellt.

```
def pathToQuery(path, typedict, graph):
    path_edges = list(zip(path[:-1], path[1:]))
    query = "{ " + resolvePathTillOnlyScalarTypesOrEnd(
       path_edges, typedict, graph) + " }"
    return query
```

Listing 7.12: Funktion pathToQuery

Die Funktion erstellt eine Query, indem der angegebene Pfad abgelaufen und mit Argumenten angereichert wird. Das Ablaufen des Pfads und die Anreichung mit Argumenten übernimmt die rekursive Funktion $resolvePathTillOnlyScalarTypesOrEnd(path, typedict, graph, query =)$ die in Listing 7.13 dargestellt ist.

```
def resolvePathTillOnlyScalarTypesOrEnd(path, typedict,
   graph, query=""):
    if path:
        edge = path.pop(0)
    else:
        return query
    edge_data = graph[edge[0]][edge[1]]["data"]
    if len(edge_data['args']) < 1:
        if edge_data["type"]["kind"] == "SCALAR":
            query = query + " " + edge_data["name"] + " "
        else:
            query = query + " " + edge_data["name"] + " { "
               + addScalarTypes(edge_data["type"], typedict,
                edge_data["name"]) + " " +
               resolvePathTillOnlyScalarTypesOrEnd(path,
               typedict, graph, query) + " } "
    else:
        argString = edge_data['name'] + "("
```

```
14          for args in edge_data['args']:
15              argString = argString + args["name"] + ": " +
                    resolveArg(args["type"], typedict) + ", "
16          argString = argString + ")"
17          query = query + argString + " { " + addScalarTypes(
                edge_data["type"], typedict, edge_data["name"]) +
                 " " + resolvePathTillOnlyScalarTypesOrEnd(path,
                typedict, graph, query) + " } "
18      return query
```

Listing 7.13: Pfadumwandlung in Query

Die Funktion fügt für jeden Knoten alle definierten *SCALAR* Felder hinzu. So wird sichergestellt, dass alle einfachen Felder eines Objektes abgefragt werden. Es wird so validiert, dass das Objekt übereinstimmt mit der Schema-Definition. Sind alle *SCALAR* Types hinzugefügt, so wird geprüft, welches Feld vom Type *OBJECT* hinzugefügt werden muss, um die Kante zum nächsten Knoten abzubilden. Sollte diese Kante Einträge im *args* Feld besitzen, so werden passende Argumente generiert. Da Argumente nur *SCALAR* Types oder Aggregationen von *SCALAR* Types sein können, werden Datengeneratoren für die Standarddatentypen benötigt. Die Zuweisung der Datengeneratoren geschieht hierbei mit der Funktion *resolveArg*. Benötigt ein *OBJECT* Feld Argumente, so werden diese der Query hinzugefügt, indem mit *resolveArg* die Argumente zur Verfügung gestellt werden. Anschließend wird die Kante aus der Liste des Pfads entfernt und die Funktion wieder rekursiv aufgerufen. Abbruchbedingung ist, dass der Pfad keine Kanten mehr besitzt. Wenn der Pfad keine Kanten mehr besitzt, so ist die entwickelte Query ein Test, der den Pfad abdeckt. Es sei angemerkt, dass durch die zufällige Argumentgenerierung keinesfalls garantiert ist, dass bei der Testausführung der volle Pfad getestet wird. Sollte zum Beispiel ein Pfad direkt am Anfang Argumente benötigen, diese aber jedoch zu keinen Daten des SUT passen, so ist es wahrscheinlich, dass der Test erfolgreich sein wird, ohne dass der eben entwickelte Pfad wirklich komplett getestet wird.

7.2.4 Tests ausführen & Testdatei generieren

Bevor die Tests ausgeführt werden, sind diese im PyTest-Format zu speichern. Dafür wird eine Datei angelegt, die alle nötigen Imports enthält. Anschließend wird für jede Query ein eigener Test angelegt und die Auswertung festgeschrieben. Implementiert wurde dies in der Funktion *generatTestFromQuery* im Modul *Testgenerator*. Ein so erstellter PyTest ist in Listing 7.14 dargestellt.

```
1 def testQuery6aa8b26(caplog):
2     caplog.set_level(logging.WARNING)
3     response = requests.post(testUrl,json={'query':"{...}")
4     response_as_dict = json.loads(response.text)
5     measurement = queries.compareQueryResults(
          response_as_dict, "{...}")
6     if measurement["expectedPathLength"] > measurement["
          pathLengthFromResult"]:
7         logging.warning(" Test hat nicht 100% Abdeckung ")
8     assert response.status_code == 200
```

Listing 7.14: PyTest einer Query

Nachdem mit den PyTests die Queries reproduzierbar gemacht wurden, werden die Queries ausgeführt. Hierbei setzt der Code aus Listing reftestt die Ausführung um.
Hierbei wird der Code aus Listing 7.15 genutzt. Eine Query wird als HTTP-POST an das SUT gestellt, die Antwort wird dann auf ihre Pfadlängen hin untersucht und in einer Datei abgespeichert.

```
1 queryResults = []
2 for testQuery in primePathQueries:
3     r = requests.post(testUrl, json={'query': testQuery},
          headers=HEADERS)
4     response_as_dict = json.loads(r.text)
5     measurement = queries.compareQueryResults(
          response_as_dict, testQuery)
6     queryResults.append([testQuery, r, measurement])
7     f.write(" => ".join([testQuery, r.text]))
8     f.write("\n")
9 f.close()
```

Listing 7.15: Ausführen einer Testquery

Um die Tests auszuführen, werden alle generierten Queries an das SUT gesendet. Die Antworten werden bemessen, gespeichert und später ausgewertet.

7.2.5 Testauswertung

Die Testauswertung erfolgt, wie zuvor gesehen, auf zweierlei Arten. Einerseits werden Tests ad-hoc ausgeführt. Andererseits wird eine Datei mit PyTests generiert. Die

Auswertung der Testqueries in der PyTest-Datei erfolgt nach PyTest-Standard. Dabei werden Hinweise geliefert, wenn etwas von der Erwartung abweicht. Eine Auswertung der Testqueries, die direkt ausgeführt wurden, erfolgt nach einer Kategorisierung. Die Kategoriesierungen sind **Good_Test**,**Perfect_Test**,**malformed_Test** und **confirmed_failed_Test**. Ein **Good_Test** ist ein Test, der keinen Fehler erzeugt hat. Dieser hat allerdings auch nicht die erwartete Pfadlänge von der Response erfüllt, es war also ein Test, der nicht die komplett gewünschte Abdeckung erreicht hat. **Perfect_Test** sind Tests, die keinen Fehler erzeugt haben und die erwartete Pfadlänge entspricht der Pfadlänge der Response. Eine solche Query hat den Pfad, der zu testen war, ideal abgedeckt. **malformed_Test** sind Tests, die fehlerhaft sind, aber allerdings aufgrund von Generierungsfehlern des Prototypen. **confirmed_failed_Test** sind Tests, bei denen tatsächlich einen Fehler gefunden wurde. Die fehlerhaften Tests werden dann im Folgenden ausgegeben mit der konkreten Fehlerbeschreibung, sodass eine Fehleranalyse möglich wird. Um Maß darüber zu halten, wie viele Tests in welcher Kategorie sind, wurde eine Auswertung geschrieben, welche in Listing 7.16 gezeigt ist.

```
successfull = 0
perfect = 0
own_failure = 0
server_failures = 0
testCount = 0

for queryResult in queryResults:
    testCount = testCount + 1
    if any(substring in queryResult[1].text for substring in
        ["GRAPHQL_PARSE_FAILED", "GRAPHQL_VALIDATION_FAILED"
       ]):
        own_failure = own_failure + 1
    elif "INTERNAL_SERVER_ERROR" in queryResult[1].text or r
       .StatusCode == 500:
        server_failures = server_failures + 1
    elif "data" in queryResult[1].text and queryResult[2]["
       expectedPathLength"] > queryResult[2]["
       pathLengthFromResult"]:
        successfull = successfull + 1
    elif "data" in queryResult[1].text and queryResult[2]["
       expectedPathLength"] == queryResult[2]["
       pathLengthFromResult"]:
        perfect = perfect + 1
```

Listing 7.16: Auswertung der Antworten

7.3 Zusammenfassung der Implementation

Der Python Prototyp stellt eine Implementierung der Methode aus Kapitel 6 dar. Wie später gezeigt wird, ist der Prototyp in der Lage, reale Fehler in GraphQL-APIs zu finden. Die modulare Aufbauweise des Prototypens erlaubt es, dass zukünftige Anpassungen an der Software einfach umsetzbar sind. So sind die Module thematisch getrennt und das Anpassen sowie Auswechseln einzelner Module ist möglich, ohne die Funktionsweise der anderen Module zu beeinträchtigen.

8 Auswertung und Vergleich mit Property-based Testing

8.1 Vergleichmetriken

Bevor ein Vergleich beider Methoden durchgeführt werden kann, müssen erst einmal Metriken eingeführt werden, in denen sich verglichen werden soll. Im Fokus des Vergleichs stehen die in *Property-based Testing* [6] genutzten Metriken.

8.1.1 Metriken aus Property-based Testing

In *Property-based Testing* wurden zwei Metriken eingeführt, um die Methode zu evaluieren. Es wurden zwei Forschungsfragen vorgestellt.

