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)

Betreuer 1: Prof. Dr. rer. nat. Leen Lambers
Betreuer 2: Prof. Dr. rer. nat. Gerd Wagner
Gutachter: 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 Zusammenfassung

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 Netzwerklast stark zu. [2] Bisheriger Standard für Kommunikation von Geräten über das Internet waren REST-APIs [3, vgl. Introduction] diese haben jedoch 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 in das 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 enorm 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, dass eine Methode vorstellt wie GraphQL-APIs mithilfe von Property-based Tests 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 Graphentheo-

rie herzustellen. Das gewonnene Wissen wurde verwendet um zu analysieren, welches Graphüberdeckungskriterium sich ideal eignen würde für einen Prototyp. Die erlangten Kenntnisse halfen bei der Entwicklung eines Prototypes für die Testentwicklung. In zwei Experimenten konnte 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 führen wir an das Thema heran und stellen unsere Motive dar. Wir definieren, welche Ziele wir in dieser Arbeit verfolgen und geben abschließend eine grobe Übersicht über die Kapitelstruktur.

2.1 Motivation

Mit einer steigenden Nutzung von GraphQL wird es immer wichtiger, Tests für GraphQL-APIs 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 selbst schreiben will oder von einem Tool automatisch generieren lasen will. Integrations-Tests, also Tests die Kombinationen einzelner Interaktionen von Funktionen miteinander testen, hingegen haben sehr oft einen sehr großen Testraum, sodass ein manuelles Schreiben dieser Tests fehleranfällig und langwierig ist. 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.

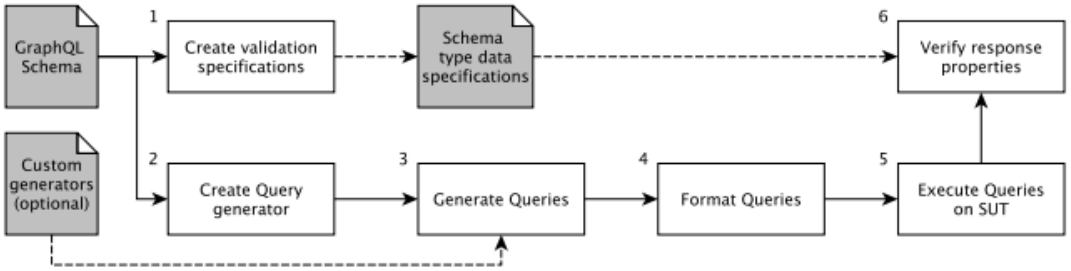


Abbildung 2.1: Methode von [6]

Es wird aus einem GraphQL-Schema ein Testgenerator entwickelt. Dieser kann aus der Typspezifikation, die GraphQL vorgibt, valide GraphQL-Querys entwickeln und diese mit verschiedenen Argumenten anreichern. Die generierten Querys stellen die entwickelten Tests dar. Das Besondere an GraphQL ist jedoch, dass es, wie der Name schon andeutet, einen Graphen umsetzt. Ein sehr einfaches Schema lässt sich beispielhaft in diesen Graphen übersetzen:

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

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

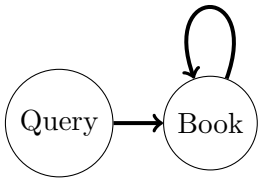


Abbildung 2.2: GraphQL-Schema als Graph

Generell starten alle GraphQL-Querys im Query-Type. Erlaubte Anfragen sind dann alle Pfade mit Ursprung im Query Knoten wobei Limitierungen implementiert werden können. Property-based Testing nimmt nun 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. Nutzen wir nun jedoch ein wesentlich größeres Schema, zum Beispiel die GraphQL-API von GitLab [10] so erkennt man schnell, dass der Graph so komplex wird, dass eine zufällige Pfadgenerierung zu unzuverlässig und ineffizient ist um eine große Struktur zuverlässlich zu testen. Einen ähnlichen Sachverhalt findet man in der Testgenerierung für Programmcode. Dieser kann sehr komplex werden und es müssen Strategien gefunden werden um diese effizient und zuverlässlich 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 man eine gute Testabdeckung entwickelt. An dieser Stelle sei insbesondere an *Introduction to Software Testing*[5] verwiesen. In [5] wird die Graphenüberdeckung 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. Wir wollen zeigen, dass das in [5] erarbeitete Wissen nicht nur für Testentwicklung von Programmcode zielführend ist, sondern auch für die Testgenerierung von anderen Graphstrukturen verwandt werden kann, in unserem Fall für die Testgenerierung für GraphQL-APIs. Unser Fokus wird sich auf die PrimePath-Überdeckung [5, vgl. Criterion 2.4] richten da wir uns erhoffen, dass diese 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. Wir beginnen mit der Definition einiger Konzepte aus der Graphentheorie in Abschnitt 3.1. Darauffolgend sehen wir uns GraphQL präziser in Absatz 3.3 an und verbinden dann beide Absätze miteinander, indem wir einen konkreten Zusammenhang zwischen beiden Themen herstellen. Abschließend für die grundlegende Theorie führen wir in Absatz 3.5 umfassend in das Thema Softwaretests ein. Die zuvor erarbeitete Theorie wird dann im Kapitel Graphüberdeckung 3.6 erweitert durch Theorien und Definitionen. In diesem Kapitel zeigen wir außerdem, inwiefern die zuvor definierten Übderdeckungskriterien hinreichend für Testgenerierung für Programmcode und GraphQL sind. Bevor wir unsere Methode entwickeln besprechen wir im Kapitel verwandte Arbeiten 5 wie der aktuelle Stand der Forschung zum Thema automatisiertes Testen von GraphQL ist. Mithilfe der zurvor entwickelten Theorie erarbeiten wir dann im Kapitel Testentwurf 6 ein Konzept, wie GraphQL automatisiert getestet werden kann. Dieses Konzept wird im Kapitel 7 eine praktische Umsetzung in einem Prototyp finden. Um nachzuweisen, dass der entwickelte Prototyp fähig ist, Fehler zu finden werden einige Experimente ausgeführt und mit der *Property-based* Methode [6] verglichen im Kapitel 8. Enden wird die Arbeit mit einem kleinen Ausblick in Kapitel 9 und einem Fazit über unsere erreichten Verbesserungen.

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 zweier Lehrbücher [5], [11] erarbeitet und in Kontext gesetzt. Wir benötigen die Graphentheorie 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)] benötigen wir die Graphentheorie, da diese Methoden liefert, um Graphen zu definieren und analysieren. Desweiteren sind die Testabdeckungskriterien die wir nutzen eng mit der Graphentheorie verbunden. Die folgenden Absätze werden eher theoretisch gehalten. Die Zusammenhänge zwischen der Graphentheorie und Testen von GraphQL-APIs werden sich jedoch später erschließen.

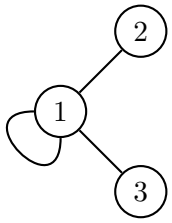
3.1.1 Graph

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

Definition 1 *Ein 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 nennt man Knoten (Vertices). Verbindungen zwischen den Knoten sind Elemente der Menge E und diese nennt man Kanten (Edges). In dieser Definition spielt die Ordnung der Elemente von E keine Rolle daher nennt man solche Graphen auch ungerichtete Graphen. 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:



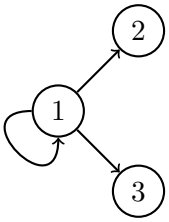
	1	2	3
1	1	1	1
2	0	0	0
3	0	0	0

3.1.2 gerichteter Graph

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

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 somit die Sortierung der Kantenpaare wichtig. Die Funktionen *init* und *ter* können am einfachsten durch die Sortierung der Kantenpaare erreicht 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 Graph sieht als gerichteter Graph so aus:



3.1.3 gewichteter Graph

Für die spätere Entwicklung unseres Testentwurfs ist es wichtig, gewichtete Graphen zu definieren. Zuvor wurden eine Kante in Definition **??** als ein Tupel (x, y) eingeführt. Ein gewichteter Graph weist jeder Kante nun ein Kantengewicht zu, dies ist im allgemeinen eine positiv, reelle Zahl [13, vgl. S. 251].

Definition 3 *Ein Graph G = (V,E) mit einer Abbildung g : E →$\mathbb{R}_{>0}$ heißt 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 unsere späteren Anwendungszwecke muss die Definition jedoch ein wenig allgemeiner gefasst werden. Wir wollen später die GraphQL-Typen, als Gewichte nutzen, um unsere Tests zu entwerfen. Dafür verallgemeinern wir die Definition 3 indem die Gewichtsfunktion angepasst wird. Wir nennen dies vorerst allgemein gewichteter Graph.

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

Wir werden später auf Definition 4 verweisen wenn der konkrete Testentwurf entwickelt wird.

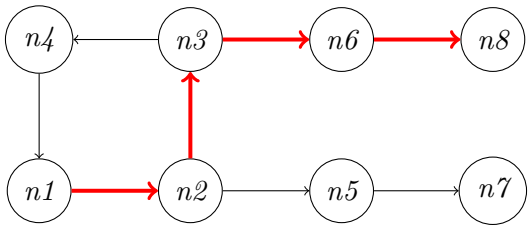
3.1.4 Pfad

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 Pfades die Summe aller Gewichte der einbezogenen Kanten. [13, vgl. 7.2 kürzeste Wege]

Beispiel 2 *Definieren wir einen Graphen G mit* $V = \{n1, n2, n3, n4, n5, n6, n7, n8\}$ *und* $E = \{(n1, n2), (n2, n3), (n3, n4), (n4, n1), (n2, n5), (n3, n6), (n5, n7), (n6, n8)\}$ *so wäre ein möglicher Pfad von Knoten n1 zu n8 der Pfad* $p = \{(n1, n2), (n2, n3), (n3, n6), (n6, n8)\}$ *Graphisch würde dies wie folgt aussehen (der Pfad in Rot markiert):*



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 die dieser Kreis bildet. Der kürzeste Kreis eines Graphens nennt sich $Tailenweiteg(G)$ und der längste Kreis ist der Umfang [11, vgl. S.8] Der Graph aus Beispiel 2 hat einen Kreis der Länge 4 und ist in Abbildung 3.1.5 rot eingezeichnet.
In unserem Kontext des Testentwurfs sind Kreise besonders interessant da diese für einen potentiell unendlich großen Testsraum sorgen. In einem azyklischen gerichteten Graphen, 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 beeinhaltet, diesen Kreis ein weiteres Mal ablaufen kann und somit stets ein neuer Pfad generiert wird.

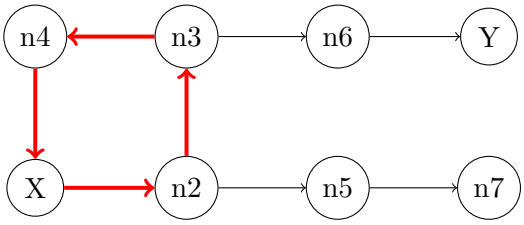


Abbildung 3.1: Ein zyklischer Graph

3.1.6 Zusammenfassung

Wir haben nun eine kleine Einführung in das Gebiet der Graphentheorie hinter uns. Mithilfe der hier erarbeiteten Begriffe und Definitionen werden wir im folgenden vermehrt arbeiten. Insbesondere, wenn wir die Überdeckungskriterien im Bezug des Testens einführen benötigen wir das hier erarbeitete Wissen.

3.2 APIs

Im folgenden wird häufig die Rede sein von APIs. Daher soll hier kurz geklärt werden, was eine API ist und wie diese im Allgemeinen funktionieren. API steht kurz für Application Programming Interface (Programmierschnittstelle) [14, vgl.]. 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. 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.

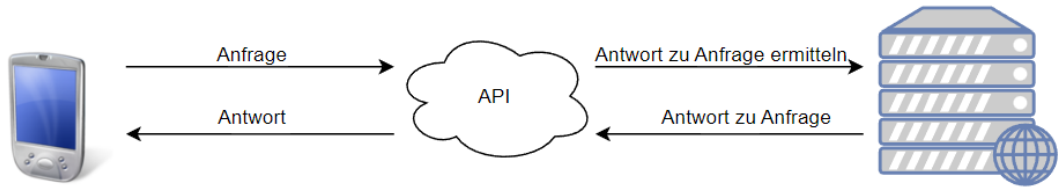


Abbildung 3.2: simple API-Kommunikation

Es gibt verschiedene Entwurfsmuster, die man für ein API Design nutzen kann. Die verschiedenen Muster werden durch Programmcode umgesetzt und sind somit fehleranfällig. In dieser Arbeit wird die Umsetzung von GraphQL-APIs betrachtet und der Programmcode hinter der GraphQL-API soll automatisiert getestet werden, sodass die Softwarequalität von der Programmierschnittstelle verifiziert werden kann.