1. Welche Schema Coverage kann mit der Methode erreicht werden? [6, vgl. RQ1]
2. Wie gut ist die Fehlerfindungskapazität der Methode? [6, vgl. RQ2]

Zur Beantwortung der Fragen wurden Experimente auf zwei Testsystemen ausgeführt. Das erste Testsystem ist eine eigens entwickelte GraphQL-API, die bekannte Fehler besitzt [6, vgl. A.1]. Testsystem zwei ist GitLab, eine häufig genutzte Software für GitServer mit DevOps Kapazitäten. Gitlab bietet seine API auch als GraphQL an und durch seine riesige Größe eignet sich GitLab als solides Testsystem [6, vgl. A2]. Der hier entwickelte Prototyp soll in exakt dem gleichen Umfeld seine Tests generieren. Es ist zu erwarten, dass die selben Fehler gefunden werden wie in der Methode des *Property-based Testings*. Ideal wäre es, neue Fehler zu finden. Beide Forschungsfragen werden im Folgenden noch einmal näher erläutert, da diese speziell sind und Wissen über die Methode wichtig ist, um die Ergebnisse korrekt einordnen zu können. Die Ergebnisse der Experimente sind im GitLab zu finden.

8.1.2 Fehlerfindungskapazität

Die Fehlerfindungskapazität ist eine Metrik, die messen soll, wie zuverlässig die Methode tatsächlich Fehler findet. Hierfür werden die beiden zuvor benannten APIs getestet und es wird geprüft, ob die Methode die Fehler finden konnte. Um zu verifizieren, dass die Methode möglichst viele Fehler findet, wurde eine Test API entwickelt, die mit bekannten Fehlern versehen wird. Der Prototyp des *Property-based Testing* fand 11 von 15 Fehlern im speziell vorbereiteten System. Bei GitLab wurden 4 Bugs im Query-Bereich gefunden. Der hier entwickelte Prototyp soll mindestens die gleichen Fehler finden und idealerweise mehr.

8.1.3 GraphQL-Schema Abdeckung

Dadurch, dass *Property-based Testing* auf zufallsbasierter Testgenerierung basiert, stellt sich die Frage, wie gut die Methode die API abdeckt und inwiefern die generierten Tests ausreichend sind. Dies kommt insbesondere zu tragen, wenn die maximale Pfadlänge ausgehend vom Query-Knoten größer ist als die erlaubte Rekursionstiefe des Prototypens. In Property-based Testing wird definiert, dass die generierten Tests eine vollständige Abdeckung erreichen, wenn Definition 18 gilt. Definition 18 entspricht im Allgemeinen der Kantenabdeckung nach Definition 12.

Definition 18 *Für alle Objekte des Schemas: Bilde alle Tupel* {*Object, Field* }*. Ein Schema hat eine ideale Coverage, wenn alle Tupel durch einen Test abgedeckt sind [6, vgl. B. Measuring Schema Coverage].*

Es sei erwähnt, dass die hier angesprochene Abdeckung eine theoretische Abdeckung ist. Die tatsächlich erreichte Abdeckung wird nicht betrachtet.

8.2 Threats to Validity / Limitierungen

Bevor der eigentliche Vergleich beginnt, soll eingeordnet werden, inwiefern die Experimente zu betrachten sind. Es wird gezeigt, welche Voraussetzungen nötig sind, um die hier erzielten Ergebnisse zu erlangen.

8.2.1 Argumentgeneratoren

Wie in Kapitel 6.2.3 erwähnt, ist es wichtig, dass GraphQL für jede Funktion einen Wert ungleich *null* liefert, damit der Pfad weitergegangen werden kann und die Funktionen in diesem getestet werden, um die tatsächliche Abdeckung zu erhöhen. Um die Wahrscheinlichkeit zu erhöhen, dass Argumentgeneratoren ein Argument zurückliefern die zum SUT passen, wurden diese angepasst. Dabei wird die Vergleichbarkeit der Arbeiten nicht verletzt, da diese Anpassungen der Argumentgeneratoren in [6] auch vorgenommen wurden [6, vgl. Experimental Setup and Method]. Hierbei sei zum Beispiel erwähnt, dass eine Type ID in GraphQL als String wert definiert ist, häufig in Implementierung jedoch als Zahlenstring genutzt wird. Eine Anpassung ist, dass der Generator, für den Type ID so angepasst wird, dass er nur Argumente für ID zurückliefert, die ein Zahlenstring sind.

8.3 Fehlerfindungskapazitäten

Zuerst soll die Fehlerfindungskapazität des Prototypens bewiesen werden. Dafür werden die beiden zuvor benannten Testsysteme GraphQL-Toy 12 und GitLab in der Version 12.6.3 verwendet. Ziel ist es mindestens die Fehler zu finden, die vom *Property-based Testtool* [6] gefunden wurden.

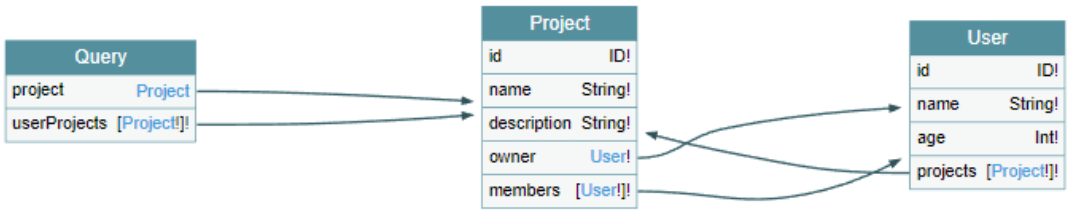

Abbildung 8.1: Graph des GraphQL-Toys

8.3.1 GraphQL-Toy

Das Testsystem GraphQL-Toy hat ein simples Schema, in dem nur drei *OBJECT* Typen existieren. Diese sind *Query*, *Project* und *User*. Das Schema ist in Abbildung 8.1 visualisiert.
Entwickelt wurde dieses System mit dem Hintergrund, dass bekannte Bugs im Code eingebracht werden und überprüft werden kann, ob das Testtool diese findet. Insgesamt wurden 15 verschiedene Bugs eingefügt, welche in verschiedene Kategorien fallen wie Syntaxfehler, falsche Rückgabedaten, falsche Datenstrukturen etc. Einige der Bugs werden im Folgenden kurz vorgestellt. Eine Liste aller eingebauten Bugs findet sich im Anhang unter *GraphQL-Toy Implementation mit Bugs* 12.

Bug 1 - SyntaxFehler

Diverse Syntaxfehler wurden an verschiedenen Stellen eingebaut. Dies bedeutet, dass jeder Funktionsaufruf dieser Funktion garantiert scheitern wird. Somit kann jede Request diesen Fehlerfall entdecken, solange die Request auch das Feld hinter der Funktion mit dem Syntaxfehler abfragt. Ein einfacher Syntaxfehler findet sich in Listing 8.1.

```
1    const resolvers = {
2        Query: {
3            project: (_, {id}, context, info) => {
4                // Example bug 1 - Syntax mistake
5                return db.projects.find(project => project.
                     id ===);
6            }
7        }
8    }
```

Listing 8.1: Ein Syntaxfehler

Hierbei fehlt der Wert, mit dem die *project.id* verglichen werden soll. Jeder Aufruf dieser Funktion mit egal welcher *ID* führt zu einem Fehler.

Bug 2 - Falscher Objekttyp

Objektfehler sind Fehler, bei denen ein falsches Objekt zurückgegeben wird. Das Objekt passt dabei nicht zu der definierten Struktur im Schema. GraphQL wird dann einen Fehler erzeugen, da die Daten nicht zum Schema passen. Code der zu solch einem Fehler führt ist in Listing 8.2 dargestellt.

```
1    const resolvers = {
2        Query: {
3            project: (_, {id}, context, info) => {
4                // Example bug 5 - wrong type "error"
5                return { ...db.projects.find(project =>
                     project.id === id), name: ["a", "b"] };
6            }
7        }
8    }
```

Listing 8.2: Ein Objektfehler

Sollte das Feld ein Argument benötigen, so muss dieses passen, damit auch wirklich ein Objekt abgefragt wird und dann der falsche Type zurückgegeben wird.

Bug 3 - Typfehler in der Eingabe

Felder wie *ID* sind im GraphQL-Standard als einzigartige Strings definiert. Im Allgemeinen wird der *ID* Type jedoch von diversen Entwicklern als Zahlenstring genutzt. Eine Funktion wandelt diesen String dann in eine Zahl, die zum Beispiel genutzt wird, um einen bestimmten Eintrag eines Arrays zu bekommen. Inputvalidierung wird also benötigt. In Listing 8.3 ist Code dargestellt, der zu solch einem Fehler führt.

```
1    const resolvers = {
2        Query: {
3            project: (_, {id}, context, info) => {
4                // Example bug 3 - Input type validation bug
5                return db.projects[id];
6            }
7        }
8    }
```

Listing 8.3: Code ohne Inputvalidierung

Es ist bei diesem Code möglich, ohne jegliche Prüfung einen Key anzugeben. Ist ein Resolver so implementiert, dann ist ein IndexOutOfBound Fehler sehr wahrscheinlich.

Mit dem Testtool nach [6, Property-based Testing] konnten 73% der Fehler, also 11 der 15 Fehler gefunden werden. Der hier entwickelte Prototyp schaffte auf derselben API auch eine Entdeckung von 11 Fehlern. Es konnte also dieselbe Fehlerfindung erreicht werden.