3.3 GraphQL

GraphQL [15] ist eine Open-Source Query-Language (Abfragesprache) und Laufzeitumgebung die von Facebook entwickelt wurde[15, vgl. Introduction]. Die Besonderheiten von GraphQL sind, dass man mit nur einer einzelnen Anfrage mehrere Ressourcen gleichzeitig abfragen kann [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 die zeitgleich eine höhere Informationsdichte haben. GraphQL erleichtert außerdem die Kommunikation von Schnittstellen, indem die gewünschten Felder schon in der Query definiert werden und direkt den erwarteten Datentyp zusichern. Hier liegt auch der große Vorteil im Vergleich zum direkten technologischen Konkurenten REST API [4, vgl. Welche REST-Einschränkungen versucht GraphQL zu überwinden?]. Bei REST-APIs sind nämlich für verschiedene Ressourcen auch jeweils eine eigene Anfrage 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 von 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 unsere Tests aus GraphQL Anfragen bestehen, ist es für uns irrelevant welche konkrete GraphQL Serverimplementierung das zu testende System verwendet. Im Kontext dieser Arbeit ist ein tiefgreifendes, technologisches Verständniss von GraphQL essenziell, deshalb wird hier eine tiefgreifende Erklärung von GraphQL folgen.

3.3.1 Schema & Typen

Grundlage einer jeden 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 exisitieren [15, vgl. 3.2 Schema]. Ein GraphQL-Schema ist eine Sammlung von einzigartigen Typen und definiert die Einstiegspunkte der API. Es gibt drei Einstiegspunkte, die *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 die 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 ScalarTypes 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 dann wieder jeweils einen Typen um (InputObjectTypeDefinition ist dabei ausgeschlossen). Ein sehr einfaches Schema wäre zum Beispiel die Beziehung zwischen Büchern und Autoren. Ein Buch hat einen Titel und einen Author. Ein Author hat einen Namen und ein Geburtsdatum. Zugehöriges Schema für dieses Beispiel sähe wie folgt aus:

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

Abbildung 3.3: 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 folgende Einschränkungen für die Elemente des Tupels:

Feldname ein eindeutiger Feldbezeichner

Feldtyp gibt Einschänkungen vor, z.B. nicht Null (durch `!`), Listentyp (durch `[]`) etc.

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

3.3.2 Query

Da der Query-Type der Einstiegspunkt in alle Abfragen ist, welche es später zu testen gilt, soll dieser hier nocheinmal nähere Betrachtung finden. Sämtliche valide Abfragen beginnen im Query-Type. 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 auch in der zustellenden Abfrage benutzt wird. Die Felder der Antwort hängen hierbei vom Feldtypen ab. Ist das Ergebnis der Query-Definition ein ScalarType, so wird es direkt ausgegeben. Ist es ein anderer definierter Typ, so müss näher bestimmt werden welche Felder erwartet werden, dabei muss die Abfrage stets mindestens ein ScalarType enthalten. Nutzen wir das Beispiel der Bücher & Autoren weiter, könnte man wie folgt einen Query-Type definieren:

```
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 3.4: Query Type für Buch und Autor

Eine solche API wäre in der Lage, 3 verschiedene Anfragen zu beantworten. Man muss beachten, dass die Anfrage `getBookByTitle("Beispieltitel"){ author }` zwingend einen Scalaren Typen aus dem Typ Autor ausweisen muss. Die Anfrage wäre somit erst in dieser Form valide: `getBookByTitle("Beispieltitel"){ author{ name } }`

3.3.3 Resolver

Bisher beschäftigen wir uns vorrangig mit der Strukturierung und Typisierung von GraphQL und den Daten die durch das GraphQL Schema dargestellt werden. Ein wichtiger

Baustein fehlt aber noch. Woher kommen die Daten? Wie werden Eingabedaten behandelt? Diese Fragen werden durch die Resolver beantwortet. Ein Resolver ist in GraphQL eine Funktion die zuständig für die Datenabfragen und Strukturierung ist [19, vgl.]. Im Schema haben wir bisher definiert in welcher Art und Weise wir die Daten haben wollen, der Resolver ist nun dafür zuständig, diese Daten im definierten Format zur Verfügung zu stellen. Die Resolver sind nicht, wie alle vorher benannten Teil 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, dies 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 welchen GraphQL-Server man nutzt und welche Programmiersprache verwandt wird. Ein beispielhafter Resolver für die Query eines Buches anhand seines Titels mit ApolloServer in Javascript könnte folgende minimale Syntax haben:

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

Abbildung 3.5: ein einfacher Resolver

Wobei zu beachten ist, 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 da dieser Code Fehlerhaft sein kann. Ein Klassiker für 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 die gewährleistet, dass die möglichen Kombinationen ausreichend durch Tests abgedeckt sind und so die Qualität und Zuverlässigkeit von GraphQL-APIs erhöht wird.

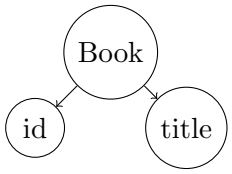
3.4 Zusammenhang Graphentheorie und GraphQL

Da wir nun ein grundlegendes Wissen über Graphentheorie und GraphQL erlangt haben, muss noch gezeigt werden, dass Graphentheorie auch anwendbar ist auf GraphQL. Wir nutzen diese Verknüpfung dann später, um Algorithmen die für Graphen gedacht sind für GraphQL zu nutzen.

3.4.1 Schema als Graph

Das GraphQL-Schema ist wie in 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. Definieren wir nun einen simplen Typen *Buch* welcher nur zwei Skalare Typen hat.

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



Dann hat der Typ *Buch* zwei Kanten, zu den jeweils beiden definierten Feldern *id* und *title*. In Kapitel 3.3.1 haben wir festegelegt, 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 nun aus einem beliebig großen Schema ein gerichteter Graph gebildet werden.

3.4.2 Abfragen im Graphen

Wie behandelt GraphQL nun Anfragen intern und wie können wir dies in unserer Graphstruktur visualisieren? Für die Erklärung definieren wir zuerst ein Schema als Beispiel:

```
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 3.6: Schemadefinition

Dieses Schema wird durch folgenden Graphen repräsentiert:

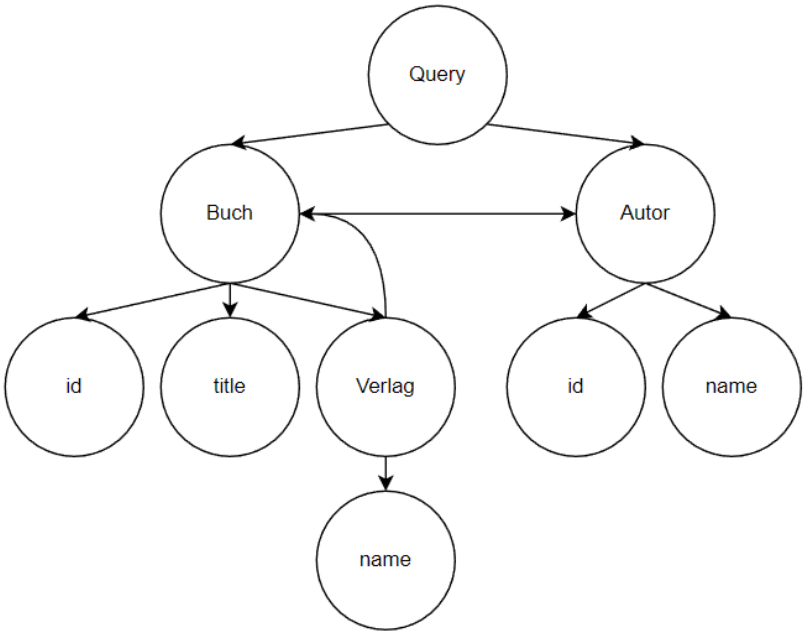


Abbildung 3.7: Graph für Schemadefinition aus Abbildung 3.4.2

Valide Anfragen an eine GraphQL-API sind nun 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. Wollen wir nun einen GraphQL-Server mit dem zuvor definierten Schema ein Buch mit id, Titel und Verlag mit Verlagsnamen abfragen, so können wir diese Query nutzen: `{ buch { id title verleger { name }}}`.

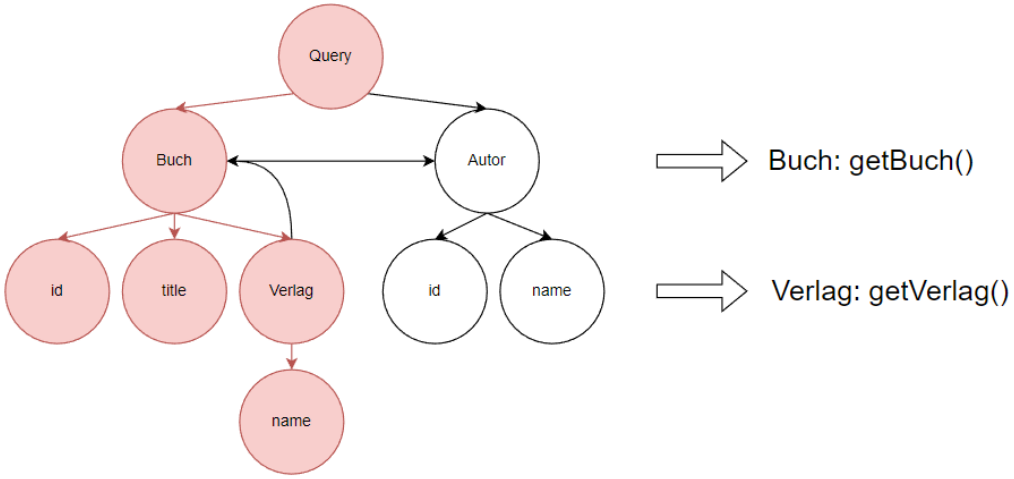


Abbildung 3.8: Graph für Abfrage nach [12]

Mit Ausführung dieser Query wird initial 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 validen 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 all dies sicherzustellen sind systematische Tests von Software nötig. Ziel ist es sicherzustellen, dass die Software den definierten Anforderungen und Spezifikation 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. 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 wir im Folgenden näher erläutern. Die Ansätze unterscheiden sich vor allem in den zur Verfügung stehenden Informationen über das System.

White-Box Testing

Im WhiteBox-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 soliden Niveau, dass begründet wird durch das Domänenwissen über das System. Verschiedene Techniken zur Analyse des Domänenwissens wurden entwickelt um die Informationen für die Testentwicklung zu nutzen. Wir werden anschließend in Kapitel 3.6 eine dieser Techniken näher untersuchen.

Black-Box Testing

Im BlackBox-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 gefordert ist [22, vgl. Specification-Based Testing]. Da der Quellcode nicht einsehbar ist muss man sich darauf verlassen, dass die Anforderungen treffend forumliert wurden [23, vgl.]. Das BlackBox-Testing hat einen methodischen Bezug zu Property-based Testing [6].

Grey-Box Testing

Das Grey-Box Testing ist eine Mischform von White und BlackBox Tests [23, vgl.]. Es sind in dieser Sicht Teile der Software bekannt aber man hat keinen umfassenden Einblick

wie im White-Box Testing. Es werden funktionale als auch strukturelle Testansätze verfolgt, je nachdem wie viel 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 das zu testende System gibt es verschiedene Granularitätsebenen der Tests. Tests können von 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, diese sind eng verbunden mit den Entwicklungsaktivitäten einer Software [5, vgl. 1.1.1].

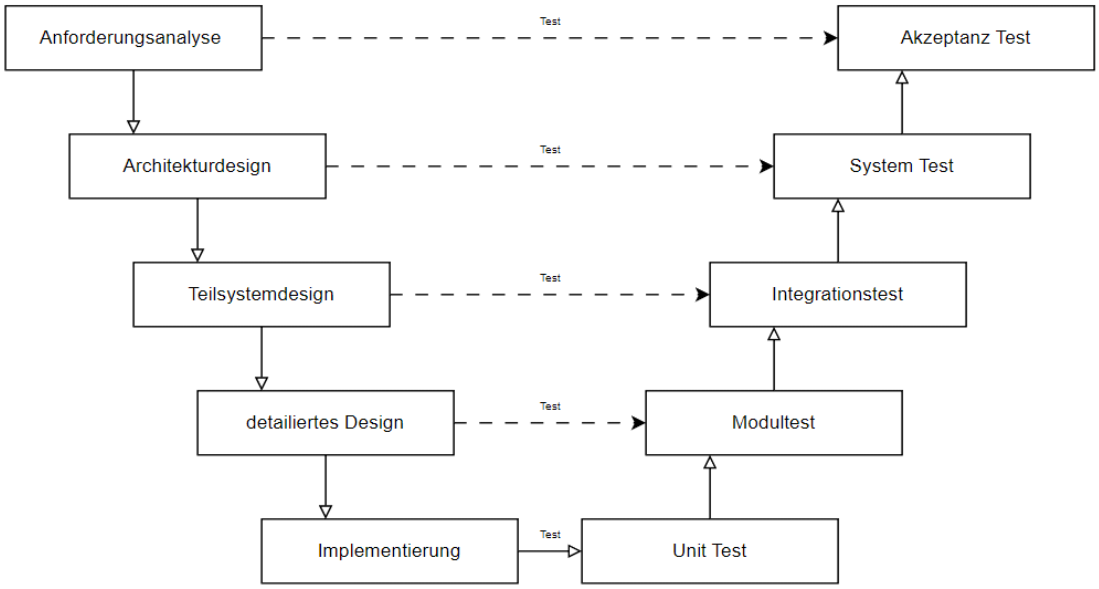


Abbildung 3.9: Softwareentwicklung und Test-Levels im V-Modell [5, vgl. Figure 1.2]

Die verschiedenen Testebenen sollten schon im Designprozess Beachtung finden, denn die Formulierung von Test kann dabei helfen Designfehler zu finden noch bevor die Software entwickelt wird . Die verschiedenen Ebenen der Tests spiegeln auch verschiedene Ansichten in den Tests wieder.

Akeptanz Test - Betrachten der Software hinsichtlich der Anforderungen

System Test - Betrachten der Software hinsichtlich der Architektur

Integrationstest - Betrachten der Software hinsichtlich der Teilsysteme

Modultest - Betrachten der Software hinsichtlich detaiiliertem Design

Unit Test - Betrachten der Software hinsichtlich konrekter Implementierung

[5, vgl. 1.1.1]

Im folgenden gehen wir noch einmal kurz auf jede einzelne Testebene etwas präziser ein. Dabei ist die Reihenfolge von feingranular zu grobgranular.

Unit Test

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

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

Abbildung 3.10: 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 3.11: Drei Unit Tests für die add-Funktion

Er prüft, ob die gegebene Einheit bei bekannten Eingaben die erwarteten Ausgaben liefert. Für diese Tests werden häufig Testframeworks genutzt die bei der Entwicklung und Ausführung der Tests helfen [5]. In Abbildung 3.10 wurde eine Funktion defineirt und in Abbildung 3.11 sind drei UnitTest mit PyTest [25] definiert. Die Tests führen verschiedene Methoden aus und prüfen, dass das Ergebnis mit der Erwartung übereinstimmt. Das *assertEqual* übernimmt dabei die Auswertung.

Modul Test

Eine Granularitätsebene höher ist das Modul Testen. 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].

```
        def add(a, b):
            return a + b
        def sub(a, b):
            return a - b
        def mul(a, b):
            return a * b
        def quo(a, b):
            return a / b
```

Abbildung 3.12: Eine Python Rechenmodul

Ein Modul wird nun getestet, indem geprüft wird, dass die einzelnen Funktionen untereinander richtig miteinander interagieren.

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

Abbildung 3.13: Eine Modul Test

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

Integrations Test

Das Integrationstesten übernimmt das Testen von Interoperabilität zwischen verschiedenen Modulen [5, vgl. S. 7]. Es wird dabei davon ausgegangen, dass die einzelnen Module zuvor korrekt getestet worden sind und die einzelnen Module korrekt arbeiten [5, vgl. S. 7]. Testobjekt sind vor allem die Schnittstellen der einzelnen Module und damit auch die Kommunikation zwischen den Modulen.

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

Abbildung 3.14: Modul 1

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

Abbildung 3.15: Modul 2

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

Abbildung 3.16: Integrationstest zwischen Modul 1 und Modul 2

Ein Integrationstest für beide Module wäre nun, zu testen, ob ein neu angelegter Nutzer ordentlich gespeichert wird und ob dabei die zugewiesenen Daten auch passend bleiben. Im Kontext dieser Arbeit werden wir untersuchen, wie solche 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 dann automatisiert zu testen.

System Test

Um zu testen, ob nun nicht nur einzelne Teile des Systems gut miteinander funktionieren, sondern auch das ganze System als solches, nutzt man die System-Tests [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. In dieser Ebene wird die Software aus Sicht des Endnutzers geprüft, oft ist der Endnutzer 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

Wir kennen nun verschiedene Ebenen und Sichtweisen des Testens. Allerdings ist noch nicht klar, wann ausreichend getestet wurde und ob wir überhaupt genügend testen können. Hierzu nutzen wir formale Abdeckungskriterien wie sie in [5] eingeführt werden.

Sie helfen uns, sinnvolle Testfälle zu entwickeln und zu entscheiden, wann ausreichend getestet wurde. Sollte man nämlich davon ausgehen, dass man einfach alle möglichen Eingaben an eine Funktion testen kann, so kommt man sehr schnell an die Erkentniss, dass dies unmöglich ist. Als Beispiel sei hier eine simple Addition von 2 64-bit Integer genannt. Für eine komplette Testung dieser simplen Addition gibt es `2^64  18 Trillionen` Kombinationen. Mit einem 3GHz Prozessor wäre eine vollständige Testung nach $2^{64}/3.000.000.000 \approx 6.149.571$ Sekunden (69 Tage) erledigt. Betrachten wir weiterhin einen Java-Compiler so ist der Eingaberaum von Programmen die zum Test stehen effektiv unendlich und somit nicht testbar [5, vgl. 1.3 Coverage Criteria for Testing].

Abdeckungskriterien

Wie gezeigt ist ein "vollständiges Testen also ein ausprobieren aller Möglichkeiten einfach unmöglich. Hierdurch sind wir gezwungen einen anderen Ansatz zu verfolgen. Abdeckungskriterien liefern hierbei einen Ansatz, die einem dabei helfen können, sinnvolle Tests zu entwickeln und zu entscheiden, wann genug Tests entwickelt wurden. Im folgenden sehen wir die Abdeckungskriterien auch als Testanforderungen [5, vgl. 1.3 Coverage Criteria for Testing]

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

Dabei soll ein Abdeckungskriterium erfüllt sein, 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 zu messen, wie gut ein Abdeckungskriterium durch eine Menge an Tests umgesetzt wird, führen wir die Abdeckung ein. Einerseits ist es manchmal sehr schwierig ein Abdeckungskriterium vollständig zu erreichen, andererseits können wir so messen, ob das Kriterium erfüllt wurde [5, vgl. S. 18].

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 soll (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 konzentrieren wir uns auf Abdeckungskriterien für Graphen.

3.6 Graphabdeckung

Wie zuvor gesehen, existieren verschiedene Abdeckungskriterien um Testabdeckung zu prüfen. Die Graphabdeckung führt verschiedene Kriterien ein, die es ermöglichen, aus Graphen Pfade für Tests zu generieren. GraphQL kann, wie in Kapitel 3.4 gesehen, durch einen Graphen repräsentiert werden. Hinter jedem Knoten in diesem Graphen steckt ein Resolver der potenziell einem eigenen Modul angehört. Anfragen an GraphQL können nun Resolver beliebig miteinander kombinieren. Um diese Kombinationen zu testen, wollen wir Graphabdeckungskriterien nutzen, um Tests zu generieren, die für eine ausreichende Testabdeckung zu sorgen. Da in GraphQL zyklische Graphen erlaubt sind, ist der Testraum potentiell unendlich groß. Dieses Problem kann durch die Verwendung von Graphabdeckung gelöst werden. Zuerst müssen wir ersteinmal weitere Theorie einführen. Um Graphcoverage zu nutzen, verfeinern wir die allgemeine Definition 2 von gerichteten Graphen. Im Testkontext definieren wir einen gerichteten Graphen wie folgt:

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 $E \subseteq N$ x N. Hierbei ist die Menge als initx x targety definiert.*

[5, 2.1 Overview]

Mithilfe dieser Definition können nun zum Beispiel Kontrollflussgraphen abgebildet werden, indem die Einstiegspunkte die Anfangsknoten sind und die Endknoten die Austrittspunkte. 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 nun mithilfe von Abdeckungskriterien Testpfade zu ermitteln die einen Graphen ausreichend abdecken. Ein Graph gilt als ausreichend abgedeckt wenn die folgende 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. [5, vgl. Def. 2.32]*

Hierfür existieren verschiedene Kriterien die wir im folgenden definieren wollen.

3.6.1 Graphabdeckungskriterien

Im folgenden Stellen wir verschiedene Graphabdeckunskriterien vor so wie sie in [5] definiert werden. Dabei ist ein Graphabdeckungskriterium eine Sammlung von Testanforderungen gemäß Definition 6 auf Graphen. Dieses Kapitel wird zunächst erst einmal sehr theoretisch, später folgt ein Vergleich der einzelnen Kriterien um diese besser einordnen zu können.

Knotenabdeckung

Erwartet man, dass beim Testen jede definierte Methode zumindest einmal ausgeführt wird, so handelt es sich hierbei um Knotenabdeckung. Diese 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 definieren wir dies in Definiton 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 Test besucht wird.

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. Man kann sagen, 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 führen wir die Kanten-Paar Abdeckung ein. Diese setzt vorraus, 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, müssen wir im folgenden auch in Beachtung ziehen, dass im Testkontext durchaus alle Pfadkombinationen die existieren relevante Testfälle darstellen können. Die Anzahl an allen azyklischen Pfadkombinationen wird jedoch selbst bei kleinen Programmen sehr 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 eingeführt.

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

und ein PrimePfad ist definiert als:

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. 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 erfüllt werden [5, vgl. S. 36]. Der Vollständigkeit halber wollen wir dieses Kriterium dennoch hier definieren.

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 und Komplexität. Ein kurzer Überblick soll hier gegeben werden.

Kriterium	Granularität	Komplexität
Knoten	Knoten	Minimal
Kanten	Kanten	Minimal
Kanten-Paar	Kantenpaare	abhängig von der Kantenmenge
PrimePfad	Hauptpfade	Komplex aber berechenbar
vollständige Pfade	Alle möglichen Pfade	Höchste, teils unberechenbar

Tabelle 3.2: Vergleich der Graphabdeckungskriterien

Besonders hervorzuheben ist, dass die hier vorgestellten Abdeckunskriterien hierarchisch sind und mit Erfüllung eines höheren Kriteriums automatisch alle tieferen Kriterien auch erfüllt sind [5, vgl. Figure 2.15]. Die Sortierung ist hierbei:

Definition 17 *Die Abdeckungskriterien sind wie folgt hierarchisch sortiert:*
Vollständige $>$ ***PrimePfad*** $>$ ***KantenPaar*** $>$ ***Kanten*** $>$ ***Knoten***
Die Sortierung legt fest, dass ein höher sortiertes Kriterium automatisch alle tiefer sortierten Kriterien erfüllt. Abdeckungskriterien sortiert nach Hierarchie [5, vgl. Figure 2.15].

Nach Definition 17 gilt also, dass zum Beispiel mit Kantenabdeckung automatisch auch Knotenabdeckung gewährleistet ist. Der interessierte Leser sei nun an [5] verwiesen falls die Anwendung von Abdeckungskriterien für Testgenerierung von Code interessant erscheint. Im Folgenden erarbeiten wir, wie Abdeckungskriterien für Testgenerierung von GraphQL nutzbar ist.

4 Graphabdeckung für GraphQL

Ziel dieses Kapitels soll es sein, 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 3.8 stellte den initialen Zugang dar, hierbei waren Knoten die ausgehende Kanten haben als Resolver zugeordnet. Ein Resolver ist wie zuvor in Kapitel 3.3.3 festgestellt eine Funktion eines Moduls das es zu testen gilt. Durch die Graphstruktur eines GraphQL-Schemas wollen wir nun Pfade generieren, 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 $N_f = N$ gilt, da jeder Knoten entlang eines Pfades des Graphens eine valide Anfrage ist. Im Sinne das Integrationstesten ist es nun wünschenswert, möglichst viele Kombinationen einzelner Module durch diese Pfade abzubilden. Wir wollen nun die zuvor eingeführten Abdeckungskriterien dahingehend untersuchen.