Bemerkenswert ist, dass das Property-based Tool wesentlich mehr Queries benötigte, um eine zufriedenstellende Coverage zu erreichen. Das Property-based Tool benötigte 30 Durchläufe, die jeweils bis zu 100% Edge-Coverage liefen, um alle Fehler zu finden. Im Gegenteil dazu konnte die hier entwickelte Methode mithilfe von nur 2 PrimePfaden eine PrimePath Coverage erreichen. Hierzu wurden für jeden Pfad 5 Testqueries entwickelt. Es war somit möglich, alle 11 Fehler zu finden. Dabei wurden zwei perfekte Queries ermittelt. Beide Queries sind in Listing 8.4 und Listing 8.5 gezeigt.

```
1    { project(id: "2", ) {  id  name  description  owner {
         id  name  age  } } }
```

Listing 8.4: Query 1

```
1    { userProjects(id: "1") { name owner { id name age
       projects { name description id owner { id name age }
       } } } }
```

Listing 8.5: Query2

Mithilfe dieser Queries kann jeder der 11 entdeckten Fehler gefunden werden. Dies liegt auch daran, dass der Argumentgenerator entsprechend angepasst wurde und nur valide IDs produziert hat. So war es sehr wahrscheinlich, dass eine ID, die generiert wird, mindestens in einer der 5 erstellten Queries zur zugrundeliegenden Datenstruktur gepasst hat und eine tatsächliche Testausführung stattfand und nicht nur ein initialer *null*-Wert, der die Query sofort vorzeitig beendet, zurückgegeben wurde. Die 4 nicht gefundenen Fehler sind dieselben Fehler wie diese, die *Property-based Testing* [6, vgl. RQ.2] nicht finden konnte. Es sind die Felder, in denen ein falscher Wert eines Objektes genutzt wurde, um ein anderes Objekt zu erlangen. Hierbei verhindert der Black-Box Ansatz, dass der Fehler gefunden wird da eine leere Rückgabe des Feldes eine valide Antwort ist. Der Black-Box Ansatz limitiert hierbei das Verhalten des Prototypen da dieser nicht genügen Domänenwissen hat um diesen Fehler zu erkennen. In diesem Beispiel hat der Prototyp dieselben Fähigkeiten wie der Prototyp von *Property-based Testing*. Die Ergebnisse der Experimente befinden sich im GitHub.

8.3.2 GitLab

Das Testsystem GitLab wurde schon in Property-based Testing verwendet, um an einem Industriereifen Projekt die Methode zu evaluieren [6, vgl. Experiment]. Die entwickelte Methode soll sich auch an GitLab beweisen. GitLab stellt sowohl eine REST als auch GraphQL-API zur Kommunikation zur Verfügung. Mit GitLab wird ein komplexes Softwareprodukt zur Versionsverwaltung und DevOps-Anwendung getestet. Die Komplexität dieser Software wird deutlich, wenn das GraphQL-Schema von GitLab betrachtet wird, welches in Abbildung 8.2 zu finden ist. Eine hochauflösendere Version ist im GitHub verfügbar. Das Schema ist sehr komplex und stark zyklisch.

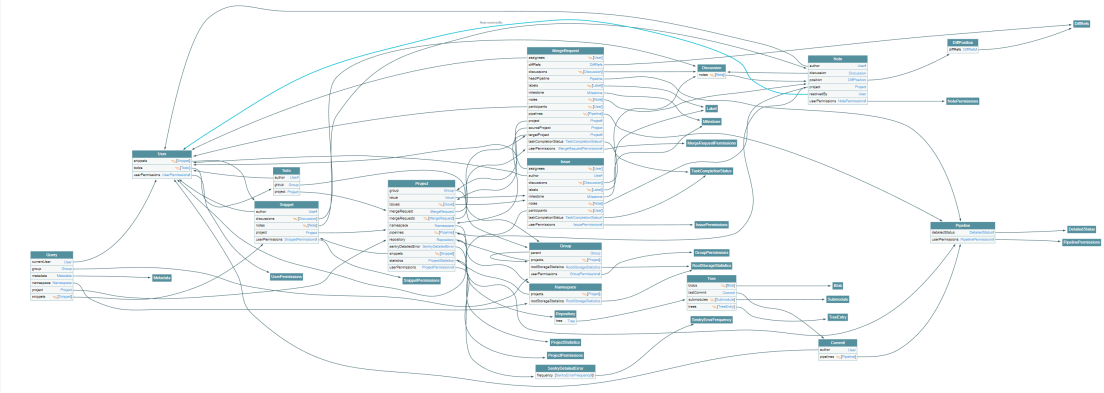

Abbildung 8.2: GitLab GraphQL-Schema

Das *Property-based Testtool* fand insgesamt 4 Fehler, die im Query-Bereich von GraphQL waren. Alle Fehler waren Fehler in der Validierung von Eingabevariablen. Hierbei lag der Fehler darin, dass die Resolver einen Fehler verursachten, wenn als Eingabe ein String mit leerem Zeichen kam. Dies bedeutet, ein leerer String `""` wird richtig behandelt (außer in Fehler 4) aber ein String mit leerem Zeichen `e\u0000\` führt zum Fehler. Die Fehler wurden gefunden durch die Queries aus Listing 8.6, 8.7, 8.8 und 8.9.

```
1 {project(fullPath: "root/test-project") {sentryDetailedError
    (id: "") {count}}}
```

Listing 8.6: Fehler 1[38]

```
1 {project(fullPath: "e\u0000") {name fullPath}}
```

Listing 8.7: Fehler 2[39]

```
1 {namespace(fullPath: "e\u0000") {fullName name fullPath}}
```

Listing 8.8: Fehler 3[40]

```
1 {group(fullPath: "e\u0000"){fullName name fullPath }}
```

Listing 8.9: Fehler 4[41]

Alle vom *Property-based Testtool* gefunden Fehler wurden durch den hier entwickelten Prototypen ebenfalls gefunden. Dafür waren die Queries Query 1, Query 2, Query 3 und Query 4 ausreichend. Es ist zu beobachten, dass die Queries wesentlich länger sind als jene, die vom *Property-based Testtool* erstellt wurden. Dies liegt an den PrimePfaden, welche die längsten einfachen Pfade darstellen. Getestet wurde auf dem offiziellen GitLab-Docker Image in der Version 12.6.3. Damit im GitLab auch Daten verfügbar sind, wurde ein Population-Skript geschrieben, das im GitLab 50 User anlegt und jedem User einige

Projekte, Commit, MergeRequests etc. zuordnet. Das Population-Skript kann im [42, Github] gefunden werden. Da die Querygenerierung stets im Query-Knoten beginnt und die PrimePaths gefunden werden sollen, ergeben sich in diesem speziellen Schema sehr viele ähnliche Queries. Diese unterscheiden sich insbesondere am Ende der jeweiligen Query. Der zuvor vorgestellte Algorithmus errechnet für das Schema von GitLab eine Pfadanzahl von 41744 für eine PrimePath-Coverage des Schemas. Eine genaue Auflistung aller Pfade findet sich im [43, GitHub]. Mit der Maßgabe, dass pro Pfad 5 Tests erzeugt werden sollen, wurden dann 208.720 Tests erzeugt. In nahezu allen Fällen haben die generierten Tests eine tatsächliche Pfadlänge, die kleiner ist als die erwartete. Dies begründet sich damit, dass an verschiedenen Stellen des Schemas von GitLab Argumente angegeben werden müssen und mit jedem Argument, das zusätzlich generiert wird, steigt die Wahrscheinlichkeit, dass die zufällige Kombination unpassend ist. In mehreren Durchläufen zeigte sich, dass im Schnitt nur ungefähr 20 Tests eine ideale Pfadlänge erreichen. Die Tests, die das erreichen, sind im Allgemeinen auch Tests, die einen sehr kurzen Pfad abbilden. Hierbei verringert sich das Risiko, dass Eingabeargumente generiert werden, die keine zugrundeliegenden Daten haben und somit die Pfadausführung verhindern. Ein Beispiel hierfür ist der Pfad $Query \rightarrow Project \rightarrow MergeRequest \rightarrow Time$.

```
{ project(fullPath: "groupx_3/projectx_2_1") {
    archived
    avatarUrl
    containerRegistryEnabled
    ...
    mergeRequest(iid: "1") {
        allowCollaboration
        createdAt
        mergeStatus
        ...
        }
    }
}
```

Durch Anpassen des *FullPath* Argumentgenerators war es möglich, eine Query zu generieren, die passende Argumente generiert hat, um für eine ideale Testabdeckung (d.h. Länge Ergebnis = Länge Testpfad) zu sorgen. Generell zeigt sich sehr schnell, dass bei der hier entwickelten Methode ähnliche Limitierungen wie im Property-based Testing auftreten. So ist eine manuelle Anpassung der Argumentgeneratoren an das Domänenwissen nötig. Für GitLab bedeutet dies unter anderem, dass die ID-Struktur, um zum Beispiel Projekte abzufragen, in der Form `<user>/<project>` sein muss [6, vgl. S.8]. Da hier ähnliche Argumentgeneratoren verwendet wurden wie in Property-based Testing, leiden diese unter derselben Limitierung des mangelnden Domänenwissens [6, vgl. S.8]. Es werden zwar sehr viele Tests entwickelt, die die PrimePfad Abdeckung umsetzen. Wenn jedoch die Generierung der Tests auf Zufall basiert, so kann nicht garantiert werden, dass die Eingabeargumente passend sind und ein valides Ergebnis zurückgeben. Wird kein Ergebnis zurückgegeben, so folgert GraphQL, dass spätere Funktionen nicht aus-

geführt werden müssen und somit werden diese auch nicht getestet. Es ist möglich dieses Problem zu beheben. Dazu jedoch in Kapitel 9 mehr.