4.1 Knotenabdeckung für GraphQL

Die Knotenabdeckung zielt darauf ab, dass jeder Knoten in mindestens einem Testpfad Berücksichtigung findet. In GraphQL sind Knoten als Type definiert. Jeder Type definiert seinen eigenen Resolver. Dadurch wird mit der Knotenabdeckung sichergestellt, dass zumindest jeder Resolver einmal ausgeführt wird. Ein Type hat jedoch ausgehende Kanten welche ihn mit anderen Resolver verbinden. Der Graph in Abbildung 3.7 vom Schema aus Abbildung 3.4.2 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 potentiell Fehler aufweisen und soll daher auch getestet werden. Gleiches gilt für die Kante *autor* des Typen *Buch*. Wr folgern also, dass dieses Abdeckungskriterium unzureichend ist um Intergrationstest abzubilden.

4.2 Kantenabdeckung für GraphQL

Zuvor wurde deutlich, dass die Abdeckung aller Kanten essenziell ist um GraphQL gut zu testen. Mit der Edge-Coverage zielen wir genau darauf ab, dass jede Kante in mindestens einem Testpfad berücksichtigt wird. Die Edge-Coverage findet auch Anwendung in *Property-based Testing*[6, vgl. D-RQ1]. Allerdings sind im Testkontext von GraphQL auch die Kombinationen von Kanten interessant. Durch Kantenabdeckung wäre der Testpfad $Query \rightarrow Buch \rightarrow Autor \rightarrow Buch \rightarrow Verlag$ im Graphen aus Abbildung 3.4.2 nicht berücksichtigt, obwohl durchaus interessant wäre, 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 betrachten wir alle Kantenpaare. Dadurch erlangen wir eine bessere Abdeckung der Funktionen, indem sichergestellt wird, dass jede Kante mit jeder darauffolgenden Kante einmal ausgeführt wird. Dieses Kriterium stellt eine Verbesserung der Kantenabdeckung dar, allerdings werden eben nur aufeinanderfolgende Kantenpaare abgedeckt. GraphQL kann aber wesentliche 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, erreichen wir eine bessere Abdeckung als die Kanten-Paar Abdeckung. Im Prinzip enthält diese Abdeckung alle möglichen Kantentupel ohne Wiederholungen von Knoten [5, vgl. S. 42]. Dadurch erreichen wir, dass jede 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

Mit der vollständigen Pfadabdeckung haben wir als Ziel, alle Pfade, die möglich sind, zu generieren und in unseren Tests zu berücksichtigen. Da GraphQL jedoch Zyklen erlaubt ist die Anzahl an potentiellen Pfaden möglicherweise unendlich. Hierdurch folgt, dass auch der Testraum unendlich werden würde mit der vollständigen Pfadabdeckung. Somit ist dies nicht umsetztbar da wir im Allgemeinen nicht ausschließen können und wollen, dass GraphQL keine Zyklen haben kann. Sollte das Schema nativ azyklisch sein, so wäre eine Umsetzung dieses Kriteriums denkbar.

4.6 Fazit

Die beiden geeignetsten Coveragekriterien sind die vollständige Pfadabdeckung und PrimePfad Abdeckung wobei die vollständige Pfadabdeckung im Allgemeinen nicht verwendet werden kann da sie zu restriktiv in der Graphstruktur ist. In der Praxis hat sich gezeigt, dass der Großteil der GraphQL-Schemas Zyklen hat. Dadurch ist die vollständige Pfadabdeckung im Allgemeinen nicht zu nutzen und das nächst schwächere Testkriterium, die PrimePfad Abdeckung, ist zu wählen. Wie in Definition 17 gezeigt, ist das nächst

schwächere die PrimePfad-Abdeckung, dieses erfüllt gleichzeitig alle schwächeren Kriterien. Mithilfe dieses Kriteriums wollen wir im folgenden einen Prototypen entwickeln, der Integrationstests für GraphQL automatisiert erstellt.

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 Routen zufällig generiert werden. Wie zuvor erwähnt soll diese Arbeit als Motivation für unsere zu entwickelnde Methode dienen, indem wir versuchen eine bessere Graphabdeckung zu erreichen als durch das Zufällige Testen welches einige Limitierungen aufweist. Wir wollen hier das *Property-based Testing* noch einmal untersuchen und die Probleme konkreter aufzeigen. Der allgemeine Funktionsablauf der Testgenerierung laut Paper ist wie folgt:

1. Vom Schema, generiere Typ-Spezifikationen
2. Generiere einen Generator der zufällig eine Liste an Query-Objekten erstellen kann
3. Generiere n Querys
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 wird der Hauptunterscheidungspunkt beider Arbeiten sein, denn hier sind dann zwei gänzlich unterschiedliche Konzepte umgesetzt Dieser ist beim Property-Based Testing,nämlich ein Query-Generator der mithilfe der Clojure-Bilbiothek Serene[26] Clojure.Specs[27] generiert und diese Clojure.Specs[27] dann nutzt, um mit der Clojure-Bilbiothek Malli[28] dann Daten für die Testqueries zu generieren. Betrachtet man die

Arbeit aus einer graphentheoretischen Sicht, so zeigt sich, dass das Property-based Testing auf Graphstrukturen arbeitet, diese allerdings in seiner eigenen Verarbeitung nicht umsetzt. Unsere herangehensweise wird sich hiervon gänzlich unterscheiden. Im Sinne vom Property-based Testing ist diese Herangehensweise allerdings sehr sinnvoll gewesen da Malli[28] de-facto Standard für Property-based Testing in der Clojure-Welt ist. Hauptbeitrag der Arbeit war, einen Generator für Malli zu schreiben der in der Lage ist GraphQL-Querys zu generieren. Geht man jedoch davon aus, dass das Ziel eine ideale Überdeckung des Graphens jeder größe und jeder Struktur 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 betrachten wir als falsch und behaupten, dass es besser möglich ist. Dies zu zeigen bleibt Gegenstand der folgenden Arbeit.

5.2 heuristisch suchen basiertes 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* [29] wurde eine Erweiterung für EvoMaster erstellt welches GraphQL Tests generieren kann. Hierbei soll sowohl WhiteBox als auch BlackBox Testing möglich sein. Das erstellte Framework in diesem Paper 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 welcher die Tests generiert. 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 [30, vgl.]. Während ein genetischer Algorithmus sich einer Lösung annähert, berechnet er diese jedoch nicht zuverlässig ideal [30, 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, dass die Testpfade erfüllen müssen, die durch unseren 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 - allerdings muss starkes Domänenwissen in den Suchalgorithmus eingear-

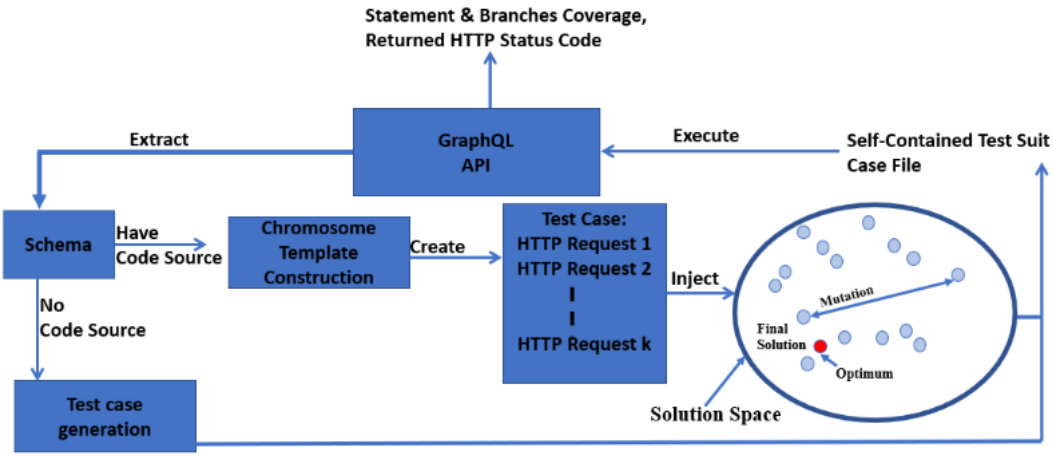


Abbildung 5.1: Arbeitsweise EvoMaster

beitet werden, sodass man noch fern von einer WhiteBox-Testautomatisierung ist [29, 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* [31] wird diese Gegebenheit benutzt und aus einer selbstdefinierten Query werden verschiedene Variationen erzeugt [31, 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 jeweils 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 Pfades zu maximieren jedoch nicht die Graphabdeckung insgesamt zu erhöhen.

5.4 Query Harvesting

Klassisches Testen von Anwendungen beeinhaltet, dass nach Möglichkeit das komplette System getestet wird bevor es verwendet wird. Im Paper *Harvesting Production GraphQL Queries to Detect Schema Faults* [32] 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 gesammelt Anfragen Tests generiert. Dabei wird für jede gesammelte Anfraege 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 Querys 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 ähnlichem. 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 Querys 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 Querys
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 wir zuvor festgestellt haben, dass die PrimePfad Abdeckung potenziell das sinnvollste Abdeckungskriterium ist, wollen wir im folgenden eine Methodik entiwckeln die es erlaubt, mithilfe dieses Abdeckunskriteriums Tests für GraphQL zu entwerfen. Die zu entwickelnde Methodik wird in einigen Teilen stark an der Methode aus [6] orientiert sein, dies wird jedoch an den betreffenden Stellen kenntlich gemacht. In diesem Kapitel wird die Methodik konzeptionell entwickelt 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 gezeigtem Muster.

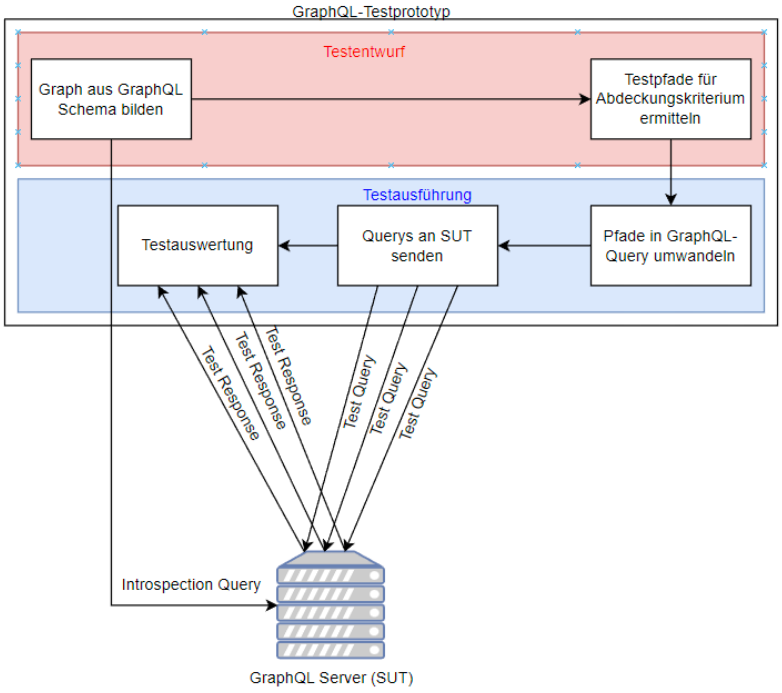


Abbildung 6.1: Grober Ablauf des Testprozesses

Wie in Abbildung 6.1 zu sehen ist der gesamte Testprozess in zwei Teile aufgeteilt. Einerseits in den Testentwurf und 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 ermittelt, 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 einer Introspection-Query 12 lässt sich das gesamte Schema eines GraphQL-Servers abrufen. Die Introspection-Query existiert in verschiedenen Varianten, wir nutzen hier die exakt gleiche Version wie sie auch von [6] genutzt wird. Ergebnis der Introspection Query ist ein JSON-Objekt mit einer Struktur wie in Listing 6.1.1 gezeigt.

```
{
    "data": {
        "__schema": {
            "queryType": {},
            "mutationType": {},
            "subscriptionType": {},
            "types": [],
        }
    }
}
```

Listing 6.1: Schema-Response

Der Eintrag *queryType* gibt den Namen des Typens an, der Startpunkt jeder Query ist, so wie in Kapitel4 festgelegt. Mit dem Eintrag *types* erhalten wir eine Liste aller Typen wobei jeder Eintrag der Struktur in Listing6.1.1 entspricht.

```
{
  "kind": "",
  "name": "",
  "description": "",
  "fields": [],
  "inputFields": [],
  "interfaces": [],
  "enumValues": [],
  "possibleTypes": []
}
```

Listing 6.2: Type-Field

Um nun aus dem Schema einen Graphen zu erstellen, benötigen wir die Felder *kind*, *name*, *fields*. *kind* ist die Angabe, 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 Schema an.

Um einen Graphen aus dem Schema zu entwickeln benötigen wir nur Felder vom Typ *OBJECT*. Die Menge aller Objekte vom Typ *OBJECT* sind die Menge aller Knoten unseres Graphens. Um nun die Kanten, also die Beziehungen zwischen diesen einzelnen Knoten zu bekommen müssen wir uns die Defintion eines Typens näher ansehen. 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 Kantensuche das Feld type besonders wichtig ist. Dieses ist wie folgt definiert:

```
1    {
2        "kind": "",
3        "name": "",
4        "ofType": null
5    }
```

Listing 6.4: Type-Field

Wenn nun der Eintrag *kind* den Wert *OBJECT* trägt, so ist klar, dass unser hier definiertes *OBJECT* eine Kante zum Knoten *name* besitzt.

6.1.2 Testpfade ermitteln

Da wir nun einen Graphen passend zum Schema ermittelt haben, gilt es, die Testpfade zu ermitteln, welche die PrimePfad-Abdeckung erfüllen. Hierzu nutzen wir den in [5, Finding Prime Test Paths] vorgestellten Algorithmus. 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 Pfades 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]. Nun wird von jedem Knoten aus expandiert und eine Liste über alle Pfade geführt. Pfade werden nicht weiter expandiert, wenn diese einen Knoten doppelt enthalten. Endergebnis ist dann eine Liste aller einfachen Pfade. Filtert man die einfachen Pfade heraus, die Teil eines anderen Pfades sind erhält man nach Definition 14 die PrimePfad Abdeckung da wir alle PrimePfade gefunden haben denn es gilt, dass die Menge der Primepfade eine echte Teilmenge der einfachen Pfade ist [5, vgl. S. 35]. Mit der Einschränkung von GraphQL, dass valide Querys stets im Query-Knoten starten müssen, muss sichergestellt werden, dass die PrimePfade dort starten. Um dies umzusetzen legen wir fest, 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.

6.2 Testausführung

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

6.2.1 Pfade in Query umwandeln

Einen Pfad wandeln wir in eine konkrete Query um indem wir die Typinformationen aus dem Schema nutzen. Beginnend im Query-Knoten wird der Pfad iteriert. Das GraphQL-Schema enthält Informationen darüber, welche Informationen nötig sind um zum nächsten adjazenten Knoten des Pfades zu kommen. Die Informationen darüber sind im Eintrag $fields$ enthalten wie in Listing 6.1.1 gesehen. Im $fields$ Eintrag steht dann, ob eine Kante Argumente benötigt und welcher Typ das Rückgabeobjekt ist. Das Rückgabeobjekt des $fields$ steht dabei aber schon fest da dieser exakt gleich sein muss mit dem nächsten Knoten des Pfades. 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 inkludieren wir schlichtweg alles. Felder vom Typ $OBJECT$ werden nur zur Query hinzugefügt, wenn der Typ des $OBJECT$ dem nächsten Knoten entspricht. Im Allgemeinen lässt sich das Verfahren in diesem Pseudocode darstellen:

```
path = (A , B , ..... , Y)
query = {}

while path not empty:
    knoten = pfad.pop()
    ScalarFields = getScalarFields(knoten)
    query.addScalars(ScalarFields)
    edge = pfad.peek()
    args = checkForEdgeArgs(edge)
    query.addArgs(args)
return query
```

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] genutzt. Dabei wird der Typ des Arguments genutzt um zufällig ein Argument des entsprechenden Typens 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 Querys an Server senden

Die generierten Querys stellen die konkreten Tests für den GraphQL-Server dar. Im folgenden nennen wir den zu testenden GraphQL-Server vermehrt SUT - System under Test. Um diese auszuführen, werden alle generierten Querys per *HTTP-POST* an den GraphQL-Server geschickt und die Antworten werden gespeichert, dies ist analog zu [6]. Es ist wünschenswert, dass die generierten Querys 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 im Grunde auf den selben 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-Server 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 gefundenere 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 6.5

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

Listing 6.5: fehlerhafte Antwort

Hierbei müssen die *errors* jedoch manuell geprüft werden ob es sich um wirkliche Programmierfehler handelt oder gewünschtem 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 umgesetzt wird. Sehr häufig kommt es vor, dass zufällig generierte Argumente schon in den Anfängen des Pfades nicht passend zu den unterliegenden Daten sind. Dadurch folgt, dass ein Großteil der Testpfade die theoretisch eine gute Abdeckung aufweisen, praktisch diese Abdeckung nicht erreichen. Um Messen zu können, ob ein Pfad seine theoretische Abdeckung auch praktisch erreicht, führen wir eine Abschätzung darüber ein.

Abschätzung der Pfadlängen

Diese Methode verbessert zwar nicht die Testergebnisse allerdings gibt Sie uns Informationen darüber wie viel von unserem Pfad in Wirklichkeit abgedeckt wurden. Dadurch lässt sich der Erfolg der Tests besser abschätzen da wir so messen können, ob die Querys wirklich die Funktionen ausgeführt haben. Hierzu wird die Pfadlänge des Pfades 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 unser 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 hätte folgende Response eine Tiefe von 2

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

Listing 6.6: vollständige Response

Und die leere Antwort hätte eine Tiefe von 1

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

Listing 6.7: mangelhafte Response

Obwohl eine leere Response zulässig ist und nicht auf einen Fehler hindeutet, signalisiert uns der Unterschied zwischen erwarteter und tatsächlicher Länge dann, ob die Query tatsächlich alle Resolver ausgeführt hat oder nur einen Teil davon. Hierdurch können wir die Tests in ihrer Qualität auswerten. Wir können die Pfadlängen aller erwarteten Pfade addieren, das gleiche müssen wir auch mit den tatsächlichen Pfadlängen machen. So erreichen wir zwei Zahlen und mit diesen können wir eine prozentuale Einschätzung abgeben, wieviel Prozent unserer Tests insgesamt ausgeführt wurden. Wir rechnen hierfür:

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

Wir sollten einen Wert von 100% anstreben. Dies würde bedeuten, dass unsere generierten Tests auch alle Funktionen getestet haben. Andernfalls bedeutet ein Prozentsatz

unter 100% eben, dass nicht alle Funktionen tatsächlich von den Querys überdeckt wurden.
Abschließend wollen wir noch kurz erläutern, wie es möglich wäre, die Tatsächliche Abdeckung zu erhöhen. Dies geschieht vor allem durch ein anpassen der Zufallsgeneratoren und die Anzahl der Querys.

Zufallsgeneratoren der Argumente

Die zuvor vorgestellte Abschätzung liefert uns einen Hinweis darauf, wie gut unsere Querys tatsächlich getestet haben. Ein Ansatz der die Querys eine bessere tatsächliche Abdeckung zu erreichen lässt ist das anpassen der Generatoren für die Argumente. In der vorgestellten Methode in Kapitel 6.2.1 erstellen wir komplett zufällig Argumente für die Funktionen. Dies bedeutet, dass z.B. der Type ID als String gewertet wird. Dieser Type ist in der Realität jedoch eingeschränkt und gleichzeitig sehr bedeutend, da dieser häufig als Argument angegeben wird und er eine spezielle Struktur hat. Es hängt natürlich stark von der eigenen Implementierung der GraphQL-API ab allerdings wenn in der Implementierung eine ID definiert ist als Zahlenstring, so kann es sich durchaus lohnen, dass der Argumentgenerator für die ID auch speziell auf Zahlenstrings angepasst wird. Alternativ kann auch eine Liste aller existenten IDs angegeben werden und zufällig 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 mit der Argumente zufällig generiert werden, die dann tatsächlich zu Zugrunde liegenden Daten passen.

Anzahl der Querys

Die Wahrscheinlichkeit passende Argumente zu generieren steigt außerdem mit der Anzahl an generierten Querys. Erhöht man die Anzahl an generierten Querys pro PFad, so erhöht sich auch die Wahrscheinlichkeit, dass zumindest ein Pfad eine gute Abdeckung erreicht. Hierfür müssen wir die Methode aus Kapitel 6.2.1 so oft wie gewünscht wiederholen, sodass wir verschiedene Querys mit verschiedenen Argumenten erhalten. Bei dieser Methode ist wichtig, dass sich die Argumente der Querys verändern da wir sonst einfach mehrfach die selbe Query stellen mit der gleichen zu erwartenden Antwort. Zusammen mit den angepassten Argumentgeneratoren kann so das zufallsbasierte Argumentgenerieren ein wenig begrenzt werden und es ist wahrscheinlicher, dass gute Tests entstehen.

6.3 Zusammenfassung der Methode

Wir wollen im folgenden die eben vorgestellte Methode noch einmal kurz zusammenfassen damit diese übersichtlicher wird. Unsere hier vorgestellte Methode funktioniert so wie in Abbildung 6.1 gezeigt. Wie zu sehen, ist der ganz grobe Ablauf ähnlich zum [6, Property-based Testing] allerdings unterscheidet sich die Methode in einigen Teilen sehr stark vom [6, Property-based Testing]. Wir fügen in unserer Methode den Schritt hinzu,

dass wir einen Graphen erstellen welcher die Knoten und Kanten des GraphQL-Schemas repräsentiert während in [6, Property-based Testing] ausgehend vom Query-Type zufällig bis zu einer bestimmten Pfadlänge (dem Rekursionslimit) die Pfade gebildet werden, indem immer zufällig Felder hinzugefügt werden. Durch unsere Methode erreichen wir, dass Pfade jeder Länge, die durchaus länger sein können als ein definiertes Rekursionslimit, abgedeckt werden und somit die Tests eine bessere Abdeckung erreichen können. Unser Ansatz erlaubt es außerdem, verschiedene Abdeckungskriterien zu implementieren. So ist man nicht gezwungen auf einer Methode zu verharren, sondern kann je nach Implementierung die Pfadgenerierung anpassen nach den individuellen Anforderungen ohne, dass in anderen Schritten etwas geändert werden muss. Bei der Umwandlung der Pfade in Querys unterscheidet sich unser Ansatz ein wenig von [6, Property-based Testing]. In unserer Methode generieren wir aus dem Pfad direkt die Query und generieren die nötigen Argumente "on-the-fly" während sie erkannt werden. Im Property-based Ansatz wird ein Datenobjekt als ganzes erstellt, dass die Query später generieren kann. In der technischen Umsetzung unterscheiden sich beide Methoden, im Ergebnis bekommen Sie jedoch strukturell gleiche Querys. Die Ausführung der Tests hingegen unterscheidet sich überhaupt nicht mehr zum [6, Property-based Testing]. Die Einführung der erwarteten gegenüber der tatsächlichen Pfadlänge ist ein neuer Ansatz, der die Qualität der zu testenden Querys messbar macht - dies fehlt im [6, Property-based Testing]. Dort ist man im unklaren darüber wie gut die Tests genau getestet haben und ob die erwartete Abdeckung auch mit der tatsächlichen Übereinstimmt.

Wir haben nun unsere Methode im groben vorgestellt und Unterschiede zum schon bestehenden Ansatz [6, Property-based Testing] erörtert. Im folgenden wollen wir uns der praktischen Umsetzung dieser Methode widmen und einen Prototypen entwickeln. Dieser Prototyp soll dann gegen das [6, Property-based Testing Tool] antreten und möglichst zeigen, dass die eben entwickelte Methode eine Verbesserung darstellt.