8.4 Schema-Abdeckung

Wie in Property-based Testing schon erwähnt: *da keine Coverage Metric für GraphQL Blackbox Test Auswertung existiert, starten wir mit einem sehr einfachen und intuitiven Ansatz* [6, vgl. B. Measuring Schema Coverage]. Das in *Property-based Testing* vorgestellte Abdeckungskriterium ist die Kantenabdeckung, wie schon in Abschnitt 8.1.3 gezeigt. Da jedoch das Rekursionslimit die Pfadlänge begrenzt, wird der Graph des Schemas künstlich beschnitten und alle Pfade, die länger als das Rekursionslimit sind, werden in der Abdeckung nicht berücksichtigt. Hierdurch folgt, dass sich die gewünschte Kantenabdeckung in Realität nicht zuverlässig ergibt, da GraphQL-Schemas durchaus Pfadlängen länger als das Rekursionslimit zulassen (das Rekursionslimit wurde standardmäßig auf 4 gesetzt [6, vgl. SourceCode]). Um die gewünschte Kantenabdeckung zu erreichen, musste im *Property-based Testing* außerdem die Generierung mehrfach ausgeführt werden, bis die Abdeckung erreicht wurde. Um eine vollständige Abdeckung beim GitLab Schema zu erreichen, waren diverse Iterationen nötig bei verschiedenen Rekursionslimits. Eine 100% Coverage wurde bei GitLab nur in einem Versuch erreicht, wenn 10000 Tests mit Rekursionslimit 4 erstellt wurden [6, vgl. Tabelle 2]. Da der Graph des GitLab-Schemas jedoch beschnitten wurde, da dieser, wie später gezeigt wird, wesentlich größer ist, kann nicht gesagt werden, dass vollständige Kantenabdeckung erreicht wurde. Der wesentliche Unterschied beider Methoden ist, dass *Property-based Testing* Experimente für die theoretische Abdeckung ausführen muss und hierbei werden mehrere Iterationen benötigt, um diese zu erreichen. Der Ansatz der PrimePfad-Abdeckung im hier entwickelten Prototypen stellt sicher, dass die generierten Pfade PrimePfade sind. Somit ist zugesichert, dass die Abdeckung erfüllt ist. Die PrimePfad-Abdeckung wird auf unbeschnittenen Graphen ausgeführt. Es kann also gefolgert werden, dass die hier entwickelte Methode eine Verbesserung der theoretischen Abdeckung erzielt hat. Ein praktischer Nachweis hierfür folgt in den nächsten Abschnitten. Die Limitierung der zufälligen Argumentgeneratoren behindert eine tatsächliche Umsetzung der theoretischen Abdeckung. Allerdings ist diese Limitierung potenziell lösbar, hierzu in Kapitel 9 mehr.

8.4.1 GraphQL-Toy Schema Coverage

Wie eingeführt in *Property-based Testing* [6] muss für eine zufriedenstellende Abdeckung die Definition 18 erfüllt sein, dass jedes Paar von `(Type, objectField)` berücksichtigt ist. Dies bedeutet für das Schema, dass die folgenden Tupel abgedeckt sein müssen, um Kantenabdeckung zu erfüllen.

- `(Query, project)`
- `(Query, userProject)`
- `(Project, owner)`
- `(Project, members)`
- `(User, projects)`

Da nur die beiden initialen Felder aus dem Query-Type Eingabeargumente benötigen, ist die Querygenerierung simpel. Es kann keine Aussage darüber gemacht werden, ob die generierten Queries von *Property-based Testing* [6] diese Abdeckung erfüllen, denn bei der Querygenerierung spielt es keine Rolle, ob dieses Kriterium erreicht wird. Es gibt lediglich eine Messung die zeigt, dass der Prototyp mit hinreichender Wahrscheinlichkeit in der Lage ist, durch zufällige Querygenerierung Tests zu generieren, die Kantenabdeckung erfüllen [6, vgl. D.Results RQ1]. Im Gegensatz zum Property-based Testing hat der hier entwickelte Prototyp den Vorteil, dass die Pfadgenerierung nicht zufällig ist. Dadurch ergibt sich, dass die Tests, die vom hier entwickelten Prototyp aufgrund eines stärkeren Abdeckungskriteriums erstellt werden, stets auch die Kantenabdeckung erfüllen. Ein einziger Durchlauf reicht aus, um sicherzustellen, dass die gewünschte Abdeckung zumindest theoretisch erreicht ist. Natürlich bleibt offen, ob die generierten Tests diese Coverage tatsächlich erreichen jedoch ist dies auch ein Problem im *Property-based Testing*. Dort wird nur geprüft ob die Felder in der Anfrage existieren, jedoch nicht, ob die Antwort diese auch enthält. Um dies messbar zu machen wurde zuvor die Abschätzung der Pfadlängen eingeführt. Hierdurch konnte gezeigt werden, dass die generierten Queries in Teilen die Abdeckung auch tatsächlich umsetzen.

8.4.2 GitLab Schema Coverage

Das GitLab Schema ist wesentlich komplexer und zyklischer als das GraphQL-Toy. Im Gegensatz zum GraphQL-Toy besteht das GitLab Schema aus 37 Knoten, welche jeweils zahlreiche Kanten hinzufügen. Generell lässt sich sagen, dass das Schema sehr komplex und stark rekursiv ist [6, vgl. Studied Cases 2]. Da der Property-based Testing Ansatz ein Rekursionslimit benötigt stellt, sich die Frage inwiefern überhaupt dieses Schema überdeckt werden kann. Laut Paper hat sich ein Rekursionslimit von vier als hinreichend ergeben [6, vgl. Table 1] und wurde auch so im Code übernommen. Ein Rekusionslimit von vier bedeutet, dass die maximale zu erreichende Pfadlänge des Testpfades vier ist. Da das GitLab-Schema aber einen Graphen aufspannt, der durchaus wesentlich längere

Pfade als vier hat, ist es fragwürdig wie die 100% Schema-Coverage in [6, Table 1] berechnet wurde. Es seien hier einige Pfade beispielhaft genannt, die einzigartig sind, bei denen sich keine Kante doppelt und deren Länge 4 stark überschreitet:

- Query →User →SnippetConnection →SnippetEdge →Snippet →DiscussionConnection →DiscussionEdge →Discussion →NoteConnection →NoteEdge →Note →Project →IssueConnection →Issue →Milestone →Time
- Query →Project →Issue →DiscussionConnection →DiscussionEdge →Discussion →NoteConnection →NoteEdge →Note →User →SnippetConnection →SnippetEdge →Snippet →Time
- Query →Namespace →ProjectConnection →Project →MergeRequestConnection →MergeRequestEdge →MergeRequest →UserConnection →User →SnippetConnection →Snippet →DiscussionConnection →PageInfo

Durch die Beschneidung des Graphens in *Property-based Testing* durch das Rekursionslimit folgt, dass die ermittelte Kantenabdeckung nicht zutreffend ist und nur für einen Teilgraphen des gesamten Graphens zutrifft. Dies ist ein sehr großer, struktureller Einschnitt und die in Property-based Testing genannten 100% Kantenabdeckung ist keine tatsächliche Kantenabdeckung, sondern eben nur für den abgeschnittenen Teilgraphen. Wie zuvor gezeigt, erzeugt der hier entwickelte Prototyp Tests, die PrimePfad-Abdeckung umsetzen und somit ein stärkeres Abdeckungskriterium erfüllen. Eine komplette Pfadgenerierung auf dem gesamten Graphen hatte als Ergebnis, dass über 40.000 Pfade benötigt werden, um eine PrimePfad-Abdeckung für das GitLab-Schema zu erreichen. Hier zeigt sich auch ein direkter Unterschied. Während in Property-based Testing gesagt wird, dass 10.000 Tests mit einem Rekursionslimit von 4 ausreichen, um eine vollständige Kantenabdeckung zu erreichen [6, vgl. Table 1] so wurde gezeigt, dass 10.000 Tests nicht ausreichen können, wenn allein über 40.000 PrimePfade existieren. Die Berechnung der Queries geschah auf einem hardwaretechnisch ähnlichem Level wie in Property-based Testing angewandt [6, vgl. Experimental Setup]. Hier wurde die Aussage getroffen, dass ein *Tiefensuchen Ansatz nicht skaliert und deswegen ein iterativer Ansatz zu präferieren ist* [6]. Der hier entwickelte Prototyp zeigt das Gegenteil.