7 Testautomatisierung

Nach der Einführung der Methode im vorherigen Kapitel soll nun der entwickelte Prototyp erklärt werden. Der entwickelte Prototyp lässt sich im GitHub und im BTU-GitLab finden und testen. Eine Anleitung findet sich in der Readme im Root-Verzeichnis. Vorraussetzungen zum Ausführen der Anwendung ist Python und einige 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 schon einige Erfahrungen mit dieser Bibliothek exisitieren 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-Biblitohek für *Erstellung, Manipulation und Untersuchung der Struktur, Dynamik und Funktionen komplexer Netzwerke* [33, vgl. Startseite] Mit einer Star-Anzahl von *12.8k*[34] auf GitHub ist networkX eine sehr beliebte Bibliothek. NetworkX ist die ideale Wahl um Graphen zu erstellen für unseren Use-Case denn es nimmt jeden möglichen Datentypen als Wert für einen Knoten und Kante. Wir können also sehr simpel Graphen definieren. Für ein enfaches Beispiel von Author, Book, Publisher und deren Verbindungen benötigen wir nur folgende Zeilen Code:

```
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")
```

Diese wenigen Zeilen reichen aus um unseren Graphen mit allen Knoten und Kanten zu definieren. Wir können hier auch direkt das Kantengewicht mit dem Feldbezeichner angeben, so ist es möglich, dass wir sofort wissen, welche Kante genutzt werden muss um zum nächsten Typ zu gelangen. Auf diesem Graphen können wir dann diverse Algorithmen ablaufen lassen. Diverse Hilfsfunktionen helfen dabei eine effiziente Programmierung zu erlangen. Hierbei seien insbesondere folgende Hilfsfunktionen genannt:

draw

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

Zeichnet einem den erstellten Graphen in ein beliebiges Format. So fällt es einfach große Graphen darzustellen.

shortest_path

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

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

neighbors

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

Diese Funktion liefert alle Nachbarn eines Knotens.

simple_paths

Diese Funktion liefert uns alle einfachen Pfade wie in Kapitel 6.1.2 gewünscht. So ist es uns direkt möglich, aus dem Ergebnis dieser Funktion die PrimePfade herauszufiltern. Somit erleichtert die Biblitohek uns die Berechnung der Pfade stark.

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

7.1.2 Faker

Die gewählte Argumentgenerierungsbibliothek ist *Faker*[35]. Mit *16k*[35] Sternen auf GitHub ist Faker auch eine beliebte Bilbiothek. Faker ist eine Bilbiothek die es sehr einfach macht Daten zu generieren. Da wir im Kontext von GraphQL Argumenten nur sehr einfache Datentypen als Argumente benötigen reicht uns diese Bibliothek komplett aus da sie es schafft uns schnell und unkompliziert Daten in genau dem Format zu generieren wie wir sie brauchen. Angenommen wir benötigen einen String der 10 Zeichen lang ist, so reicht eine Zeile:

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

Selbiges falls wir eine Zufallszahl benötigen zwischen 1 bis 1000

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

Dieses Schema des Einzeilers gilt für alle simplen *SCALAR* Types in GraphQL. Daher fällt die Wahl für die Datengenerierung auf diese Bibliothek.

7.1.3 PyTest

Um die gewünschte Reproduzierbarkeit aus Kapitel 6.2.2 zu erreichen, nutzen wir das Testframework PyTest. Dies ist ein Testframework für Python welches eine simple und einfache Testdefinition ermöglicht. Ein Test für eine einfache Funktion *inc* kann mit *test_inc* umgesetzt werden.

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

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

Die einfache Syntax von PyTest reicht für unseren Anwendungsfall vollkommen aus. Gleichzeitig sind Imports in PyTest sehr einfach umzusetzen daher nutzen wir dieses Testframework.

7.2 Umsetzung der Methode

Für die Umsetzung der Methode werden wir durch die einzelnen Teile des Codes gehen und die jeweiligen Schritte erklären. Hierbei gehen wir chronologisch in den einzelnen Schritten vor so wie in der Methode definiert. Im Allgemeinen funktioniert der Prototyp so wie in dem Sequenzdiagramm in Abbildung 7.1 gezeigt.

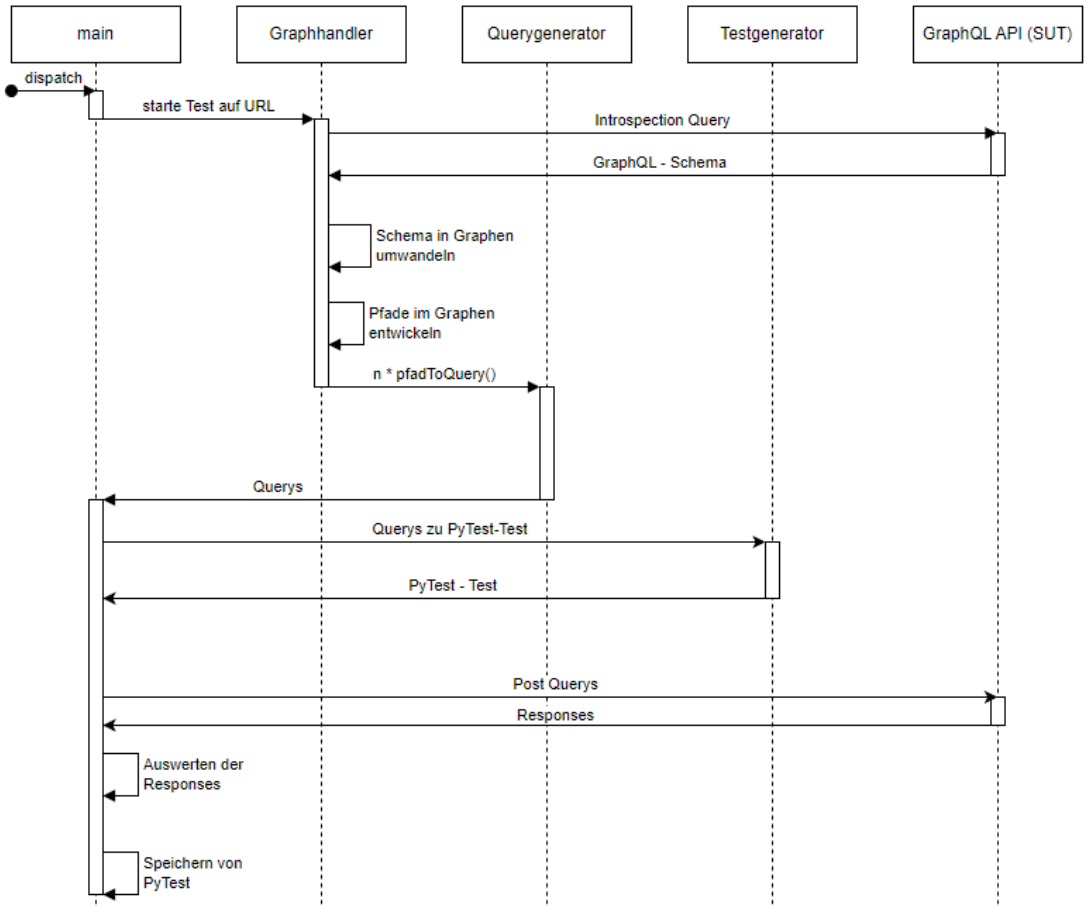


Abbildung 7.1: Sequenzdiagramm des Prototypens

Hierbei sind auch die einzelnen Module zu erkennen. Dabei sind die Module main, Graphhandler, Querygenerator und Testgenerator Teile des Prototypens. Das Modul GraphQL API stellt das zu testende System dar und ist extern.

7.2.1 Schema in Graph abbilden

Wie in der Vorstellung der Methode in Kapitel 6.1.1 bilden wir das GraphQL-Schema in einem NetworkX-Graphen ab. Um die Informationen zu erlangen die für die Bildung des Graphens wichtig sind führen wir zuerst die Introspection-Query 12 aus. Das Ergebnis ist dann das vollständige GraphQL-Schema der API. Hierbei sei angemerkt, dass einige GraphQL APIs so eine Introspection-Query verbieten, sei es einerseits durch direktes verbieten oder ein Tiefenlimit in den Querys. Egal was hierbei der Fall ist, die zu testende API muss unsere Introspection-Query 12 unterstützen, da wir sonst keine Informationen erlangen 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)
```

Und die Response wird als JSON-Objekt in *json_data* gespeichert. Es wurde ein Modul *Graphhandler* entwickelt, dass verschiedene Graphoperationen Übernimmt. Im Graphhandler ist eine Funktion *buildGraph* definiert. Diese generiert einen Graphen von einem gegebenen Startknoten, einem leeren Graphen und dem Schema. Hierbei werden nur erreichbare Knoten vom Startknoten berücksichtigt. Setzt man den Startknoten auf den Knoten *Query* so inkludieren wir auf diese Weise nur alle erreichbaren Teile des Graphens ausgehend von *Query*. Dies ist insofern sinnvoll da 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 (im Allgemeinen *Query*) aus rufen wir die Funktion auf allen 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 schließen wir explizit aus genau wie alle $SCALAR$ Types. Jeder Knoten definiert nun in seinem $fields$ Eintrag zu welchen Feldern er Beziehungen hat. Hierbei muss unterschieden werden, dass ein Eintrag entweder vom Type $OBJECT$ oder $LIST$ sein kann, um zulässig zu sein. Sollte es sich um einen $LIST$ Eintrag handeln müssen wir prüfen, von welchem Type die $LIST$ ist. Wenn ein Knoten nun unsere Bedingungen erfüllt, so wird dieser dem Graphen hinzugefügt und auf ihm selbst wird *buildGraph* ausgeführt. So erlangen wir die gesamte Graphstruktur da ausgehend von Query jeder erreichbare Knoten hinzugefügt wird und dann von diesem Knoten eben wieder alle erreichbaren Knoten hinzugefügt werden.

```
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 Graphen aufspannt

7.2.2 Pfade aus Graph bilden

Der Graphhandler implementiert verschiedene Abdeckungskriterien. Das Tool benötigt im späteren Verlauf lediglich 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

Da wir jedoch in Kapitel 4.6 feststellten, dass die PrimePfade-Abdeckung am geeignetsten ist, erklären wir dieses Kriterium hier. Die PrimePfad Abdeckung ist durch die Funktion $generate_prime_paths$ implementiert. Diese Funktion verknüft dabei allerdings nur zwei andere Funktionen.

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

Abbildung 7.4: valide PrimePfad Generierung

Da PrimePfade nicht im Query Knoten starten müssen, rufen wir zuerst die Funktion *get_prime_paths* auf um eine Liste aller PrimePfade zu bekommen. Da jedoch ein Pfad stets im Query-Knoten starten muss, rufen wir anschließend *shortest_path_to_prime*. Diese Funktion ermittelt den kürzesten Weg vom QueryKnoten zum Startknoten des PrimePfades. So können wir sicherstellen, dass die generierten Pfade stets valide Testpfade sind und dennoch alle PrimePfade abdecken. Die Funktion *get_prime_paths* ist in Abbildung 7.5 dargestellt.
Sie nutzt die Gegebenheit, dass Anfragen zwar im Query-Knoten starten müssen, jedoch jeder andere Knoten des Graphens ein potentieller Endknoten ist. Jeder Knoten wird dabei einmal als Endknoten behandelt und die einfachen Pfade von Query-Knoten zu diesem Knoten werden ermittelt. Anschließend werden die Pfade so gefiltert, dass nur die längsten Pfade, die keine Unterpfade haben zurückgegeben werden. Dies entsprich laut Definition 15 einem PrimePfad und der Algorithmus ist Deckungsgleich mit [5, Finding Prime Test Paths S.39].

```python
def get_prime_paths(G, start_node):
    all_prime_paths = []
    nodes = list(G.nodes())

    for end_node in nodes:
        if end_node == start_node:
            continue
        simple_paths = list(nx.all_simple_paths(G, source=
           start_node, target=end_node))

        # Ein Pfad ist prime, wenn er kein Teilpfad ist
        prime_paths_for_end_node = []
        for path in simple_paths:
            is_prime = True
            for other_path in simple_paths:
                if set(path).issubset(set(other_path)) and
                   path != other_path:
                    is_prime = False
                    break
            if is_prime:
                prime_paths_for_end_node.append(path)

        all_prime_paths.extend(prime_paths_for_end_node)
    return all_prime_paths
```