8.5 Zusammenfassung der Experimente

Mit den beiden Experimenten konnte gezeigt werden, dass der Prototyp dieselben Fehler findet wie der Prototyp aus Property-based Testing. Es wurde gezeigt, dass die theoretische Abdeckung eines GraphQL-Schemas mit dem Prototypen immens gesteigert wird ohne, wie in [6] behauptet, die Berechnungszeit signifikant zu erhöhen. Während die theoretische Abdeckung des Graphens erhöht wurde, so zeigte sich, dass die reale Abdeckung

nicht mit der theoretischen Abdeckung mithalten kann. Die Pfadlänge der generierten PrimePfade ist im Allgemeinen sehr lang und hierdurch steigt die Wahrscheinlichkeit, dass ein Pfad nicht komplett ausgeführt wird. Eine Anpassung der Argumentgeneratoren an das jeweilige SUT ist sinnvoll und verbessert die Pfadlängen der Tests. Das mangelnde Domänenwissen limitiert die Resolver in derselben Weise wie im *Property-based Testing* wo es heißt: *.. das Domänenwissen von zugrundeliegenden Entitäten eine stärkere Testumgebung erzeugen kann* [6, S.8]. Möglichkeiten, um das Domänenwissen über das SUT zu erhöhen, werden in Kapitel 9 vorgestellt.

9 Zukünftige Arbeit

Einige bereits angesprochene Punkte bieten Möglichkeiten für eine Erweiterung der hier geleisteten Arbeit. Während in dieser Arbeit ein erheblicher Beitrag zur theoretischen Testabdeckung von GraphQL-APIs geleistet wurde, so ist nicht garantiert, dass die entwickelten Tests tatsächlich die ermittelte Abdeckung erreichen. Dies folgert sich aus der zufälligen Argumentgenerierung in den einzelnen Queries. Ziel ist es nun, den Zufall möglichst weit zu begrenzen oder aber die erlangten Ergebnisse sinnvoller zu nutzen. Im Folgenden werden zwei Ansätze vorgestellt, die das Potenzial haben, eine praktische Testausführung zuverlässiger und präziser machen zu können. Ziel von weiterführenden Arbeiten sollte es sein, dass die tatsächliche Pfadabdeckung sich der theoretischen Pfadabdeckung annähert.

9.1 BlackBox-Testing in WhiteBox-Testing umwandeln

Das Testsystem hat im BlackBox-Testing keinerlei Informationen über das SUT und Testgenerierung auf GraphQL-APIs ohne die Argumentgeneratoren anzupassen, führt häufig dazu, dass die generierten Daten nicht zum SUT passen. Im experimentellen Ansatz wurde der BlackBox Ansatz abgeschwächt und zu einem GreyBox-Ansatz verändert, indem die Argumentgeneratoren mit Domänenwissen an das jeweilige SUT angepasst wurden, sodass die zufällige Argumentgenerierung mit höherer Wahrscheinlichkeit ein Argument liefert, das dem Test eine bessere, tatsächliche Abdeckung liefert. Ideal wäre nun, dass die Testgenerierung auf einem WhiteBox-Ansatz basiert. Hierdurch ist spezifisches Domänenwissen über das SUT vorhanden, insbesondere dadurch, dass die zugrundeliegenden Daten, Datenstruktur, Programmcode analysierbar sind. Durch einen White-Box Ansatz wäre es möglich, die Argumentgeneratoren automatisiert anzupassen, sodass sich diese am Schema und den zugrunde-liegenden Daten orientieren. Außerdem wäre eine Code-Analyse möglich, die dazu führen kann, dass Testfälle noch präziser und exakter Fehler finden. Die vom Prototypen generierten Testpfade können hierbei weiterhin als Basis dienen, um eine gute Abdeckung sicherzustellen. Mit optimierten Argumentgeneratoren sollte es möglich sein, die tatsächliche Pfadabdeckung stark zu erhöhen.

9.2 Adaptive Generierung

Die aktuelle Testgenerierung geschieht in einzelnen Phasen. Es werden erst aus einem Graphen die Pfade generiert, hieraus werden Tests erzeugt und diese werden dann an das SUT gestellt und ausgewertet. Dabei werden sämtliche Argumente sofort generiert.

Eine adaptive Testgenerierung wäre denkbar, sodass aus einem Testpfad die Query nur in Teilen erstellt wird. Dabei wird ein Testpfad erst weiter in der Query abgedeckt, wenn ein Argumentgenerator ein Ergebnis zurückliefert, dass die Pfadlänge tatsächlich erhöht und somit signalisiert, dass der zugrundeliegende Test tatsächlich ausgeführt wird. Eine solche Methode könnte nach dem Ablauf, wie in Abbildung 9.1 dargestellt, funktionieren. Es werden dabei Queries mit Zufallsargumenten erstellt, die einen Teilpfad des Testpfads bilden und erst wenn der Teilpfad des Testpfads tatsächlich abgedeckt ist, wird mit dem nächsten adjazenten Knoten fortgefahren, sodass die tatsächliche Pfadabdeckung mit der theoretischen Pfadabdeckung übereinstimmt.

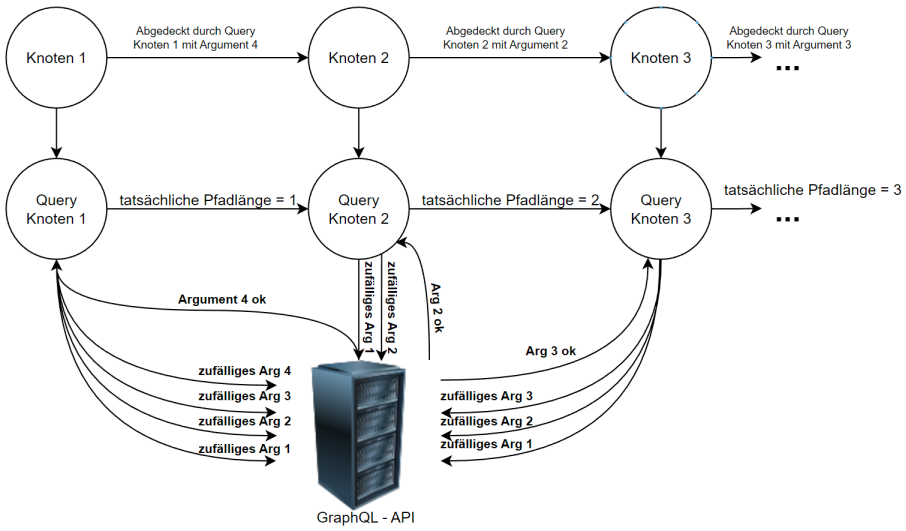

Abbildung 9.1: Beispielablauf einer adaptiven Generierung

Mögliche Limitierungen wären hierbei jedoch, dass ein Pfad niemals seine gewünschte Pfadlänge erreicht, weil zum Beispiel Daten für den Pfad fehlen und somit jedes Argument unzureichend ist. Denkbar wäre auch, dass Argumentgeneratoren nicht in der Lage sind, passende Daten zu erzeugen. Es wären Strategien zu entwickeln, die sicherstellen, dass solche Limitierungen korrekt behandelt werden. Mit einer solchen Query-Generierung ist eine Steigerung der tatsächlichen Pfadabdeckung möglich und gleichzeitig kann das erlangte Wissen in anderen Pfaden genutzt werden, um die Argumentgenerierung zu vereinfachen.

10 Fazit

Die Behauptung, dass die Methode des *Property-based Testings* um einen besseren Pfadfindungsalgorithmus als ein simples, zufälliges, begrenztes Raten verbessert werden kann, hat sich als wahr herausgestellt. Indem ein theoretischer Rahmen geschaffen wurde, war es möglich, Graphabdeckungskriterien zu nutzen, um Tests für GraphQL automatisiert zu generieren. Die entwickelten Methoden liefern eine gute Grundlage, um das automatisierte Testen von GraphQL weiter voranzutreiben. Final entstand dann daraus ein Prototyp, der fähig ist, Fehler in GraphQL-APIs zu finden. Dies wurde an zwei Beispielen gezeigt und nachgewiesen, indem Fehler gefunden werden konnten. Es wurden Schwachstellen offengelegt und Ansätze entwickelt, die zur Weiterarbeit anregen. Verwandte Arbeiten wurden betrachtet und dabei konnte festgestellt werden, dass ein Ansatz wie in dieser Arbeit vorher noch nicht betrachtet wurde. Generell lässt sich sagen, dass GraphQL eine berechtigterweise stetig wachsende Technologie ist und Arbeiten wie diese dazu beitragen, dass die Popularität von GraphQL wächst, da die bisherigen Nachteile von GraphQL gegenüber REST 3.3 so stückweise aufgelöst werden und die großen Vorteile erkennbar werden.

11 Glossar

Im Text werden einige Fachbegriffe genutzt. Hier findet sich deren Erklärung

Begriff Erklärung

IEEE/ACM Ein Journal über Transaktionen in Netzwerken das regelmäßig Konferenzen veranstaltet

HTTP HyperTextTransferProtocoll ist ein Übertragunsprotokoll für Datenübertragung

HTTP-Request Ist eine Anfrage zum zusenden von Daten

API Application Programmable Interface ist eine Schnittstelle wie Systeme miteinander kommunizieren

REST Ein Architekturdesign für APIs

GraphQL Eine Abfragesprache für APIs die den GraphQL Standard implementieren

Overfetch Das Abfrgaen von zu vielen Informationen

Underfetch Das Abfragen von zu wenigen Informationen

SUT System under Test - ist eine Kurzform für das System, dass es zu testen gilt

IoT Internet of Things - meint die Verknüpfung von diversen Geräten mit dem Internet

12 Anhang

GraphQL-Toy Implementation mit Bugs

```
1
2  const {ApolloServer, gql, ApolloError} = require('apollo-
       server');
3
4  const typeDefs = gql`
5      type User {
6          id: ID!
7          name: String!
8          age: Int!
9          projects: [Project!]!
10     }
11
12     type Project {
13         id: ID!
14         name: String!
15         description: String!
16         owner: User!
17         members: [User!]!
18     }
19
20     type Query {
21         project(id: ID!): Project
22         userProjects(id: ID!): [Project!]!
23     }
24 `;
25
26 const db = {
27     projects: [
28         {
29             id: "1",
30             name: "Project 1",
31             description: "Awesome project!",
32             owner: "100",
33             members: ["100", "200"],
```

```
        },
        {
            id: "2",
            name: "Project 2",
            description: "Not an awesome project!",
            owner: "200",
            members: ["200"],
        },
    ],
    users: [
        {
            id: "100",
            name: "Burt",
            age: 23,
            projects: ["1", "2"],
        },
        {
            id: "200",
            name: "Earnie",
            age: 32,
            projects: ["2"],
        },
    ],
};

const resolvers = {
    Query: {
        project: (_, {id}, context, info) => {

            // Example bug 1 - Syntax mistake
            // return db.projects.find(project => project.id
                ===);

            // Example bug 2 - Give "foo", input validation
            // return db.projects[parseInt(id)];

            // Example bug 3 - Input type validation bug
            //return db.projects[id];

            // Example bug 4 - Using the wrong field
            // return db.projects.find(project => project.
               name === id);

            // Example bug 5 - wrong type "error"
```

```
76              // return { ...db.projects.find(project =>
                   project.id === id), name: ["a", "b"] };
77
78              // Example bug 6 - IndexOutOfBounds
79              // return db.projects[parseInt(id)];
80
81              // Correct implementation
82              return db.projects.find(project => project.id
                   === id);
83          },
84          userProjects: (_, {id}, context, info) => {
85              const user = db.users.find(user => user.id ===
                   id);
86
87              // Example bug 1 - Syntax Error
88              // return db.projects.filter(project => user.
                   projects.includes());
89
90              // Example bug 2 - Using the wrong field
91              // return db.projects.filter(project => user.
                   projects.includes(project.name));
92
93              // Example bug 3 - wrong type "errors"
94              // return db.projects.filter(project => user.
                   projects.includes);
95
96              // Correct implementation
97              return db.projects.filter(project => user.
                   projects.includes(project.id));
98          },
99      },
100     Project: {
101         owner: (project) => {
102             // Example bug 1 - Syntax mistake
103             // return db.users.find(user => user.id ===);
104
105             // Example bug 2 - Using the wrong field
106             // return db.users.find(user => user.name ===
                   project.owner);
107
108             // Example bug 3 - wrong type "error"
109             // return { ...db.users.find(user => user.id ===
                    project.owner), name: ["a", "b"] };
110
```

```
111             // Correct implementation
112             return db.users.find(user => user.id === project
                    .owner);
113         },
114         members: (project) => {
115             // Example bug 1 - logic error
116             // return db.users.filter(user => project.
                    members.includes());
117 
118             // Example bug 2 - Using the wrong field
119             // return db.users.filter(user => project.
                    members.includes(user.name));
120 
121             // Example bug 3 - wrong type "errors"
122             // return db.users.filter(user => project.
                    members.includes);
123 
124             // Correct implementation
125             return db.users.filter(user => project.members.
                    includes(user.id));
126         },
127     },
128     User: {
129         projects: (user) => {
130             // Example bug 1 - Syntax Error
131             // return db.projects.filter(project => user.
                    projects.includes());
132 
133             // Example bug 2 - Using the wrong field
134             return db.projects.filter(project => user.
                    projects.includes(project.name));
135 
136             // Example bug 3 - wrong type "errors"
137             // return db.projects.filter(project => user.
                    projects.includes);
138 
139             // Correct implementation
140             return db.projects.filter(project => user.
                    projects.includes(project.id));
141         },
142     },
143 };
144 
145 const server = new ApolloServer({typeDefs, resolvers});
```

```
146
147 server.listen().then(({url}) => {
148     console.log(`Server ready at ${url}`);
149 });
```

Introspection-Query

```
1     query IntrospectionQuery {
2   __schema {
3     queryType {
4       name
5     }
6     mutationType {
7       name
8     }
9     subscriptionType {
10      name
11    }
12    types {
13      ...FullType
14    }
15    directives {
16      name
17      description
18      locations
19      args {
20        ...InputValue
21      }
22    }
23  }
24 }
25
26 fragment FullType on __Type {
27   kind
28   name
29   description
30   fields(includeDeprecated: true) {
31     name
32     description
33     args {
34       ...InputValue
35     }
```

```
    type {
      ...TypeRef
    }
    isDeprecated
    deprecationReason
  }
  inputFields {
    ...InputValue
  }
  interfaces {
    ...TypeRef
  }
  enumValues(includeDeprecated: true) {
    name
    description
    isDeprecated
    deprecationReason
  }
  possibleTypes {
    ...TypeRef
  }
}

fragment InputValue on __InputValue {
  name
  description
  type {
    ...TypeRef
  }
  defaultValue
}

fragment TypeRef on __Type {
  kind
  name
  ofType {
    kind
    name
    ofType {
      kind
      name
      ofType {
        kind
        name
```

```
        ofType {
          kind
          name
          ofType {
            kind
            name
            ofType {
              kind
              name
              ofType {
                kind
                name
              }
            }
          }
        }
      }
    }
  }
}
```

minimale Schema Response

```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      },
      "mutationType": null,
      "subscriptionType": null,
      "types": [
        {
          "kind": "OBJECT",
          "name": "Query",
          "description": null,
          "fields": [
            {
              "name": "book",
              "description": null,
              "args": [
                {
```

```
                    "name": "id",
                    "description": null,
                    "type": {
                      "kind": "SCALAR",
                      "name": "ID",
                      "ofType": null
                    },
                    "defaultValue": null
                  }
                ],
                "type": {
                  "kind": "OBJECT",
                  "name": "Book",
                  "ofType": null
                },
                "isDeprecated": false,
                "deprecationReason": null
              },
              {
                "name": "author",
                "description": null,
                "args": [
                  {
                    "name": "id",
                    "description": null,
                    "type": {
                      "kind": "SCALAR",
                      "name": "ID",
                      "ofType": null
                    },
                    "defaultValue": null
                  }
                ],
                "type": {
                  "kind": "OBJECT",
                  "name": "Author",
                  "ofType": null
                },
                "isDeprecated": false,
                "deprecationReason": null
              },
              {
                "name": "publisher",
                "description": null,
```

```
            "args": [
              {
                "name": "id",
                "description": null,
                "type": {
                  "kind": "SCALAR",
                  "name": "ID",
                  "ofType": null
                },
                "defaultValue": null
              }
            ],
            "type": {
              "kind": "OBJECT",
              "name": "Publisher",
              "ofType": null
            },
            "isDeprecated": false,
            "deprecationReason": null
          }
        ],
        "inputFields": null,
        "interfaces": [],
        "enumValues": null,
        "possibleTypes": null
      },
      {
        "kind": "OBJECT",
        "name": "Book",
        "description": null,
        "fields": [
          {
            "name": "id",
            "description": null,
            "args": [],
            "type": {
              "kind": "SCALAR",
              "name": "ID",
              "ofType": null
            },
            "isDeprecated": false,
            "deprecationReason": null
          },
          {
```

```
              "name": "title",
              "description": null,
              "args": [],
              "type": {
                "kind": "SCALAR",
                "name": "String",
                "ofType": null
              },
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "author",
              "description": null,
              "args": [],
              "type": {
                "kind": "OBJECT",
                "name": "Author",
                "ofType": null
              },
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "publisher",
              "description": null,
              "args": [],
              "type": {
                "kind": "OBJECT",
                "name": "Publisher",
                "ofType": null
              },
              "isDeprecated": false,
              "deprecationReason": null
            }
          ],
          "inputFields": null,
          "interfaces": [],
          "enumValues": null,
          "possibleTypes": null
        },
        {
          "kind": "OBJECT",
          "name": "Author",
```

```
          "description": null,
          "fields": [
            {
              "name": "id",
              "description": null,
              "args": [],
              "type": {
                "kind": "SCALAR",
                "name": "ID",
                "ofType": null
              },
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "name",
              "description": null,
              "args": [],
              "type": {
                "kind": "SCALAR",
                "name": "String",
                "ofType": null
              },
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "books",
              "description": null,
              "args": [],
              "type": {
                "kind": "LIST",
                "name": null,
                "ofType": {
                  "kind": "OBJECT",
                  "name": "Book",
                  "ofType": null
                }
              },
              "isDeprecated": false,
              "deprecationReason": null
            }
          ],
          "inputFields": null,
```

```
            "interfaces": [],
            "enumValues": null,
            "possibleTypes": null
          },
          {
            "kind": "OBJECT",
            "name": "Publisher",
            "description": null,
            "fields": [
              {
                "name": "id",
                "description": null,
                "args": [],
                "type": {
                  "kind": "SCALAR",
                  "name": "ID",
                  "ofType": null
                },
                "isDeprecated": false,
                "deprecationReason": null
              },
              {
                "name": "name",
                "description": null,
                "args": [],
                "type": {
                  "kind": "SCALAR",
                  "name": "String",
                  "ofType": null
                },
                "isDeprecated": false,
                "deprecationReason": null
              },
              {
                "name": "books",
                "description": null,
                "args": [],
                "type": {
                  "kind": "LIST",
                  "name": null,
                  "ofType": {
                    "kind": "OBJECT",
                    "name": "Book",
                    "ofType": null
```

```
                }
              },
              "isDeprecated": false,
              "deprecationReason": null
            }
          ],
          "inputFields": null,
          "interfaces": [],
          "enumValues": null,
          "possibleTypes": null
        }

      ]
    }
  }
}
```