Abbildung 7.5: PrimePfad Generierung

7.2.3 Querys aus Pfad ermitteln

Die entwickelten Pfade werden in diesem Schritt nun in Querys umgewandelt, sodass diese an die GraphQL-API gestellt werden können. Eine Query beginnt in GraphQL immer im Query-Knoten und so beginnen auch alle unseren ermittelten Pfade in diesem Knoten. Da wir die Wahrscheinlichkeit erhöhen wollen, dass ein Pfad gut abgedeckt wird, generieren wir pro Pfad nicht einen Test sondern führen eine Variable $testPerPath$ ein, die festlegt, wieviele Querys pro Pfad generiert werden sollen. Standardmäßig legen wir fest: $testPerPath = 5$. Um Querys aus einem Pfad zu erstellen wurde der *querygenerator* entwickelt. Der Querygenerator hat die Methode *pathToQuery* welche den Pfad, ein Typedict und den Graphen erwartet. Das Typedict ist ein Python-Dict, dass alle Informationen über das Schema enthält. Die Funktion erstellt nun eine Query indem der angegebene Pfad abgelaufen wird. Diese Funktionalität setzt die rekursive Funktion $resolvePathTillOnlyScalarTypesOrEnd(path, typedict, graph, query =)$ um die in Abbildung 7.6 dargestellt ist.
Die Funktion arbeitet dabei so, dass sie für jeden Ausgangspunkt einer Kante alle $SCALAR$ Felder hinzufügt. So wird sichergestellt, dass alle einfachen Felder eines Objektes abgefragt werden. Es kann so validiert werden, 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 die Argumente generiert. Da Argumente nur $SCALAR$ Types oder Aggregationen von $SCALAR$ Types sein können, benötigen wir Datengeneratoren für die Standarddatentypen. Die Zuweisung der Datengeneratoren geschieht hierbei mit der Funktion $resolveArg$. Benötigt ein $OBJECT$ Feld nun Argumente, so werden diese der Query hinzugefügt indem mit $resolveArg$ die Argumente zur Verfügung gestellt wurden. Anschließen wird die Kante aus der Liste des Pfades 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 dann 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 sehr wahrscheinlich, dass der Test erfolgreich sein wird, ohne dass der eben entwickelte Pfad wirklich komplett getestetet wird.

```
1  def resolvePathTillOnlyScalarTypesOrEnd(path, typedict,
       graph, query=""):
2      if path:
3          edge = path.pop(0)
4      else:
5          return query
6      edge_data = graph[edge[0]][edge[1]]["data"]
7      if len(edge_data['args']) < 1:
8          if edge_data["type"]["kind"] == "SCALAR":
9              query = query + " " + edge_data["name"] + " "
10         else:
11             query = query + " " + edge_data["name"] + " { "
                   + addScalarTypes(edge_data["type"], typedict,
                    edge_data["name"]) + " " +
                   resolvePathTillOnlyScalarTypesOrEnd(path,
                   typedict, graph, query) + " } "
12     else:
13         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
```

Abbildung 7.6: Pfadumwandlung in Query

7.2.4 Tests ausführen & Testdatei generieren

Bevor wir die Tests ausführen, werden diese im PyTest-Format gespeichert. Hierzu wird eine Datei angelegt, die alle nötigen Imports enthält. Anschließend wird für jede Query ein eigener Test angelegt und die Auswertung festgeschrieben. Dafür existiert die Funktion *generateTestFromQuery* im Modul Testgenerator. Ein so erstellter PyTest ist in Abbildung 7.7 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
```

Abbildung 7.7: PyTest einer Query

Nachdem mit den PyTests die Querys reproduzierbar gemacht wurden, führen und werten wir diese aus. Hierbei nutzen wir den Code aus Abbildung 7.8. 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()
```

Abbildung 7.8: Ausführen einer Testquery

Um die Tests auszuführen, senden wir alle zuvor generierten Querys an das SUT. Die Antworten werden zuerst gespeichert und dann später ausgewertet.

7.2.5 Testauswertung

Die Testauswertung erfolgt, wie zuvor gesehen, in zweierlei Arten. Einerseits werden Tests ad-hoc ausgeführt. Andererseits wird eine Datei mit PyTests generiert. Die Auswertung der Testquerys in der PyTest Datei erfolgt nach PyTest Standard, dabei werden Hinweise geliefert falls etwas abweicht von der Erwartung. Eine Auswertung der Testquerys die direkt ausgeführt wurden erfolgt nach einer Kategoriesierung. Die Kategoriesierungen sind **Good_Test**,**Perfect_Test**,**malformed_Test** und **confirmed_failed_Test**. Ein **Good_Test** ist ein Test, der keinen Fehler erzeugt hat allerdings ist die erwartete Pfadlänge von der Response nicht erfüllt worden, 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 Generierungsfehler vom Prototypen. **confirmed_failed_Test** sind Tests bei denen wir tatsächlich einen Fehler bekommen. 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 Abbildung 7.9 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
```

Abbildung 7.9: Auswertung der Antworten

7.3 Zusammenfassung der Implementation

Der Python Prototyp stellt eine Implementierung der Methode aus Kapitel 6 dar. Wie wir später sehen werden 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 zu beeinträchtigen.

8 Auswertung und Vergleich mit Property-based Testing

8.1 Vergleichmetriken

Bevor wir einen tatsächlichen Vergleich beider Methoden durchführen werden erst einmal die Metriken eingeführt, in denen sich Verglichen wird. Hierdurch wird einfacher verständlich welche Punkte miteinander verglichen werden. Wir nutzen 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. Hierbei wurden zwei Forschungsfragen entwickelt.

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 2 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] Unser entwickelter Prototyp soll in exakt dem gleichen Umfeld seine Tests generieren. Wir erwarten, dass wir möglichst dieselben Fehler finden wie die ursprüngliche Methode und ideal wäre es, wenn wir mehr und neue Fehler finden würden. Beide Forschungsfragen werden im folgenden noch einmal näher erläutert da diese ein wenig spezialisiert sind und Wissen über Methode ist wichtig um die Ergebnisse korrekt einordnen zu können.

8.1.2 Fehlerfindungskapazitäten

Mit Fehlerfindungskapazitäten ist gemeint wie zuverlässig die Methode tatsächliche 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, gibt es eine Test API die initial mit bekannten Fehlern versehen wird. Die *Property-based Methode* hat 11 von 15 Fehlern im speziell vorbereiteten System gefunden. Bei GitLab wurden 4 Bugs im Query-Bereich gefunden. Unsere entwickelte Methode 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 hier 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 gilt:

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

8.2 Threats to Validity / Limitierungen

Bevor wir mit dem eigentlichen Vergleich beginnen muss noch kurz eingeordnet werden, inwiefern die Experimente zu betrachten sind und unter welchen Voraussetzungen der Vergleich geschieht.

8.2.1 Argumentgeneratoren

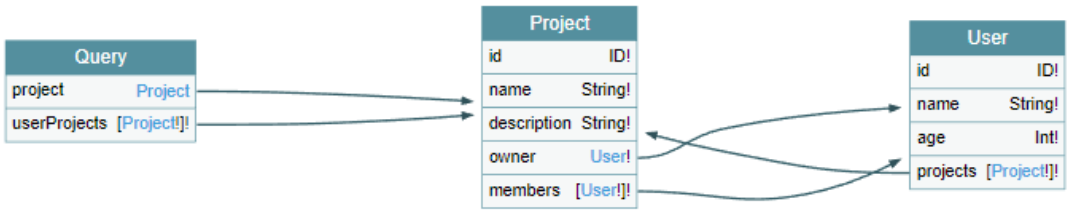
Wie in Kapitel 6.2.3 erwähnt, ist es wichtig, dass GraphQL für jede Funktionen einen Wert ungleich *null* bekommt, sodass der Pfad weitergegangen werden kann und die Funktionen in diesem getestet werden um die tatsächliche Abdeckung zu erhöhen. Um die Wahrscheinlichkeit zu erhöhen, dass Argumentgeneratoren ein Argument zurückliefern, dass zum SUT passt, wurden diese teilweise angepasst. Wir verletzen hierbei nicht die Vergleichbarkeit der Arbeiten, da diese Anpassung der Argumengeneratoren in [6] auch stattfand [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 beispielhafte Anpassung wäre hier nun, dass wir den Generator für den Type ID so anpassen, dass er nur Argumente für ID zurückliefert die ein Zahlenwert sind in einem gewissen Bereich der durch das SUT abgebildet wird.

8.3 Fehlerfindungskapazitäten

Zuerst wollen wir die Fehlerfindungskapazitäten des Prototypens beweisen. Hierfür nutzen wir die beiden, zuvor benannten, Testsyteme GraphQL-Toy (eine experimentelle GraphQL-Implementierung) und GitLab in der Version 12.6.3. Ziel ist es mindestens die Fehler zu finden die vom *Property-based Testtool*[6] gefunden wurden. Idealerweise wollen wir jedoch sogar mehr Fehler finden.

8.3.1 GraphQL-Toy

Das Testssystem GraphQL-Toy hat ein simples Schema in dem nur drei *OBJECT* Typen exisitieren. Diese sind *Query*, *Project* und *User*. Das Schema hat folgende Struktur:



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 lässt sich im Anhang unter *GraphQL-Toy Implementation mit Bugs* 12 finden.

Bug 1 - SyntaxFehler

Einfache 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 wäre zum Beispiel folgender Code:

```
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    }
```

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

Bug 2 - Falscher Objekttyp

Objektfehler sind ein wenig unoffensichtlichere Fehler. Hierbei gibt der Code ein Objekt zurück, dass nicht der definierten Struktur im Schema entspricht. GraphQL wird hierfür dann einen Fehler erzeugen da die Daten eben nicht zum Schema passen.

```
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    }
```

Um diesen Fehler ausführen zu können ist es wichtig, dass das Feld auch abgefragt wird. Sollte das Feld ein Argument benötigen, so muss dieses passen, sodass auch wirklich ein Objekt abgefragt wird und dann der falsche Type zrückgegeben 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 um die z.B. genutzt wird um einen bestimmten Eintrag eines Arrays zu bekommen. Inputvalidierung ist also von Nöten.