Query1

```
{
  group(fullPath: "e\u0000") {
    avatarUrl
    description
    descriptionHtml
    fullName
    fullPath
    id
    lfsEnabled
    name
    path
    requestAccessEnabled
    visibility
    webUrl
    projects(
      includeSubgroups: false,
      after: "CXPnWOYLTuOjSbbwJqqY",
      before: "CrGVlZBseDurRlgzEbtU",
      first: 2522,
      last: 3011
    ) {
      nodes {
        archived
```

```
        avatarUrl
        containerRegistryEnabled
        createdAt
        description
        descriptionHtml
        forksCount
        fullPath
        httpUrlToRepo
        id
        importStatus
        issuesEnabled
        jobsEnabled
        lastActivityAt
        lfsEnabled
        mergeRequestsEnabled
        mergeRequestsFfOnlyEnabled
        name
        nameWithNamespace
        onlyAllowMergeIfAllDiscussionsAreResolved
        onlyAllowMergeIfPipelineSucceeds
        openIssuesCount
        path
        printingMergeRequestLinkEnabled
        publicJobs
        removeSourceBranchAfterMerge
        requestAccessEnabled
        sharedRunnersEnabled
        snippetsEnabled
        sshUrlToRepo
        starCount
        tagList
        visibility
        webUrl
        wikiEnabled
        issues(
          iid: "ijrhRHNqHyAjlpaJknYi",
          iids: "oSTpjUyfHvXKPFvrnNAK",
          state: closed,
          labelName: "NGJwsiFoOWbPIDPCEOPS",
          createdBefore: "2023-07-19T22:26:43",
          createdAfter: "2023-04-15T21:00:50",
          updatedBefore: "2023-07-10T11:48:12",
          updatedAfter: "2023-02-24T21:34:52",
          closedBefore: "2023-04-23T17:41:34",
```

```
          closedAfter: "2023-01-28T01:02:27",
          search: "ZnbnsbggmMpWkRkQfZDT",
          sort: DUE_DATE_DESC,
          after: "onPOzYJlSSdeVODfTFZd",
          before: "crsihXiPwGbeXvUzAHCM",
          first: 7038,
          last: 5497
        ) {
          nodes {
            closedAt
            confidential
            createdAt
            description
            descriptionHtml
            discussionLocked
            downvotes
            dueDate
            iid
            reference
            relativePosition
            subscribed
            timeEstimate
            title
            titleHtml
            totalTimeSpent
            updatedAt
            upvotes
            userNotesCount
            webPath
            webUrl
            notes(
              after: "EdfPOYFMhnmaRwXCdIXk",
              before: "wMdIUdZSYCYGUHShEdSY",
              first: 7363,
              last: 8871
            ) {
              edges {
                cursor
                node {
                  body
                  bodyHtml
                  createdAt
                  id
                  resolvable
```

```
                    resolvedAt
                    system
                    updatedAt
                    createdAt
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Query2

```
{
  namespace(fullPath: "e\u0000") {
    description
    descriptionHtml
    fullName
    fullPath
    id
    lfsEnabled
    name
    path
    requestAccessEnabled
    visibility
    projects(
      includeSubgroups: true,
      after: "LzGUwINJWDSdeHYamsoy",
      before: "VdRsarUsfXODNvMdrBWx",
      first: 6482,
      last: 3087
    ) {
      nodes {
        archived
        avatarUrl
        containerRegistryEnabled
        createdAt
        description
        descriptionHtml
```

```
        forksCount
        fullPath
        httpUrlToRepo
        id
        importStatus
        issuesEnabled
        jobsEnabled
        lastActivityAt
        lfsEnabled
        mergeRequestsEnabled
        mergeRequestsFfOnlyEnabled
        name
        nameWithNamespace
        onlyAllowMergeIfAllDiscussionsAreResolved
        onlyAllowMergeIfPipelineSucceeds
        openIssuesCount
        path
        printingMergeRequestLinkEnabled
        publicJobs
        removeSourceBranchAfterMerge
        requestAccessEnabled
        sharedRunnersEnabled
        snippetsEnabled
        sshUrlToRepo
        starCount
        tagList
        visibility
        webUrl
        wikiEnabled
        repository {
          empty
          exists
          rootRef
          tree(
            path: "XNQtaUctgSGfziijhXQv",
            ref: "zCYbJRMCKGotQJmpBcZa",
            recursive: true
          ) {
            lastCommit {
              authorName
              authoredDate
              description
              id
              message
```

```
                sha
                signatureHtml
                title
                webUrl
                author {
                  avatarUrl
                  name
                  username
                  webUrl
                  snippets(
                    ids: ["e\u0000", "", "TEST", "2"],
                    visibility: private,
                    type: project,
                    after: "gVweJjzTOptsXXmrvTtA",
                    before: "uSKONOPlLKFpiKNhBIyN",
                    first: 3982,
                    last: 2317
                  ) {
                    pageInfo {
                      endCursor
                      hasNextPage
                      hasPreviousPage
                      startCursor
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Query 3

```
{
  project(fullPath: "e\u0000") {
    archived
    avatarUrl
    containerRegistryEnabled
    createdAt
```

```
    description
    descriptionHtml
    forksCount
    fullPath
    httpUrlToRepo
    id
    importStatus
    issuesEnabled
    jobsEnabled
    lastActivityAt
    lfsEnabled
    mergeRequestsEnabled
    mergeRequestsFfOnlyEnabled
    name
    nameWithNamespace
    onlyAllowMergeIfAllDiscussionsAreResolved
    onlyAllowMergeIfPipelineSucceeds
    openIssuesCount
    path
    printingMergeRequestLinkEnabled
    publicJobs
    removeSourceBranchAfterMerge
    requestAccessEnabled
    sharedRunnersEnabled
    snippetsEnabled
    sshUrlToRepo
    starCount
    tagList
    visibility
    webUrl
    wikiEnabled
    snippets(
      ids: ["e\u0000", "e\u0000", "e\u0000", "2"],
      visibility: private,
      after: "GCGqcYIymVfIjZJrghZx",
      before: "jbZllxYHDwZVZKgtdtBQ",
      first: 7612,
      last: 3392
    ) {
      edges {
        cursor
        node {
          content
          createdAt
```

```
            description
            descriptionHtml
            fileName
            id
            rawUrl
            title
            updatedAt
            webUrl
            discussions(
              after: "XwwfgUTrYgPYJwgMBmmb",
              before: "oUeZWQMPwSveCHQrBEts",
              first: 8210,
              last: 974
            ) {
              edges {
                cursor
                node {
                  createdAt
                  id
                  replyId
                  notes(
                    after: "GfzdyMEZcMMPkByUduYi",
                    before: "MMqCqCBXDTYiNgFgERvS",
                    first: 95,
                    last: 5742
                  ) {
                    nodes {
                      body
                      bodyHtml
                      createdAt
                      id
                      resolvable
                      resolvedAt
                      system
                      updatedAt
                      position {
                        filePath
                        height
                        newLine
                        newPath
                        oldLine
                        oldPath
                        width
                        x
```

```
                        y
                        positionType
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Query 4

```
{
  project(fullPath: "groupx_3/projectx_32_6") {
    archived
    avatarUrl
    containerRegistryEnabled
    createdAt
    description
    descriptionHtml
    forksCount
    fullPath
    httpUrlToRepo
    id
    importStatus
    issuesEnabled
    jobsEnabled
    lastActivityAt
    lfsEnabled
    mergeRequestsEnabled
    mergeRequestsFfOnlyEnabled
    name
    nameWithNamespace
    onlyAllowMergeIfAllDiscussionsAreResolved
    onlyAllowMergeIfPipelineSucceeds
    openIssuesCount
    path
    printingMergeRequestLinkEnabled
```

```
    publicJobs
    removeSourceBranchAfterMerge
    requestAccessEnabled
    sharedRunnersEnabled
    snippetsEnabled
    sshUrlToRepo
    starCount
    tagList
    visibility
    webUrl
    wikiEnabled
    issues(
      iid: "ksXuZMlxdVxzlAaqAjrf",
      iids: "NNAPRcqvnZwDucszkDnh",
      state: locked,
      labelName: "UfeLauqqxLVxuylCFelM",
      createdBefore: "2023-06-30T19:05:54",
      createdAfter: "2023-07-10T04:47:34",
      updatedBefore: "2023-05-14T23:47:22",
      updatedAfter: "2022-09-13T21:09:10",
      closedBefore: "2023-04-28T02:51:59",
      closedAfter: "2022-09-21T20:23:50",
      search: "vAlPWbLSJmlVURpSjmwp",
      sort: created_asc,
      after: "ZuiCQglYmENZJKKScvzv",
      before: "CKMgjxLkrigiXrEPsCPO",
      first: 1092,
      last: 7980
    ) {
      nodes {
        closedAt
        confidential
        createdAt
        description
        descriptionHtml
        discussionLocked
        downvotes
        dueDate
        iid
        reference
        relativePosition
        subscribed
        timeEstimate
        title
```