```
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    }
```

Es ist hier möglich, ohne jegliche Prüfung einen Key anzugeben. Ist ein Resolver wie hier implementiert so ist es erlaubt in der Query jeglichen String anzugeben. Es ist also sehr wahrscheinlich, dass beispielsweise ein IndexOutOfBound Fehler auftreten kann.

Mit dem Testtool nach [6, Property-based Testing] konnten 73% der Fehler, also 11 der 15 Fehler gefunden werden. Unser entwickeltes Testtool schaffte auf derselben API auch eine Entdeckung von 11 Fehlern. Wir konnten also dieselbe Fehlerfindung erreichen wie das Property-based Tool. Bemerkenswert hierbei ist allerdings, dass das Property-based Tool hierfür 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 Kontrast dazu konnte unsere hier entwickelte Methode mithilfe von nur 2 PrimePfaden eine PrimePath Coverage erreichen. Hierzu wurden für jeden Pfad 5 Testquerys entwickelt. Es war somit möglich, alle 11 Fehler zu finden. Bemerkenswert war, dass zwei perfekte Querys ermittelt wurden. Das Testtool fand diese beiden Querys hierfür:

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

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

Mithilfe dieser Querys konnte 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 Querys zur unterliegenden Datenstruktur gepasst hat und wir somit die eine tatsächliche Testausführung haben und nicht nur einen initialen *null* Wert der die Query sofort erfolgreich sein lässt. Die 4 nicht gefundenen Fehler sind dieselben Fehler wie diese, die *Property-based Testing* [6, vgl. RQ.2] nicht finden konnte. Dies 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. Wir wurden also limitiert vom Black-Box Ansatz, da das Verhalten für unseren Prototypen als fehlerfrei gilt es aber eigentliche eine fehlerhafte Ausgabe ist. In diesem Beispiel hat unser Prototyp die selben 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 verwandt, um an einem Industriereifen Projekt die Methode zu evaluieren [6, vgl. Experiment]. Wir wollen unseren Prototypen auch an diesem System testen. 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 wir uns das GraphQL-Schema von GitLab ansehen welches in Abbildung 8.1 gezeigt wird. Eine hochauflösendere Version ist im GitHub verfügbar. Das Schema ist sehr komplex und stark zyklisch.

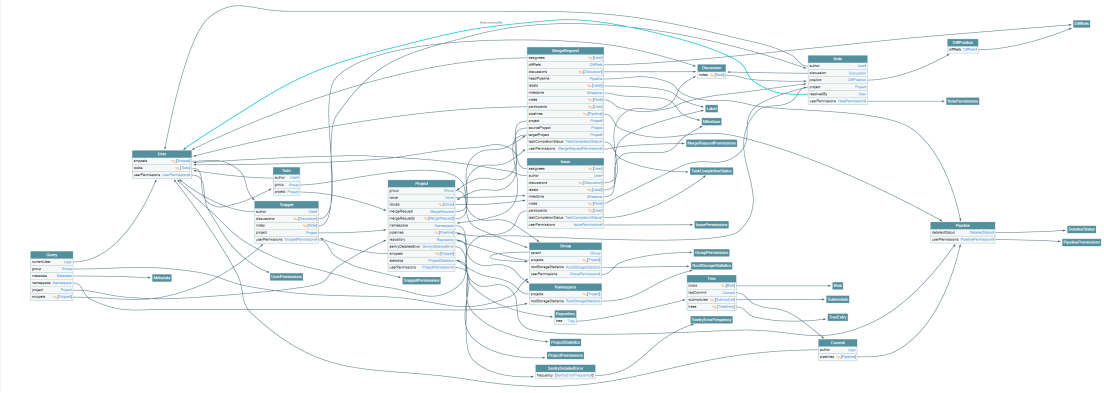


Abbildung 8.1: GitLab GraphQL-Schema

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 ein Fehler 4) aber ein String mit leerem Zeichen führt zum Fehler: `e\u0000\` Die Fehler wurden gefunden durch folgende Querys:

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

Listing 8.1: Fehler 1[36]

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

Listing 8.2: Fehler 2[37]

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

Listing 8.3: Fehler 3[38]

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

Listing 8.4: Fehler 4[39]

Alle vom Property-based Tool gefunden Fehler wurden durch unseren Prototypen gefunden mit den Querys
Query 1, Query 2, Query 3 und Query 4.
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, dass im GitLab 50 User anlegt und jedem User einige Projekte, Commit, MergeRequests usw. zuordnet. Das Population-Skript kann im [40, Github] gefunden werden. Da die Querygenerierung stets im Query-Knoten beginnt und die PrimePaths gefunden werden sollen,

ergeben sich in diesem Schema sehr viele ähnliche Querys. 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 [41, GitHub]. Mit der Maßgabe, dass wir pro Pfad 5 Tests erzeugen wollen 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 daran, 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. Hier verringert sich einfach das Risiko, dass Eingabeargumente generiert werden, die keine zugrundeliegenden Daten haben und somit die Pfadausführung verhindern. Ein Beispiel hierfür ist der Pfad $Query- > Project- > MergeRequest- > Time$

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

durch gutes Mocken des FullPath Argumentgenerators war es möglich, eine Query zu generieren, die somit passende Argumente generiert hat um für eine ideale Testausführung (d.h. Länge Ergebniss = Länge Testpfad) zu sorgen. Generell zeigt sich sehr schnell, dass bei unserer Methode ähnliche Limitierungen wie im Property-based Testing auftreten. So ist eine Anpassung 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 wir ähnliche Argumentgeneratoren verwendet haben wie in Property-based Testing, haben wir auch deren Limitierungen von mangelndem Domänenwissen [6, vgl. S.8]. Wir generieren zwar sehr viele Tests die PrimePfad Abdeckung umsetzen. Wenn jedoch die Generierung der Tests auf Zufall basiert, so können wir auch nicht garantieren, dass unsere Eingabeargumente passend sind und ein valides Ergebniss zurückgeben. Wird kein Ergebniss zurückgegeben, so folgert GraphQL, dass spätere Funktionen nicht ausgeführt werden müssen und somit werden diese auch nicht getestet. Es ist möglich, dieses Problem zu beheben, hierzu 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 exisitiert, starten wir mit einem sehr einfachen und intuitiven Ansatz* [6, vgl. B. Measuring Schema Coverage]. In der Tat ist das vorgestellte Abdeckungskriterium ein sehr einfaches Kriterium. Es lässt zum Beispiel die Beziehungen zwischen allen Knoten aus und beachtet nur, dass alle Knoten inbegriffen sind mit allen Feldern. Hiermit entspricht das definierte Abdeckungskriterium der Kantenabdeckung nach Definition 3.6.1. Denn alle Knoten müssen abgedeckt sein und alle Kanten ausgehend von den Knoten. 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 wir später sehen werden 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 mehrere Iterationen benötigt um diese zu erreichen. Der Ansatz der PrimePfad-Abdeckung in unserem Prototypen stellt sicher, dass die generierten Pfade PrimePfade sind. Wir stellen somit sicher, dass stets die Abdeckung erfüllt ist. Gleichzeitig nutzen wir ein stärkeres Abdeckungskriterium auf unbeschnittenen Graphen. Man kann also folgern, dass unsere Methode eine starke Verbesserung der theoretischen Abdeckung erzielt hat. Die Limitierung der zufälligen Argumentgeneratoren behindern eine tatsächliche Umsetzung der theoretischen Abdeckung. Allerdings ist diese Limitierung potentiell 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 Defintion 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. Wir können keine Aussage darüber machen, ob die generierten Querys von *Property-based Testing*[6] diese Coverage erfüllen denn bei der Querygenerierung spielt es keine Rolle ob dieses Measurement 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 Edge-Coverage 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. Wir berechnen PrimePfade, diese sind in einem stärkeren Abdeckungskriterium enthalten als die Kantenabdeckung. Dadurch ergibt sich, dass die Tests, die vom hier entwickelten Prototyp erstellt werden, stets auch die Kantenabdeckung erfüllen. Die Abdeckung muss nun also nicht mehr durch hinreichend viele Tests sichergestellt werden. Ein einziger Durchlauf reicht aus, um sicherzustellen, dass die gewünschte Coverage 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 auch nur geprüft, ob die Felder in der Anfrage existieren jedoch nicht, ob die Antwort diese auch enthält. Um dies messbar zu machen haben wir zuvor die Pfadlängen eingeführt. Hierdurch konnten wir zeigen, dass die generierten Querys in Teilen die Abdeckung auch tatsächlich umsetzen.

8.4.2 GitLab Schema Coverage

Das GitLab Schema ist wesentlich komplexer. 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 hier die Frage inwiefern überhaupt das Schema überdeckt werden kann. Laut Paper hat sich ein Rekursionslimit von 4 als hinreichend ausgezeichnet [6, vgl. Table 1] und wurde auch so im Code übernommen. Ein Rekusionslimit von 4 bedeutet, dass die maximale zu erreichende Pfadlänge des Testpfades 4 ist. Da das GitLab-Schema aber nun einen Graphen aufspannt, der durchaus wesentlich längere Pfade als 4 hat, ist es fragwürdig

wie die 100% Schema-Coverage in [6, Table 1] berechnet wurden. 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 auch zuvor erwähnt, erzeugt unser hier entwickelter Prototyp Tests, die PrimePfad-Abdeckung umsetzen und somit ein stärkeres Abdeckungskriterium ist. Wir führen die Pfadgenerierung auf dem gesamten Graphen aus und erhalten, wie zuvor erwähnt, über 40.000 Pfade zurück die nötig sind um eine PrimePath-Coverage 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 ein 100% Edge-Coverage zu erreichen [6, vgl. Table 1] so sehen wir, dass 10.000 Tests nicht reichen können, wenn über 40.000 PrimePaths existieren. Insbesondere sei hierbei angemerkt, dass die Pfad- & Testgenerierung auf dem GitLab-Schema keine allzu rechenintensive Aufgabe war. Die Berechnung der Querys geschah auf einem hardwaretechnisch ähnlichem Level wie in Property-based Testing verwandt. [6, vgl. Experimental Setup]. Hier wurde die Aussage getroffen, dass ein [6, Tiefensuchen Ansatz nicht skaliert und deswegen ein iterativer Ansatz zu präferieren ist]. Der hier entwickelte Prototyp zeigt das Gegenteil.

8.5 Zusammenfassung der Experimente

Wir konnten in beiden Experimenten zeigen, dass unser Prototyp dieselben Fehler findet wie der Prototyp aus Property-based Testing. Es wurde gezeigt, dass die theoretische Abdeckung eines GraphQL-Schemas mit unserem Prototypen immens gesteigert wird

ohne, wie in [6] behauptet, die Berechnungszeit signifikant zu erhöhen. Während wir die theoretische Abdeckung des Graphens erhöhen konnten, 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 eben nicht komplett ausgeführt wird. Eine Anpassung der Argumentgeneratoren an das jeweilige SUT ist sinnvoll und verbessert die Pfadlängen der Tests. Unser Prototyp leidet unter der selben Limitierung wie das *Property-based Testing* indem wir mehr Domänenwissen über die zugrundeliegenden Resolver und Daten haben müssen um sinnvolle Test zu generieren. Dies ist Analog zu [6] 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 ergründen wir in Kapitel 9

9 Zukünftige Arbeit

Einige bereits angesprochene Punkte bieten Möglichkeiten für eine Erweiterung der hier geleisteten Arbeit. Während hier 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 Querys. Ziel ist es nun, den Zufall möglichst weit zu begrenzen oder aber die erlangten Ergebnisse intelligenter zu nutzen Wir wollen im folgenden zwei Ansätze vorstellen, die eine praktische Testausführung zuverlässiger und präziser machen 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 haben wir den BlackBox Ansatz ein wenig abgeschwächt und zu einem GreyBox-Ansatz verändert, indem wir die Argumentgeneratoren mit Domänenwissen an das jeweilige SUT angepasst haben, sodass die zufällige Argumentgenerierung mit höherer Wahrscheinlichkeit ein Argument liefert, dass dem Test eine bessere, tatsächliche Abdeckung liefert. Idealerweise 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 nun möglich, die Argumentgeneratoren automatisch anzupassen, sodass sich diese am Schema und den zugrunde-liegenden Daten orientieren. Außerdem wäre eine Code-Analyse möglich, die dazu führen kann, dass Testcases 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, dass die tatsächliche Pfadabdeckung stark erhöht wird.

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 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 uns 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 Querys mit Zufallsargumenten erstellt die einen Teilpfad des Testpfades bilden und erst wenn der Teilpfad des Testpfades abgedeckt ist, wird mit dem nächsten adjazenten Knoten fortgefahren, sodass die tatsächliche Pfadabdeckung mit der theoretischen Pfadabdeckung übereinstimmt.

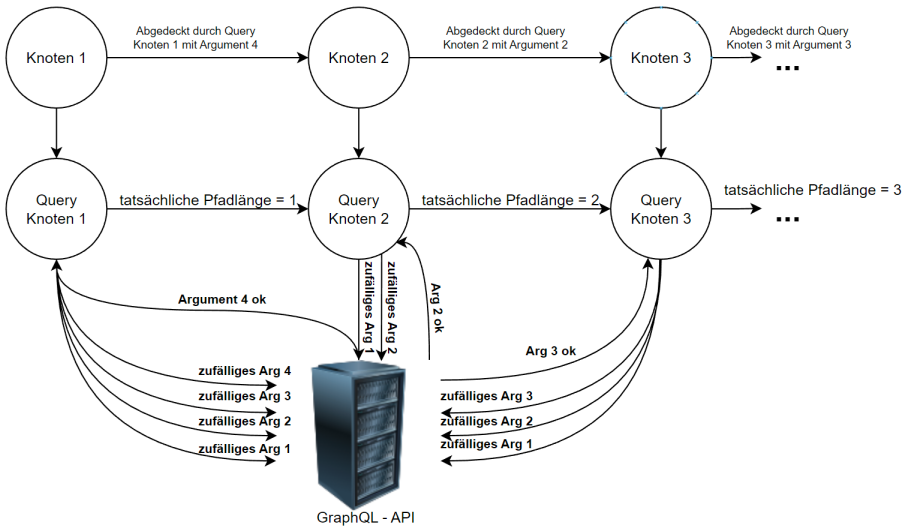


Abbildung 9.1: Beispielablauf einer adaptiven Generierung

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. 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

Unsere Behauptung, dass wir die Methode des *Property-based Testings* um einen besseren Pfadpfindungsalgorithmus als ein simples, zufälliges, begrenztes raten verbessern können hat sich als wahr herausgestellt. Indem wir einen theoretischen Rahmen geschaffen haben, der zeigt, dass Graphabdeckunskriteren auf ein GraphQL-Schema anwendbar sind, konnten wir eine Methode entwickeln, die für die Testentwicklung hinreichend ist. 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 wir Fehler finden konnten. Es wurden Schwachstellen offen gelegt und Ansätze entwickelt, die zur Weiterarbeit anregen. Verwandte Arbeiten wurden betrachtet und wir konnten dabei feststellen, dass ein Ansatz wie in dieser Arbeit vorher noch nicht betrachtet wurde. Generell lässt sich sagen, dass GraphQL eine zurecht stetig wachsende Technologie ist und Arbeiten wie diese hier dazu beitragen, dass die Popularität von GraphQL wächst da die bisherigen Nachteile von GraphQL gegenüber REST so stückweise aufgelöst werden und die großen Vorteile herausstechen. Für mich persönlich hat diese Arbeit ein tiefgreifendes Verständnis von GraphQL gebracht und ich werde diese Technologie in Zukunft wesentlich häufiger Nutzen.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Onlineressourcen wurden am 14. August 2023 auf ihre Verfügbarkeit hin überprüft.