```
        titleHtml
        totalTimeSpent
        updatedAt
        upvotes
        userNotesCount
        webPath
        webUrl
        discussions(
          after: "dDaOgakGAttuToCxHVCh",
          before: "GcPWhODVHIJXhRgyVMIo",
          first: 326,
          last: 1256
        ) {
          edges {
            cursor
            node {
              createdAt
              id
              replyId
              notes(
                after: "OKLaEyaCRXZzPbczOzzL",
                before: "aALrKAxqGqTKmbIaQBIW",
                first: 6876,
                last: 1571
              ) {
                edges {
                  cursor
                  node {
                    body
                    bodyHtml
                    createdAt
                    id
                    resolvable
                    resolvedAt
                    system
                    updatedAt
                    author {
                      avatarUrl
                      name
                      username
                      webUrl
                      snippets(
                        ids: [
                          "gid://gitlab/PersonalSnippet/20",
```

```
                              "gid://gitlab/PersonalSnippet/23",
                              "e\u0000",
                              "e\u0000"
                            ],
                            visibility: public,
                            type: project,
                            after: "VkZPAFgXGsLOdfHFUDRv",
                            before: "nWqYnCGxOSAGVQZHyclO",
                            first: 8620,
                            last: 9077
                          ) {
                            edges {
                              cursor
                              node {
                                content
                                createdAt
                                description
                                descriptionHtml
                                fileName
                                id
                                rawUrl
                                title
                                updatedAt
                                webUrl
                                visibilityLevel
                              }
                            }
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] *Digitale Transformation*, https://www.netzwerk-stiftungen-bildung.de/wissenscenter/glossar/digitale-transformation, zuletzt besucht: 03.08.2023.

[2] *Weltweiter IP-Traffic verdreifacht sich durch IoT und Video-Nutzung bis 2021*, https://www.zdnet.de/88300485/weltweiter-ip-traffic-verdreifacht-sich-durch-iot-und-video-nutzung-bis-2021/, zuletzt besucht: 03.0.2023.

[3] *GraphQL vs REST APIs*, https://hygraph.com/blog/graphql-vs-rest-apis, zuletzt besucht: 03.08.2023.

[4] *Was ist der Unterschied zwischen GraphQL und REST?* https://aws.amazon.com/de/compare/the-difference-between-graphql-and-rest/, zuletzt besucht: 18.06.2023.

[5] P. A. J. Offutt, *Introduction to Software Testing*. 2008, ISBN: 978-0-521-88038-1.

[6] D. S. Stefan Karlsson Adnan Causevic, "Automatic Property-based Testing of GraphQL APIs," *International Conference on Automation of Software Test*, 2021.

[7] *Evo Master*, https://github.com/EMResearch/EvoMaster, zuletzt besucht: 15.06.2023.

[8] S. Karlsson, A. Causevic und D. Sundmark, *QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs*, 2019. arXiv: 1912.09686 [cs.SE].

[9] *Rest Test Gen*, https://github.com/SeUniVr/RestTestGen/, zuletzt besucht: 15.06.2023.

[10] *GraphQL-API*, https://docs.gitlab.com/ee/api/graphql/, zuletzt besucht: 03.08.2023.

[11] R. Diestel, *Graphentheorie*. 2000, ISBN: 3-540-67656-2.

[12] *The Graph in GraphQL*, https://dev.to/bogdanned/the-graph-in-graphql-1l99, zuletzt besucht: 04.08.2023.

[13] H. Knebl, *Algorithmen und Datenstrukturen: Grundlagen und probabilistische Methoden für den Entwurf und die Analyse*. 2019, ISBN: 978-3-658-26512-0.

[14] *Was ist eine API?* https://www.redhat.com/de/topics/api/what-are-application-programming-interfaces, zuletzt besucht: 03.08.2023.

[15] *GraphQL-Specification*, https://spec.graphql.org/June2018/, zuletzt besucht: 16.06.2023.

[16] *GraphQL is the better REST*, https://www.howtographql.com/basics/1-graphql-is-the-better-rest/, zuletzt besucht: 15.06.2023.

[17] *State of GraphQL 2022 survey*, https://blog.graphqleditor.com/state-of-graphql-2022, zuletzt besucht: 15.06.2023.

[18] *Code using GraphQL*, https://graphql.org/code/, zuletzt besucht: 05.08.2023.

[19] *Apollo Server*, https://www.apollographql.com/docs/apollo-server/, zuletzt besucht: 05.08.2023.

[20] *Express GraphQL*, https://github.com/graphql/express-graphql, zuletzt besucht: 05.08.2023.

[21] *HyGraph*, https://hygraph.com/, zuletzt besucht: 05.08.2023.

[22] P. C. Jorgensen, *Software Testing: A Craftsman's Approach*. 2014, ISBN: 978-1-4665-6069-7.

[23] *Teststufen: WhiteBox und BlackBox-Testing*, https://hmc2.net/page12/page9/, zuletzt besucht: 05.08.2023.

[24] *Gray Box Testing*, https://www.geeksforgeeks.org/gray-box-testing-software-testing/, zuletzt besucht: 28.07.2023.

[25] M. Y. Mauro Pezze, *Software Testing and Analysis:Process, Principles, andTechniques*. 2007, ISBN: 978-0-471-45593-6.

[26] *Pytest: Helps you write better programs*, https://docs.pytest.org/en/7.4.x/, zuletzt besucht: 06.07.2023.

[27] P. A. J. Offutt, "Overview Graph Cooverage Criteria,"

[28] *Serene - Clojure.Spec from GraphQL Schema*, https://github.com/paren-com/serene, zuletzt besucht: 15.06.2023.

[29] *Clojure Spec - Data structure definiton*, https://clojure.org/guides/spec, zuletzt besucht: 15.06.2023.

[30] *Mali - Data Driven Specification Library for Clojure*, https://github.com/metosin/malli, zuletzt besucht: 15.06.2023.

[31] A. Belhadi, M. Zhang und A. Arcuri, *White-Box and Black-Box Fuzzing for GraphQL APIs*, 2022. arXiv: 2209.05833 [cs.SE].

[32] *Genetische Algorithmen - Optimierung nach dem Ansatz der natürlichen Selektion*, https://www.cologne-intelligence.de/blog/genetische-algorithmen, zuletzt besucht: 14.08.2023.

[33] S. D. et. al., "Deviation Testing: A Test Case Generation Technique for GraphQL APIs,"

[34] e. a. Louise Zetterlung Deepika Tiwari, "Harvesting production GraphQL Queries to Detect Schema Faults,"

[35] *NetworkX - Network Analysis in Python*, https://networkx.org, zuletzt besucht: 06.07.2023.

[36] *NetworkX - Network Analysis in Python*, https://github.com/networkx/networkx, zuletzt besucht: 06.07.2023.

[37] *Faker - Faker is a Python package that generates fake data for you.* https://github.com/joke2k/faker, zuletzt besucht: 06.07.2023.

[38] *Issue 1*, https://gitlab.com/gitlab-org/gitlab/-/issues/208672, zuletzt besucht: 30.07.2023.

[39] *Issue 2*, https://gitlab.com/gitlab-org/gitlab/-/issues/208125, zuletzt besucht: 30.07.2023.

[40] *Issue 3*, https://gitlab.com/gitlab-org/gitlab/-/issues/208122, zuletzt besucht: 30.07.2023.

[41] *Issue 4*, https://gitlab.com/gitlab-org/gitlab/-/issues/208121, zuletzt besucht: 30.07.2023.

[42] *Gitlab API Population Script*, https://github.com/gernhard1337/graphql-primepath-tester/blob/master/scripts/gitlab_population_skript.py, zuletzt besucht: 30.07.2023.

[43] *Gitlab Paths from PrimePath Generation*, https://github.com/gernhard1337/graphql-primepath-tester/blob/master/paths.txt, zuletzt besucht: 30.07.2023.

[44] *Explaining GraphQL Connections*, https://www.apollographql.com/blog/graphql/explaining-graphql-connections/, zuletzt besucht: 18.06.2023.

[45] *The Query and Mutation types*, https://graphql.org/learn/schema/the-query-and-mutation-types, zuletzt besucht: 04.08.2023.

[46] *Faker*, https://faker.readthedocs.io/en/master/, zuletzt besucht: 06.07.2023.

[47] *Max Query complexity*, https://docs.gitlab.com/ee/api/graphql/#max-query-complexity, zuletzt besucht: 28.07.2023.

[48] H. Noltemeier, *Graphentheoretische Konzepte und Algorithmen*. 2012, ISBN: 978-3-8348-1849-2.

[49] *Experimente Directory*, https://github.com/gernhard1337/GraphQL-Testautomatisierung/tree/main/experiment/toy-experiment, zuletzt besucht: 20.7.2023.

[50] *HTTP*, https://developer.mozilla.org/en-US/docs/Web/HTTP, zuletzt besucht: 09.07.2023.

Onlineressourcen wurden am 28. August 2023 auf ihre Verfügbarkeit hin überprüft.